POLITECHNIKA POZNAŃSKA

Patryk Dąbrowski 100584
Aleksander Kędzierski 98875
Paweł Lampe 99277
Mateusz Sikora 99615

Platforma zarządzania zdarzeniami na urządzeniach mobilnych if{y}

Praca inżynierska

Promotor: dr inż. Jerzy Błaszczyński

Poznań, 2014

Spis treści

Rozdział 1

Wstęp

1.1 Motywacje

Współczesne urządzenia mobilne dysponują ogromnym zbiorem modułów generujących zdarzenia. Zdarzeniami mogą być przykładowo: zmiana poziomu naładowania baterii, przychodzące połączenie czy odczyt danych akcelerometru. Niektóre zdarzenia obsługuje właściciel urządzenia, podczas gdy innymi zajmują się dedykowane aplikacje. Aby przykładowo wyłączyć moduł WiFi po rozłączeniu z siecią, należy pobrać aplikację *WiFi Auto Turn Off*. Aplikacji podejmujących akcje w reakcji na zdarzenia jest na rynku bardzo dużo.

Wyzwanie mające na celu zbudowanie pojedynczej aplikacji do zarządzania zdarzeniami oraz akcjami zostało obecnie podjęte przez zaledwie kilka firm. Największą z nich jest Microsoft. Wszelkie implementacje znane twórcom niniejszej pracy są komercyjne. Mają też inne wady, takie jak wymagane logowanie przez Facebooka.

Pomysł będący fundamentem tej pracy wybiega nieco dalej niż istniejące rozwiązania. Największą innowacją w if{y} jest obsługa zdarzeń dotyczących grup użytkowników – akcja na jednym urządzaniu może spowodować reakcję na pozostałych. Poza tym nie ma to być jedna aplikacja, a cała otwarta platforma, w ramach której zaawansowani użytkownicy mogą mieć własny serwer obsługujący zdarzenia grupowe lub przystosowaną do swoich potrzeb aplikację. Takie rozwiązanie powinno zadowolić programistów używających telefonów z systemem Android, a w szczególności tych, którzy nie chcą zagłębiać się w tę platformę aby uzyskać prostą aplikację automatyzującą jakąś czynność.

Jeśli uda się udostępnić serwis do dzielenia się zdefiniowanymi akcjami, także użytkownicy nie będący programistami będą mogli korzystać z platformy.

1.2 Cele i zakres pracy

Podstawowym celem niniejszej pracy jest stworzenie otwartoźródłowej (ang. open source) biblioteki ułatwiającej obsługę zdarzeń generowanych przez podzespoły urządzenia mobilnego. Podejście takie tworzy warstwę abstrakcji nad systemem operacyjnym. Ma to potencjalnie zapewnić przenośność kodu, pisanego przy użyciu biblioteki, pomiędzy dowolnymi systemami operacyjnymi. Niniejsza praca zakłada implementację biblioteki tylko dla systemu Android.

Pozostałe cele to stworzenie prostej aplikacji mobilnej prezentującej możliwości biblioteki oraz dwóch internetowych aplikacji pomocniczych. Fakt implementacji biblioteki tylko dla systemu Android wymusza implementację aplikacji mobilnej również dla tego systemu operacyjnego.

Implementacja aplikacji pomocniczych jest nieunikniona ze względu na potrzebę centralizacji niektórych danych poza obrębem pojedynczego urządzenia.

Reasumując, należy stworzyć platformę, w skład której wchodzą: biblioteka dla systemu Android, aplikacja mobilna dla systemu Android oraz dwie aplikacje internetowe.

1.3 Podział prac

Podział obowiązków został dokonany na podstawie kompetencji poszczególnych osób. Częściami platformy związanymi ściśle z systemem Android zajęli się Aleksander Kędzierski oraz Mateusz Sikora. Częściami związanymi z technologiami internetowymi zajęli się Patryk Dąbrowski i Paweł Lampe. Nad kwestiami wspólnymi dla obu części pracował cały zespół.

Ostatecznie, poszczególne osoby dokonały:

- Patryk Dąbrowski
  - Implementacji serwera zdarzeń grupowych
  - Projektowania protokołu komunikacji serwera z aplikacją
- Aleksander Kędzierski
  - Zaprojektowania i implementacji większości biblioteki if{y}, w szczególności związanej z obsługą Recept.
  - Implementacji części aplikacji appIFY.
  - Współprojektowania protokołu komunikacji serwera z aplikacją
- Paweł Lampe
  - Zaprojektowania i implementacji Targowiska
  - Administracji serwerem z systemem Linux
- Mateusz Sikora
  - Implementacji części biblioteki if{y}
  - Implementacji większości aplikacji appIFY

Rozdział 2

Wymagania

W niniejszym rozdziale zebrano wymagania funkcjonalne oraz pozafunkcjonalne. Stanowią one opis pożądanego zachowania platformy. Wszystkie wymagania powstały w procesie zbierania wymagań, w który zaangażowany był promotor pracy oraz grupa inżynierów ze specjalizacji TWO: Grzegorz Hauska, Tomasz Kuliński, Wojciech Mioduszewski i Karol Tatała.

2.1 Wymagania funkcjonalne

2.1.1 Przypadki użycia platformy

ID:	UC1
Nazwa:	Logowanie użytkownika
Scenariusz główny:	**1** Użytkownik wprowadza login oraz hasło. **2** System weryfikuje użytkownika.
Scenariusz alternatywny i rozszerzenia:	**1a** Błędne dane. **1a.1** System wyświetla informację o błędnych danych. **1a.2** Powrót do 1.

ID:	UC2
Nazwa:	Pobieranie Recept
Scenariusz główny:	**1** Użytkownik otwiera menu zdalnego źródła Recept. **2** System wyświetla listę Recept. **3** Użytkownik wybiera Receptę do pobrania. **4** System wyświetla szczegółowe dane Recepty. **5** Użytkownik wybiera opcję download. **6** System pobiera Receptę.

ID:	UC3
Nazwa:	Aktywowanie Recept
Scenariusz główny:	**1** Użytkownik wybiera opcję aktywowania Recept. **2** System wyświetla Recepty do aktywacji. **3** Użytkownik wybiera Recepty do aktywacji. **4** System aktywuje Recepty. **5** System wyświetla stosowną informację.
Scenariusz alternatywny i rozszerzenia:	**2a** Nie ma żadnych Recept do aktywowania. **2a.1** System wyświetla informację o braku wymaganych Recept. **2a.3** Koniec.

ID:	UC4
Nazwa:	Dezaktywowanie Recept
Scenariusz główny:	**1** Użytkownik wybiera opcję dezaktywowania Recept. **2** System wyświetla aktywne Recepty. **3** Użytkownik wybiera Recepty do dezaktywacji. **4** System dezaktywuje Recepty. **5** System wyświetla stosowną informację.
Scenariusz alternatywny i rozszerzenia:	**2a** Nie ma żadnych aktywnych Recept. **2a.1** System wyświetla informację o braku aktywnych Recept. **2a.3** Koniec.

ID:	UC5
Nazwa:	Zapraszanie do grupy
Warunki początkowe:	Użytkownik jest zalogowany do systemu.
Scenariusz główny:	**1** Użytkownik 1 wchodzi w opcje zarządzania grupami. **2** System wyświetla listę grup, w których użytkownik 1 jest administratorem. **3** Użytkownik 1 wybiera grupę. **4** Użytkownik 1 zaprasza Użytkownika 2 do grupy.

ID:	UC6
Nazwa:	Akceptacja zaproszenia
Warunki początkowe:	Użytkownik jest zalogowany do systemu.
Scenariusz główny:	**1** Użytkownik wchodzi w opcje zarządzania zaproszeniami. **2** System wyświetla listę grup, do których Użytkownik jest zaproszony. **3** Użytkownik wybiera jedno z zaproszeń. **4** Użytkownik akceptuje zaproszenie.

ID:	UC7
Nazwa:	Tworzenie grupy
Warunki początkowe:	Użytkownik jest zalogowany do systemu.
Scenariusz główny:	**1** Użytkownik wchodzi w opcje tworzenia grupy. **2** Użytkownik wpisuje nazwę nowej grupy. **3** Użytkownik tworzy grupę.
Scenariusz alternatywny i rozszerzenia:	**3a** Nazwa grupy nie jest unikalna. **3a1** System informuje o tym fakcie użytkownika. **3a1** Powrót do 2.

2.2 Wymagania pozafunkcjonalne

ID:	PF01
Nazwa:	System operacyjny dla aplikacji mobilnej
Kategoria:	Środowisko
Priorytet:	Wysoki
Opis:	Systemem operacyjnym aplikacji mobilnej jest Android w wersji minimum 2.1.

ID:	PF02
Nazwa:	Środowisko uruchomieniowe dla aplikacji serwerowej
Kategoria:	Środowisko
Priorytet:	Wysoki
Opis:	Aplikacja serwerowa powinna działać na maszynie wirtualnej Java.

ID:	PF03
Nazwa:	Używane technologie
Kategoria:	Technologie
Priorytet:	Wysoki
Opis:	Wykorzystane technologie nie mogą być płatne.

ID:	PF04
Nazwa:	Zunifikowane środowisko programistyczne
Kategoria:	Narzędzia
Priorytet:	Wysoki
Opis:	Programiści muszą zdecydować się na wspólne narzędzie do redagowania kodu (np. Eclipse).

ID:	PF05
Nazwa:	Ograniczone zużycie energii urządzenia mobilnego
Kategoria:	Wydajność i niezawodność
Priorytet:	Średni
Opis:	Działanie aplikacji nie powinno w znaczącym stopniu skracać czasu pracy urządzenia na baterii.

ID:	PF06
Nazwa:	Ograniczone zużycie zasobów urządzenia mobilnego
Kategoria:	Wydajność i niezawodność
Priorytet:	Średni
Opis:	Aplikacja nie powinna spowalniać działania innych aplikacji.

ID:	PF07
Nazwa:	Czas reakcji aplikacji na zdarzenie
Kategoria:	Wydajność i niezawodność
Priorytet:	Wysoki
Opis:	Aplikacja powinna reagować na zdarzenia lokalne w mniej niż 2 sekundy.

ID:	PF08
Nazwa:	Zgodność ze standardami kodowania dla języka Java
Kategoria:	Zgodność ze standardami
Priorytet:	Wysoki
Opis:	Zarówno kod aplikacji mobilnej, jak i serwerowej powinien być redagowany zgodnie ze standardami dla języka Java.

ID:	PF09
Nazwa:	Przechowywanie haseł
Kategoria:	Bezpieczeństwo
Priorytet:	Wysoki
Opis:	Szyfrowane zapamiętywanie hasła użytkownika.

ID:	PF10
Nazwa:	Przechowywanie haseł
Kategoria:	Bezpieczeństwo
Priorytet:	Wysoki
Opis:	Przechowywanie skrótu hasła na serwerze.

ID:	PF11
Nazwa:	Długość Recepty
Kategoria:	Użyteczność
Priorytet:	Wysoki
Opis:	Długość kodu Recepty nie powinna w średnim przypadku przekraczać 100 linii.

ID:	PF12
Nazwa:	Tworzenie Recept
Kategoria:	Użyteczność
Priorytet:	Wysoki
Opis:	Proces tworzenia Recepty nie powinien zajmować więcej niż 15 minut.

ID:	PF13
Nazwa:	Dystrybucja Recept
Kategoria:	Użyteczność
Priorytet:	Średni
Opis:	Proces dystrybucji Recepty nie powinien zajmować więcej niż 5 minut.

ID:	PF14
Nazwa:	Bezpieczeństwo Recept
Kategoria:	Bezpieczeństwo
Priorytet:	Wysoki
Opis:	Recepta powinna korzystać tylko z biblioteki if{y} oraz pakietów narzędziowych Java.

Rozdział 3

Zarządzanie zdarzeniami na urządzeniach mobilnych

Jak wspomniano w rozdziale 1, zarządzanie zdarzeniami na urządzeniach mobilnych doczekało się zaledwie kilku implementacji. Tematyka jest więc otwarta zarówno na pomysły jak i terminologię.

W niniejszym rozdziale wprowadzono przede wszystkim pojęcia które pojawiają się w dalszych częściach pracy. Dokonano również przeglądu istniejących rozwiązań pod kątem mocnych oraz słabych stron.

3.1 Definicja pojęć

- Podfunkcjonalność (ang. Feature) – Część biblioteki zapewniająca Receptom dostęp do podzbioru funkcjonalności Androida.
- Zdarzenie (ang. Event) – Zmiana stanu systemu, która powoduje uruchomienie kodu Recepty.
- Recepta (ang. Recipe) – Napisany przez użytkownika fragment kodu opisujący, co ma się zdarzyć po spełnieniu pewnych warunków.
- Targowisko (ang. Market) – Aplikacja internetowa pozwalająca tworzyć i pobierać Recepty.
- Aplikacja Kliencka – Aplikacja androidowa wykorzystująca bibliotekę if{Y}.
- Serwer Grup – Komputer z działającą aplikacją, która zarządza grupami użytkowników i Zdarzeniami Grupowymi.
- Zdarzenie Grupowe – Zdarzenie związane z Grupą, wysyłane lub odbierane przez Aplikację z Serwera Grup.
- Grupa – Zbiór użytkowników identyfikowalny przez nazwę zdefiniowany na Serwerze Grup.
- Dziennik (ang. Log) – Moduł systemu odpowiedzialny za zapis zdarzeń.

3.2 Istniejące rozwiązania

3.2.1 On X[4]

Aplikacja firmy Microsoft umożliwiająca kontrolowanie telefonu z systemem Android przy użyciu kodu napisanego w JavaScript. Umożliwia wysyłanie Zasad (and. Rules) na telefon poprzez stronę internetową. Dostęp do funkcjonalności systemu Android jest zapewniony przez API w postaci Wyzwalaczy (ang. Triggers) i Akcji (ang. Actions). System ten posiada wiele wad:

- jest zamkniętym rozwiązaniem
- ma jeden centralny serwer
- login tylko przez konto Facebook
- brak Zdarzeń Grupowych

3.2.2 Tasker

Rozbudowana aplikacja służąca do automatyzacji zdarzeń. Umożliwia tworzenie zadań (tasks) bezpośrednio z poziomu telefonu. Są one definiowane poprzez wybieranie opcji z wielu list – bywa to czasochłonne przez konieczność szukania opcji. Zaletą tego rozwiązania jest dostępność dla użytkowników nie będących programistami oraz liczba dostępnych akcji. Posiada on także wady:

- jest zamkniętym rozwiązaniem
- aplikacja jest płatna – kosztuje 14,99 zł w Google Play. [10]
- brak możliwości definiowania zadań poprzez kod
- brak Zdarzeń Grupowych

Rozdział 4

Architektura platformy

System składa się z biblioteki if{y}, przykładowej aplikacji appIFY oraz aplikacji działających na serwerze - Serwera Grup oraz Targowiska. Biblioteka zawiera moduł zarządzania Receptami i zbiór Podfunkcjonalności. Aplikacja korzysta z biblioteki, umożliwia przeglądanie zasobów Targowiska i zarządzanie grupami. Targowisko umożliwia przechowywanie Recept i tworzenie ich z poziomu przeglądarki. Serwer grup zapewnia komunikację między Receptami uruchomionymi na różnych urządzeniach w ramach grupy oraz zarządzanie grupami.

4.1 Recepty

Miejscem, gdzie zdefiniowane jest właściwe działanie Aplikacji są Recepty – są w nich opisane wszystkie Zdarzenia, które mają nastąpić po spełnieniu pewnych warunków. Docelowo będą one tworzone przez użytkowników i pobierane z Targowiska, jednak istnieją także przykładowe Recepty wbudowane w Aplikację, mające na celu ułatwienie użytkownikom tworzenia nowych na ich podstawie oraz rozszerzenie początkowej funkcjonalności aplikacji. Na Receptę składają się:

- opis używanych podfunkjonalności
- opis wymaganych parametrów
- opis jej właściwego działania

Deklarowanie używanych Podfunkcjonalności ma dwa główne cele - po pierwsze, użytkownik widzi, czego używa Recepta, co nieco poprawia jego bezpieczeństwo przy używaniu Recept innych użytkowników, po drugie pozwala to inicjalizować nasłuchiwanie Zdarzeń systemowych tylko wtedy, gdy istnieje aktywna Recepta, która na nie reaguje - kod Recepty nie musi inicjalizować większości Podfunkcjonalności, wystarczy deklaracja ich używania. Wyjątkiem jest Podfunkcjonalność grup, gdzie komunikację należy zainicjalizować.

Parametry pozwalają użytkownikowi na dostosowanie Recepty do swoich wymagań, bez potrzeby pisania nowej. W naszych przykładowych Receptach były to np. numer telefonu do wysłania SMS lub jego tekst czy też zasięg znajdowania znajomych na podstawie GPS. Właściwa logika Recepty jest zawarta w funkcji reakcji na Zdarzenie. Jest to rozwiązanie podobne do wzorca obserwatora, Recepta staje się jednak obserwatorem

automatycznie na podstawie zadeklarowanych Podfunkcjonalności, co pozwala zmniejszyć ilość kodu w Receptach.

4.1.1 Cykl życia

W kontekście platformy if{y} zdefiniować można następujący cykl życia Recepty:

1. Pisanie kodu
2. Kompilacja i budowa
3. Dystrybucja
4. Pobranie na urządzenie mobilne
5. Uruchomienie
6. Działanie

4.1.2 Recepty Grupowe

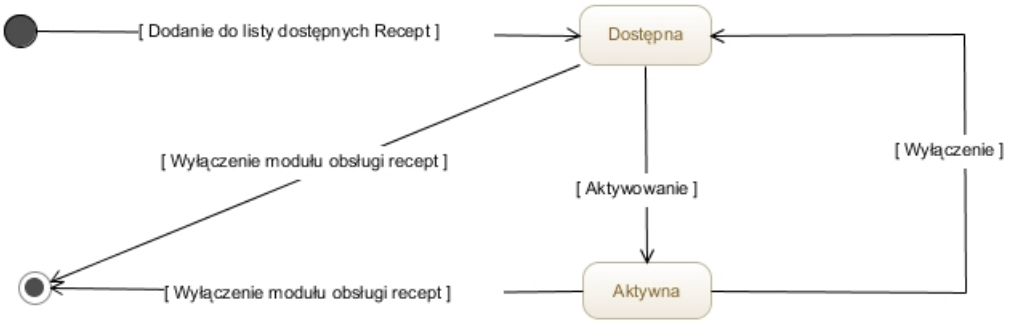
Główną ideą Recept grupowych jest działanie jednocześnie na wielu urządzeniach, tak, aby Zdarzenia na jednym urządzeniu mogły wpływać na drugie. Zatem komunikacja jest możliwa między instancjami tej samej Recepty na różnych urządzeniach. Jest ona możliwa na dwa sposoby:

- Wysyłanie Zdarzeń
  Polega na utworzeniu Zdarzenia, które jest obsługiwane na innym urządzeniu, które nasłuchuje go automatycznie.
- Wysyłanie danych
  Polega na wysyłaniu na Serwer danych, które potem mogą być odczytane na innym urządzeniu.

Nie da się natomiast napisać w Recepcie kodu, który byłby wywoływany na Serwerze. Kłóciłoby się to z prostotą mechanizmu Recepty - musiałaby się ona składać zarówno z kodu dla aplikacji, jak i dla Serwera, a programista musiałby zadbać o ich współpracę. Mogłoby to też stanowić zagrożenie dla stabilności Serwera w wypadku złośliwego kodu. Nie udało się też znaleźć scenariuszy użycia, w których taka funkcja byłaby konieczna. Dodatkowo przyjęta architektura pozwala na odseparowanie Serwera Grup od Targowiska.

4.2 Biblioteka

Biblioteka zawiera API dostępne z poziomu Recept, czyli między innymi Podfunkcjonalności, które agregują i upraszczają dostęp do metod z API systemu Android oraz umożliwiają reagowanie na Zdarzenia zachodzące na urządzeniu mobilnym. Istotnym elementem Biblioteki jest moduł zarządzający cyklem życia Recept i Podfunkcjonalności. W kontekście Biblioteki, cykl życia Recepty wygląda tak jak na rysunku 4.1.

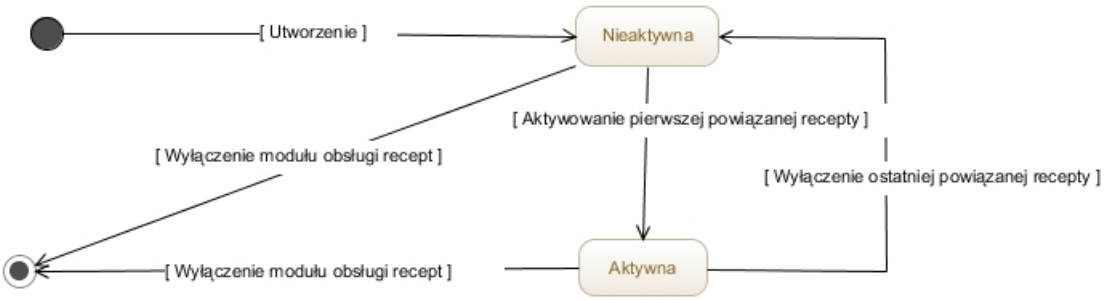


Rysunek 4.1: Cykl życia Recepty z punktu widzenia biblioteki

Wyróżniamy dwa stany w których znajdują się Recepty:

- Dostępna - udostępnia podstawowe informacje - nazwę, zbiór parametrów potrzebnych do uruchomienia i deklaracje Podfunkcjonalności, z których będzie korzystać.
- Aktywna - została uruchomiona na urządzeniu mobilnym, działa cały czas w tle nasłuchując na Zdarzenia, podczas jej uruchamiania zostały podane parametry i przydzielone Podfuncjonalności.

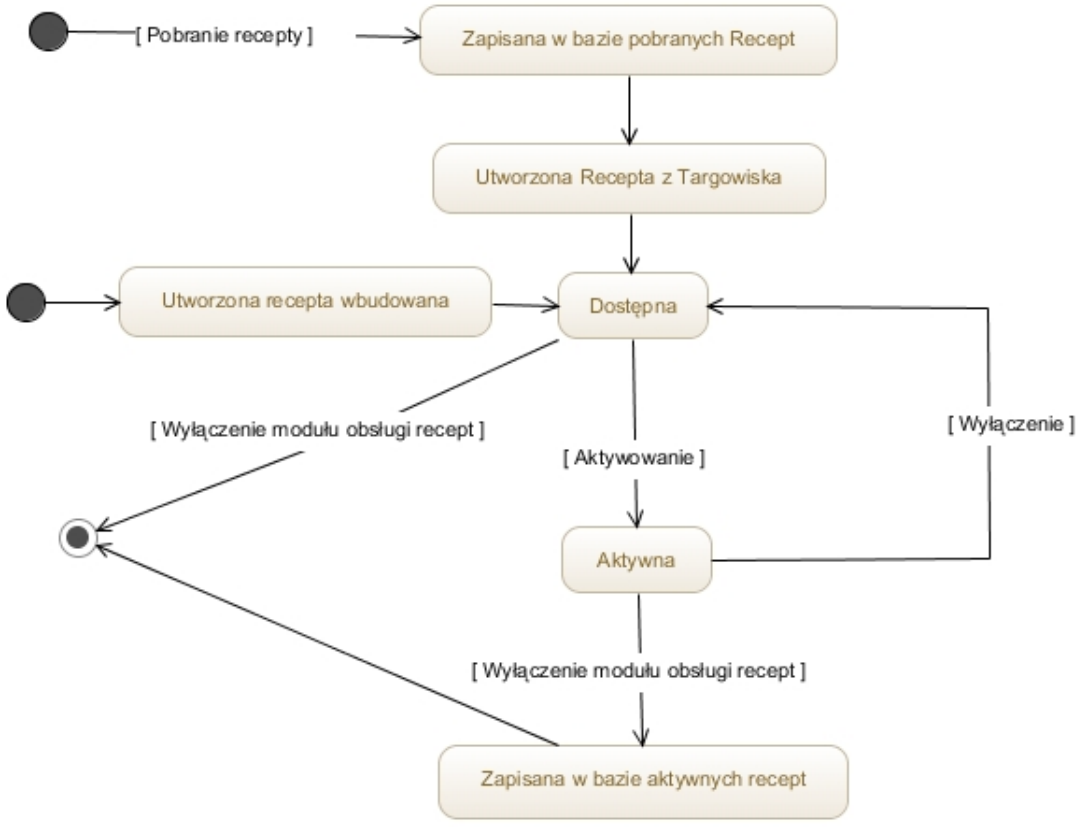
Warto zauważyć, że możliwe jest uruchomienie wielu instancji tej samej Recepty - mogą się różnić nadanymi parametrami. Cykl życia Podfunkcjonalności jest ściśle związany z Receptami - dana Podfunkcjonalność jest aktywna w systemie tylko jeżeli istnieją aktywne Recepty, które z niej korzystają. Moduł zarządzania Receptami aktywuje Podfunkcjonalność podczas uruchamiania pierwszej Recepty, która zadeklarowała jej użycie i wyłącza ją, kiedy nie pozostaną w systemie żadne aktywne Recepty, które mogłyby z niej korzystać. Podczas wyłączania zwalniane są wykorzystywane zasoby. Biblioteka zawiera również mechanizm komunikacji z Aplikacjami Klienckimi. Podczas projektowania biblioteki zapewniono możliwość rozszerzenia istniejącego zbioru Podfunkcjonalności. Moduł zarządzania Receptami składa się z listy aktywnych i dostępnych Recept oraz wykorzystywanych Podfunkcjonalności.



Rysunek 4.2: Cykl życia Podfunkcjonalności

4.3 Aplikacja appIFY

Aplikacja appIFY stanowi interfejs graficzny biblioteki na urządzeniu z systemem Android. Umożliwia dostęp do listy dostępnych i aktywnych Recept, ustalanie ich parametrów przy włączaniu i wyłączanie. Umożliwia korzystanie z zasobów Targowiska. Zapewnia możliwość zarządzania grupami użytkowników korzystając z API wystawionego przez Serwer Grup. W Aplikacji rozszerzono funkcje modułu obsługi Recept. Możliwe jest wykorzystywanie tych pobranych z Targowiska i zapisywanie stanu aktywnych Recept w bazie danych. Zapis stanu jest niezbędny, ze względu na mechanizm zarządzania zasobami w systemie Android – każdy proces może zostać zatrzymany w dowolnej chwili, dlatego potrzebny był sposób na odtworzenie stanu Aplikacji sprzed zamknięcia. Zmodyfikowany cykl życia Recepty przedstawia rysunek 4.3.



Rysunek 4.3: Cykl życia Recepty w appIFY

4.4 Targowisko

Podrozdział 2.2 definiuje zbiór wymagań pozafunkcjonalnych. Wymagania bezpośrednio dotyczące Recept takie jak PF12 czy PF13 są w praktyce niemożliwe do zrealizowania na

aplikacji mobilnej. W związku z powyższym postanowiono stworzyć aplikację pomocniczą realizującą w praktyce pierwsze trzy fazy cyklu życia Recepty w ramach określonych przez wymagania pozafunkcjonalne.

Pierwsze trzy fazy cyklu życia Recepty to kolejno:

1. Pisanie kodu
2. Kompilacja i budowa
3. Dystrybucja

Pierwsze dwa punkty powyższej listy można zrealizować poprzez dostarczenie zintegrowanego środowiska programistycznego. Trzeci, poprzez stworzenie aplikacji internetowej korzystającej z bazy danych. Powstało więc Targowisko – aplikacja internetowa przypominająca nieco Google Play, ale z dodatkowymi funkcjami. Pierwszą z nich jest zintegrowane środowisko programistyczne wbudowane bezpośrednio w Targowisko. Drugą, zastosowanie idei rozwidlania (ang. fork) Recept. Idea ta zakłada tworzenie kodu nie od podstaw, lecz przez klonowanie rozwiązań istniejących. Zostało to opisane szerzej w sekcji 4.4.2.

4.4.1 Zintegrowane środowisko programistyczne

Funkcjonalność zintegrowanego środowiska programistycznego jest realizowana poprzez osadzenie edytora kodu Java na stronie interentowej Targowiska. Strona posiada także mechanizmy umożliwiające kompilację i budowę kodu Recepty. Po pomyślnej kompilacji uzyskuje się archiwum Java (*jar*) gotowe do wgrania na Aplikację Kliencką if{y}.

Okazuje się, że istnieją darmowe edytory kodu nadające się do umieszczenia na stronie. Co więcej, wywoływanie komend systemu operacyjnego z poziomu aplikacji internetowej jest również w pełni możliwe. Wybrano więc napisany w JavaScript edytor Ace – ze względu na najlepszą dokumentację oraz język PHP który bardzo dobrze radzi sobie z wywoływaniem poleceń systemu operacyjnego.

4.4.2 Rozwidlanie

Pierwotne założenie dotyczące budowania nowych Recept zakładało istnienie generatora kodu. Miał on na celu umożliwienie wygenerowania części kodu w zależności od wybranych przez użytkownika Podfunkcjonalności z biblioteki if{y}. Generator taki był dyskusyjny z powodu pracochłonnej implementacji oraz specyfiki biblioteki. Generator powinno się implementować po ostatecznej implementacji biblioteki. Biblioteka jednak z założenia jest kodem, który dynamicznie rozwija się w czasie.

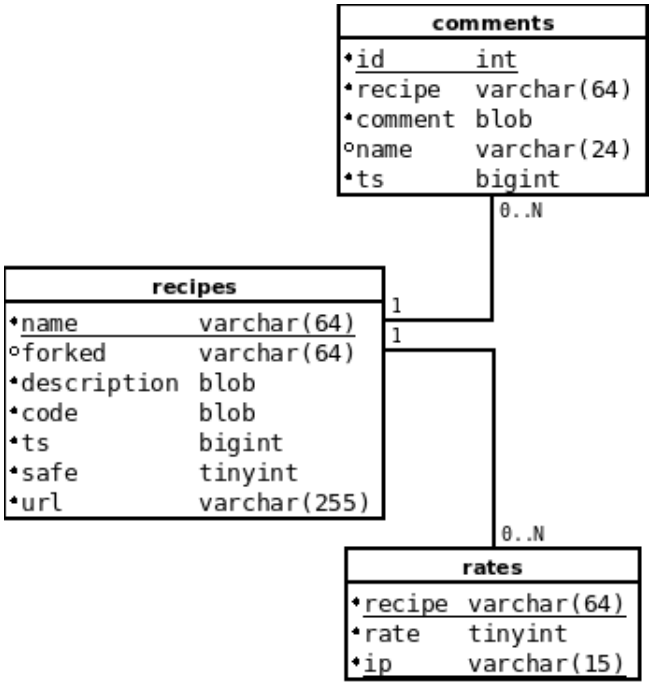
Ostatecznie, zamiast generatora, postanowiono wykorzystać ideę rozwidlania (ang. fork). Idea ta w praktyce oferuje prawie to samo co generator kodu. Istnieje jednak jedna spora różnica: rozwidlanie pielęgnuje się samo w sobie. Wystarczy stworzyć jeden działający kawałek kodu i opublikować go. Od tego momentu każdy może zacząć pisanie swojego kodu, przyjmując jako punkt wyjścia kod opublikowany wcześniej. Proces ten może się powtarzać rekursywnie w nieskończoność.

Stosowane podejście nie jest wolne od wad – kluczową może być fakt, iż działający kod niskiej jakości może być niepotrzebnie propagowany. Propagacja może też dotyczyć wysokiej jakości kodu, przez co problem propagacji można uznać za mało szkodliwy. Aby jednak

zachęcić programistów do pisania wartościowego kodu umożliwiono ocenianie Recept. Pomaga to przestrzegać użytkowników przed niedziałającymi i niebezpiecznymi Receptami.

4.4.3 Repozytorium Recept

Zarządzanie Receptami postanowiono zrealizować angażując strukturę katalogową Targowiska oraz bazę danych. Pliki ze zbudowanym kodem zgromadzono w katalogu *jar*. Wszelkie informacje pomocnicze umieszczono w bazie danych o schemacie widocznym na rysunku 4.4, gdzie:



Rysunek 4.4: Schemat bazy danych Targowiska

- recipes – jest to relacja, której każda krotka utożsamiana jest z pojedynczą Receptą. Każda Recepta posiada własną, unikalną nazwę, opcjonalnie nazwę Recepty, z której dana została wywiedziona, opis, kod źródłowy, znacznik czasu dodania, flagę informującą o niebezpiecznych konstrukcjach w kodzie oraz łącze do pliku *jar*.
- comments – jest to relacja której krotki reprezentują komentarze użytkowników na temat Recept. Każdy komentarz posiada unikalny identyfikator liczbowy, nazwę Recepty, której dotyczy, treść komentarza, opcjonalnie nazwę autora oraz znacznik czasu dodania.
- rates – jest to relacja, której krotki reprezentują oceny w całkowitej skali od 1 do 5, przyznawane Receptom. Na każdą ocenę składa się nazwa Recepty ocenianej, wartość całkowitoliczbowa oceny oraz adres IP oceniającego.

Stworzono także interfejs programowania aplikacji (API). Ma on na celu umożliwienie dostępu do Targowiska z poziomu Aplikacji Klienckiej if{y}. Interfejs ten pozwala na uzyskanie całej zawartości bazy danych w formacie JSON.

4.5 Serwer Grup

Głównym zadaniem Serwera Grup jest przekazywanie wiadomości umożliwiających komunikację miedzy klientami i wymianę danych. Informacje te powinny być jednocześnie rozsyłane w sposób prosty do odczytania przez każdą ze stron. Bezpośrednia komunikacja z użyciem połączenia internetowego między urządzeniami mobilnymi nie jest możliwa, dlatego niezbędnym było wprowadzenie urządzenia pośredniczącego w przesyłaniu danych. Rozwiązaniem tego problemu mogła być usługa Google Cloud Messaging(GCM), lecz nie spełniała jednego z założeń projektu o odseparowaniu aplikacji od sieci społecznościowych i innych popularnych serwisów. Innym z pomysłów było wykorzystanie protokołu MQTT (MQ Telemetry Transport), który jest jednak trudny w implementacji i prawdopodobnie zwiększyłby liczbę używanych bibliotek, nie przynosząc większych korzyści w porównaniu z odpytywaniem. Ostatecznie zaimplementowana koncepcja komunikacji opiera się o tak zwany mechanizm odpytywania (ang. polling), który jest prosty do zaimplementowania, a jednocześnie spełnia wszystkie wymagania stawiane w projekcie - odbieranie danych z Serwera wykonywane poprzez cykliczne zapytania eliminuje problem łączności między aplikacją a pośredniczącym Serwerem.

Istotnym aspektem w wymianie danych między użytkownikami są ograniczenia do komunikacji aby niemożliwe było otrzymanie wiadomości od niezidentyfikowanego użytkownika. Rozwiązaniem jest połączenie użytkowników w grupy, w obrębie których będą mogli rozsyłać wiadomości. W związku z tym Serwer musi także umożliwiać zarządzanie grupami.

Rozdział 5

Opis implementacji

5.1 Recepty

Recepty dziedziczą po klasie abstrakcyjnej YRecipe i implementują jej abstrakcyjne metody. Obrazuje to poniższy przykład Recepty, która odrzuca wszystkie nadchodzące połączenia i wysyła SMS o zdefiniowanej przez użytkownika treści do dzwoniącej osoby.

```
public class YSampleCallsSMS extends YRecipe {
   @Override
   public void requestParams(YParamList params) {
      //Message to send in SMS
      params.add("MSG",YParamType.String, "Sorry, I'm busy.");
   }

   @Override
   public long requestFeatures() {
      return Y.Calls | Y.SMS;
   }

   @Override
   public void handleEvent(YEvent event) {
      //event is incoming call
      if(event.getId() == Y.Calls){
         YCallsEvent e = (YCallsEvent) event;
         //extract phone number
         String phone = e.getIncomingNumber();
         //discard call
         mFeatures.getCalls().discardCurrentCall();
         //send sms
         mFeatures.getSMS().sendSMS(phone, mParams.getString("MSG"));
      }
   }
   @Override
   public String getName() {
```

```
      return "YSampleCallsSMS";
   }
   @Override
   public YRecipe newInstance() {
      return new YSampleCallsSMS();
   }
}
```

5.1.1 Parametry – requestParams

Metoda requestParams ma za zadanie poinformować, jakich parametrów Recepta wymaga do działania. Początkowo miała ona po prostu zwrócić listę i wyglądałaby tak:

```
public void requestParams() {
   YParamList params = new YParamList();
   params.add("MSG",YParamType.String, "Sorry, I'm busy.");
   return params;
}
```

jednak tworzenie listy i zwracanie jej to dwie linie, które byłyby identyczne w każdej Recepcie - ich wpisywanie może nieco irytować. Wobec tego obecnie metoda ta przyjmuje jako argument pustą listę parametrów, którą ma za zadanie wypełnić, zgodnie z założeniem maksymalnego uproszczenia kodu Recepty.

5.1.2 Używane Podfunkcjonalności – requestFeatures

Metoda requestFeatures ma za zadanie poinformować system, jakich Podfunkcjonalności używa Recepta. Początkowo była ona podobna do requestParams i wypełniała listę nowymi obiektami odpowiedniej klasy, co wyglądałoby tak:

```
public void requestFeatures(YFeatureList features) {
   features.add(new YCallsFeature());
   features.add(new YSMSFeature());
   return params;
}
```

Przy takim rozwiązaniu jednak tworzyło się wiele niepotrzebnych obiektów - poprawnie zainicjalizowane Podfunkcjonalności powinny być tworzone w systemie tylko raz. Wystarczyłaby zatem lista identyfikatorów, pozwalająca zainicjalizować odpowiednie Podfunkcjonalności. Identyfikatorów jest jednak na tyle mało, że tak naprawdę nie potrzeba prawdziwej listy, wystarczy maska bitowa. Ułatwia to przesyłanie takiej listy między modułami systemu, działającymi w różnych procesach - nie trzeba się martwić o implementację w liście interfejsu Parcelable, potrzebnego do przesyłania obiektów między procesami w Androidzie.

Ostatecznie zatem metoda ta zwraca liczbę typu long, będącą sumą bitów reprezentujących poszczególne Podfunkcjonalności. Mapowanie tych bitów jest zawarte w klasie Y.

```
[...]
public static final long Wifi = 0x0008;
public static final long GPS = 0x00010;
[...]
```

Dodatkowo warto zauważyć, że nazwy stałych w tej klasie odpowiadają nazwom Podfunkcjonalności oraz Zdarzeń - dla stałej **ABC** klasa z Podfunkcjonalnością nazywa się Y**ABC**Feature, a Zdarzenie - Y**ABC**Event. Powinno to ułatwić automatyczne generowanie kodu Recept.

5.1.3 Logika Recept – handleEvent

Metoda jest wywoływana, gdy w systemie nastąpi Zdarzenie związane z Podfunkcjonalnością używaną przez Receptę. W argumencie podawane jest Zdarzenie – obiekt typu YEvent. Aby poznać szczegóły Zdarzenia Recepta musi sprawdzić jego typ porównując wartość zwracaną przez getId() ze stałymi z klasy Y. Następnie można rzutować Zdarzenie na odpowiedni typ i poznać jego szczegóły.

Recepty mogą też zażądać od systemu pewnych danych, które są dostarczane asynchronicznie - na przykład przetłumaczenie danych z GPS na adres (Geocoder). Wyniki tego typu operacji również są przekazywane do Recepty jako typ YEvent.

Z poziomu obsługi Zdarzenia można także dostać się do listy Podfunkcjonalności oraz listy Parametrów poprzez metody getFeatures() i getParams(). Początkowo dostęp do Podfunkcjonalności odbywał się następująco:

```
YCallsFeature cf = (YCallsFeature) mFeatures.get(Y.Calls);
```

Jednak wymuszało to rzutowanie i niepotrzebnie wydłużało kod, zatem obecnie klasa YFeatureList zawiera metody pobierające konkretne podfuncjonalności.

```
public YCallsFeature getCalls() {
    return (YCallsFeature) get(Y.Calls);
}
```

Ich utrzymanie może być później nieco kłopotliwe - każde dodanie Podfunkcjonalności będzie wymagało dodania odpowiedniej metody, jednak uproszczenie kodu Recepty jest tego warte.

Warto również wspomnieć, że metoda handleEvent może rzucić dowolny wyjątek - Recepta zostanie wówczas wyłączona. Ułatwia to pisanie Recept zapewniając jednocześnie stabilność aplikacji.

5.1.4 Aktywacja

Fragmenty kodu przedstawione poniżej różnią się od oryginalnych – dla poprawy czytelności nie ma w nich tworzenia logów. Recepta jest aktywowana przez serwis, na podstawie nazwy i listy parametrów.

```
   public int enableRecipe(String name, YParamList params) {
      int id = ++mRecipeID;
```

```
    int timestamp = (int) (System.currentTimeMillis() / 1000);
    YRecipe recipe = mAvailableRecipesManager.getRecipe(name).newInstance();
    long feats = recipe.requestFeatures();
    YFeatureList features = new YFeatureList(feats);
    initFeatures(features);
    params.setFeatures(feats);
    if(!recipe.initialize(this, params, features, id, timestamp)){
       return 0;
    }
    for (Entry<Long, YFeature> entry : features) {
       entry.getValue().registerRecipe(recipe);
    }
    mActiveRecipesManager.put(id, recipe);
    return id;
 }
```

Generowany jest ID konkretnej instancji Recepty oraz zapisywany jest czas jej uruchomienia. Następnie tworzony jest nowy obiekt typu właściwego do konkretnej Recepty. W tym celu znajdujemy niezainicjalizowaną Receptę w bazie i posługujemy się metodą newInstance – w tym miejscu kodu nie jest znana nazwa klasy Recepty, aby móc wprost wywołać konstruktor. Innym możliwym rozwiązaniem byłby mechanizm refleksji, który jednak nie może być zoptymalizowany przez maszynę wirtualną [5], więc przyjęte rozwiązanie jest szybsze, Dalej na podstawie zwróconej przez Receptę maski bitowej tworzona jest lista Podfunkcjonalności wymaganych przez Receptę do działania. Następnie podfunkcjolności które już są aktywne są wpisywane do listy w miejsce niezainicjalizowanych, a pozostałe są aktywowane i dodawane do listy aktywnych.

```
 protected void initFeatures(YFeatureList features) {
    for (Entry<Long, YFeature> entry : features) {
       Long featId = entry.getKey();
       YFeature feat = mActiveFeatures.get(featId);
       if (feat != null) {
          entry.setValue(feat);
       } else {
          feat = entry.getValue();
          feat.initialize(this);
          mActiveFeatures.add(feat);
       }
    }
 }
```

Po zainicjalizowaniu Podfunkcjonalności Recepta jest w nich rejestrowana. Umożliwia to wywoływanie metody handleEvent w odpowiedzi na Zdarzenia systemowe. Warto zauważyć, że zarówno Recepty jak i Podfunkcjonalności są leniwie inicjalizowane, co pozwala tymczasowo używać niezainicjalizowanych obiektów, a potem zastępować je innymi bez wykonywania zbędnych operacji.

```
public final boolean initialize(IYRecipeHost host, YParamList params,
      YFeatureList features, int id, int timestamp) {
   mHost = host;
   mParams = params;
   mFeatures = features;
   mId = id;
   mTimestamp = timestamp;
   Log = new YLogger(createTag(mId, getName()), host);
   try {
      init();
   } catch (Exception e) {
      e.printStackTrace();
      return false;
   }
   return true;
}
```

Sama inicjalizacja Recepty to głównie wstrzyknięcie jej parametrów, Podfunkcjonalności, ID oraz czasu aktywacji. Oprócz tego jest tworzony Dziennik Recepty oraz jest wywoływana funkcja init() zawierająca kod inicjalizacyjny specyficzny dla danej Recepty (na przykład otwarcie kanału komunikacji z Serwerem Grup). Takie rozwiązanie w połączeniu w modyfikatorem final w metodzie, zapewnia jej wywołanie, a kod Recepty nie ma dostępu do danych, które nie są mu potrzebne. Dodatkowo funkcja init() może się nie powieść - wyjątki są wówczas łapane, metoda initialize() zwraca wartość false, a Recepta nie jest dodawana do listy aktywnych.

5.1.5 Dezaktywacja

Dezaktywacją Recepty również zajmuje się serwis. Polega ona na wyrejestrowaniu jej z Podfunkcjonalności, co powoduje, że nie dostanie ona powiadomienia o Zdarzeniach, a następnie usunięciu jej z listy dostępnych Recept. Dodatkowo są odinicjalizowane Podfunkcjonalności, z których nie korzysta żadna inna Recepta. Ich usuwanie z listy odbywa się w drugim przebiegu pętli, aby zabezpieczyć się przed wyjątkiem ConcurrentModificationException.

```
public void disableRecipe(Integer id) {
   YRecipe recipe = mActiveRecipesManager.get(id);

   List<Long> toDelete = new ArrayList<Long>();
   for (Entry<Long, YFeature> entry : recipe.getFeatures()) {
      YFeature feat = entry.getValue();
      YLog.d("SERVICE", "UnregisterRecipe: " + recipe.getName()
            + " from " + entry.getKey());
      feat.unregisterRecipe(recipe);
      if (!feat.isUsed()) {
         toDelete.add(entry.getKey());
```

```
            YLog.d("SERVICE", "UninitializeFeature: " + feat.getId());
            feat.uninitialize();
        }
    }
    mActiveFeatures.removeAll(toDelete);
    mActiveRecipesManager.remove(id);
}
```

5.1.6 Recepty Grupowe

Specyficzną grupą Recept są Recepty grupowe. Wyróżniają się one używaniem Podfunkcjonalności YGroupFeature, co pozwala im komunikować się z innymi urządzeniami. Do komunikacji służy klasa YComm, którą należy zainicjalizować w metodzie init():

```
private YComm comm;
@Override
public void init() {
    comm = getFeatures().getGroup()
         .createPoolingComm(this, getParams().getString("GROUP"), 5);
}
```

Obecnie jedynym sposobem na inicjalizację jest użycie metody createPoolingComm, podając jej jako argumenty Receptę, nazwę grupy, której dotyczy oraz okres czasu między kolejnymi odpytywaniami Serwera o nowe Zdarzenia. Nic jednak nie stoi na przeszkodzie aby, rozwijając bibliotekę, umożliwić tworzenie obiektów klasy YComm opartych na innych rozwiązaniach komunikacyjnych niż odpytywanie.

Gdy obiekt YComm zostanie utworzony, Recepta automatycznie odbiera Zdarzenia.

Aby wysłać Zdarzenie, należy użyć metod sendEvent lub broadcastEvent. Pozwalają one wyzwolić Zdarzenie odpowiednio na konkretnym urządzeniu (na podstawie nazwy użytkownika) lub na wszystkich należących aktualnie do grupy. Metody te mają kilka wersji różniących się parametrami. Dzięki temu można wysłać samo Zdarzenie bez danych, jedną zmienną dołączoną do Zdarzenia lub też cały zbiór. Parametrem, który zawsze występuje, jest tag - liczbowy identyfikator typu Zdarzenia, który można później sprawdzić w kodzie jego obsługi, co pozwala stworzyć kilka typów komunikatów bez potrzeby dodawania do nich danych. Jego wartość powinna być dodatnia - ujemne są zarezerwowane dla Zdarzeń systemowych.

W reakcji na odebranie Zdarzenia wywoływana jest metoda handleEvent z parametrem typu YGroupEvent, zawierającym obiekt typu YCommData, który umożliwia odczyt taga, danych oraz informacji o nadawcy.

Oprócz wysyłania Zdarzeń można też po prostu wysyłać dane na Serwer. Służą do tego metody sendVariable oraz sendVariables, wysyłające odpowiednią jedną zmienną lub ich zbiór. Aby je odczytać, należy użyć metody getVariables, pobierającej dane konkretnego użytkownika lub getAllVariables, pobierającej dane całej grupy. Odpowiedź przychodzi do Recepty w formie Zdarzenia.

Dane przysyłane między Receptami są typu YParam - ta sama klasa jest używana do parametrów, jednak potrzeby są tutaj takie same - jest to klasa opakowująca obiekty

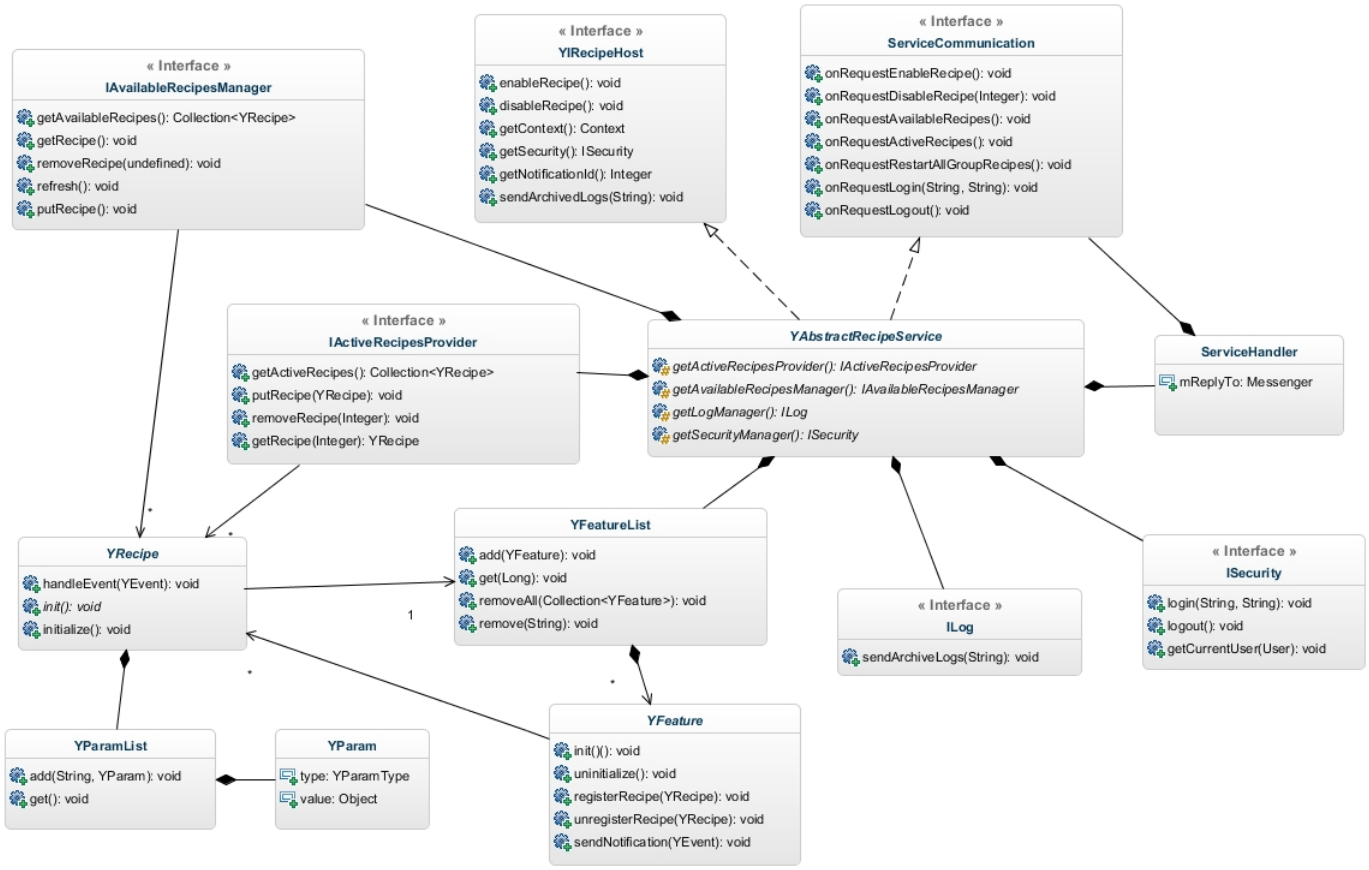
różnych typów, łatwo konwertowalna do ciągu znaków i zawierająca informacje o przechowywanym typie.

Zbiory danych zapisywane są w formie słownika, gdzie kluczem jest nazwa zmiennej (String), a wartością zmienna (YParam).

Odczyt danych odbywa się poprzez metodę getData na obiekcie typu YComm, przyjmującym jako argument nazwę zmiennej. W przypadku Zdarzeń będących odpowiedzią na metody getVariables i getAllVariables, należy użyć wersji dwuargumentowej, z dodatkowym argumentem określającym właściciela zmiennej. Innym sposobem dostępu jest pobranie całego słownika z danymi metodą getValues.

5.2 Biblioteka

5.2.1 Zarządzanie Receptami



Rysunek 5.1: Diagram klas modułu zarządzania Receptami

Moduł zarządzania Receptami został zaimplementowany w klasie abstrakcyjnej YAbstractRecipeService, dziedziczącej po Serwisie. Serwis w Androidzie to komponent aplikacji przeznaczony do długotrwałego wykonywania operacji w tle, nieposiadający interfejsu użytkownika. [6] Zaleca się uruchamianie Serwisu Biblioteki w osobnym procesie, dzięki czemu uruchomione Recepty mogą działać niezależnie od Aplikacji. Warto w tym miej-

scu nawiązać do zarządzania pamięcią w systemie Android - każda aplikacja uruchomiona przez użytkownika może zostać w dowolnym momencie zamknięta ze względu na konieczność zwolnienia zasobów. Serwis Biblioteki został zaimplementowany w taki sposób, aby być z powrotem uruchamiany po zamknięciu. Możliwe jest także uruchamianie go po starcie systemu. Podczas projektowania tego modułu kluczowe było zapewnienie maksymalnego uproszczenia mechanizmów zarządzania Receptami z punktu widzenia programisty Aplikacji, przy jednoczesnej dowolności implementacji list dostępnych i aktywnych Recept. Pozwoliło to na stworzenie appIFY, w której użytkownik ma pełną kontrolę nad modułem zarządzania, jak i wykorzystanie Biblioteki do napisania aplikacji wykorzystującej jedną wbudowaną Receptę, niewidoczną dla użytkownika końcowego.

5.2.2 Podfunkcjonalności

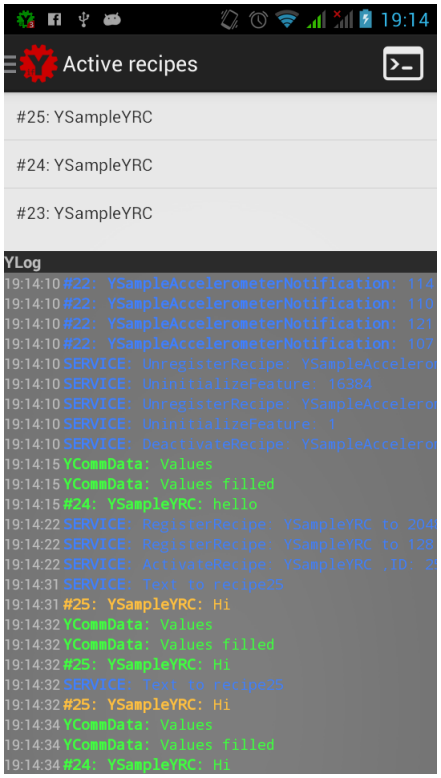
Podfunkcjonalności to klasy agregujące pewne funkcje związane z systemem. Muszą być inicjalizowane, gdy zajdzie taka potrzeba i przechwytywać Zdarzenia systemowe, przekazując je odpowiednim Receptom. Klasą bazową jest dla nich YFeature. Są w niej zaimplementowane metody związane z czasem życia Podfunkcjonalności i Recepty, odpowiedzialne za rejestrowanie i wyrejestrowywanie Recept, inicjalizację i deinicjalizację Podfunkcjonalności oraz sprawdzanie, czy Podfunkcjonalność jest używana przez Recepty. Poza tym znajduje się w niej metoda odpowiedzialna za wysyłanie Zdarzenia do Recept - sendNotification, wykorzystywana w poszczególnych Podfunkcjonalnościach. Zaimplementowano następujące Podfunkcjonalności:

- Akcelerometr (YAccelerometerFeature.java)
  Umożliwia reagowanie na odczyty akcelerometru wbudowanego w urządzenie.
- AudioManager (YAudioManager.java)
  Umożliwia zarządzanie poziomem głośności dzwonka.
- Battery (YBatteryFeature.java)
  Umożliwia reagowanie na zmiany poziomu naładowania baterii urządzenia.
- Calls (YCallsFeature.java)
  Umożliwia reagowanie na połączenia przychodzące i inicjowanie połączeń wychodzących.
- Files (YFilesFeature.java)
  Umożliwia tworzenie i odczytywanie plików z pamięci urządzenia lub nośnika zewnętrznego.
- Geocoder (YGeocoderFeature.java)
  Umożliwia pobranie adresu związanego z podaną długością i szerokością geograficzną.
- GPS (YGPSFeature.java)
  Umożliwia śledzenie pozycji urządzenia za pomocą modułu GPS.
- Group (YGroupFeature.java)
  Niezbędny do obsługi Zdarzeń grupowych.

- Intent (YIntentFeature.java)
  Pozwala wysyłać intencje[8] umożliwiające m. in. uruchamianie innych aplikacji.
- Internet (YInternetFeature.java)
  Umożliwia wysyłanie i pobieranie danych z podanego adresu.
- Notification (YNotificationFeature.java)
  Umożliwia wyświetlanie powiadomień w interfejsie graficznym urządzenia.
- RawPlayer (YRawPlayerFeature.java)
  Umożliwia odtwarzanie dźwięków na podstawie tablicy częstotliwości.
- Shortcut (YShortcutFeature.java.java)
  Pozwala na tworzenie skrótów do Recepty na głównym ekranie.
- SMS (YSMSFeature.java)
  Umożliwia wysyłanie wiadomości SMS oraz reagowanie na wiadomości przychodzące.
- Sound (YSoundFeature.java)
  Pozwala odtwarzać pliki dźwiękowe.
- Text (YTextFeature.java)
  Umożliwia wprowadzanie tekstu do Recepty z poziomu aplikacji.
- Time (YTimeFeature.java)
  Tworzy Zdarzenia w równych odstępach czasu.
- Wifi (YWifiFeature.java)
  Umożliwia włączanie i wyłączanie modułu WiFi urządzenia.

5.2.3 Dziennik

Biblioteka posiada moduł Dziennika, pozwalającego na zapisywanie i wyświetlanie informacji. Dostęp do niego miał przypominać jak najbardziej klasę Log [9] w Androidzie. Powstała zatem klasa YLog, zawierająca między innymi statyczne metody v,d,i,w,e, identyczne jak w Androidzie. Wywołują one swoje odpowiedniki, aby umożliwić debugowanie poprzez narzędzie logcat. Oprócz tego wpisy są przechowywane wewnętrznie. Do ich wyświetlania służy specjalny widok, wyświetlany jako nakładka. Nie reaguje on na dotyk, przez co umożliwia użytkowanie telefonu, gdy jest widoczny. Jest to jedyny widok obsługiwany przez Bibliotekę, a nie Aplikacje Klienckie. Dodatkowo istnieje klasa YLogger, której instancje są przypisane do Recepty. Wpisy przez nią utworzone są przypisane do konkretnej Recepty. Dzięki temu można je łatwo odfiltrować i wyświetlać na ekranie Recepty w Aplikacji Klienckiej. Recepty mają do niej dostęp poprzez pole Log – jego nazwa wydaje się przeczyć konwencji, jednak przypomina dzięki temu oryginalną klasę Log ze statycznymi metodami. Jedyną różnicą są argumenty metod – zamiast taga i wiadomości przyjmowana jest jedynie wiadomość, tag jest tworzony automatycznie na podstawie Recepty.



Rysunek 5.2: Widok Dziennika nad ekranem Aplikacji Klienckiej

5.3 Aplikacja kliencka

5.3.1 Obsługa Targowiska

Moduł obsługi Targowiska jest odpowiedzialny za wyświetlanie danych, dotyczących Recept dodanych w aplikacji internetowej oraz pobieranie plików .jar ze skompilowanymi Receptami, które następnie są zapisywane w pamięci wewnętrznej urządzenia mobilnego (w celu zachowania tej samej bazy Recept w przypadku, w którym użytkownik usunie zewnętrzny nośnik pamięci z urządzenia). Informacje o plikach z Receptami (ich nazwy oraz ścieżki) przechowywanymi na telefonie zapisywane są po pomyślnym pobraniu w bazie danych.

5.3.2 Obsługa pobranych Recept

W Aplikacji Klienckiej, zrealizowanej w ramach pracy inżynierskiej, rozróżniamy dwa typy Recept - wbudowane i pobrane z Targowiska. Kod źródłowy Recept pierwszego typu jest zawarty w kodzie źródłowym Aplikacji. W przypadku Recept pobranych z Targowiska, w celu umożliwienia Aplikacji korzystania z takiej Recepty, wykorzystywane jest archiwum .jar, zawierające plik .dex (Dalvik Executable) z kodem wykonywalnym zrozumiałym dla maszyny wirtualnej Dalvik. Informacje potrzebne do załadowania kodu Recepty (nazwa klasy oraz ścieżka dostępu do pliku .jar) przechowywane są w bazie danych Recept pobranych na urządzenie.

5.3.3 Komunikacja serwisu z Aplikacją Kliencką

W komunikacji między Aplikacją Kliencką a serwisem Recept wykorzystane będą klasy z Android SDK - Messenger, Bundle i interfejs Parcelable. Klasa Messsenger umożliwia przesyłanie danych między procesami. [7] Do opakowania danych wykorzystywana jest klasa Bundle, która przechowuje obiekty i typy prymitywne w postaci mapy. Warto wspomnieć, że aby uzyskać możliwość przechowania obiektu w tej klasie, musi on implementować interfejs Serializable lub Parcelable. Pierwszy z nich umożliwia serializacje obiektów znaną z Javy, natomiast drugi został zaimplementowany w Android SDK w celu zwiększenia wydajności serializacji. W pracy inżynierskiej wykorzystujemy drugi z mechanizmów. Po uruchomieniu serwis Recept wystawia obiekt implementujący interfejs IBinder służący do wiązania obiektów klasy Activity z obiektami klasy Service, z którym z kolei jest związany obiekt klasy Messenger zaimplementowany w serwisie Recept. Aby ustanowić połączenie, Aktywność musi stworzyć obiekt klasy Intent, sparametryzować go klasą Service, z którą nawiązywane jest połączenie i zapewnić obiekt implementujący interfejs ServiceConnection, który reaguje na uzyskanie i zerwanie połączenia, a następnie wywołać metodę bindService jako parametr podając wspomniany wyżej obiekt klasy Intent. Po nawiązaniu połączenia następuje wymiana obiektów klasy Messenger, dzięki czemu możliwa jest komunikacja w obie strony. Warto dodać, że ten mechanizm komunikacji jest asynchroniczny. Wiadomości wysyłane przez klasę Messenger odbierane są przez klasę Handler, ich zawartość jest interpretowana dzięki wysyłanemu kluczowi a następnie dane są przekazywane serwisowi Recept lub aktywności Aplikacji Klienckiej w celu dalszego przetwarzania. W pracy inżynierskiej wykorzystano dwie klasy dziedziczące po klasie Handler - ServiceHandler dla obsługi wiadomości przychodzących do serwisu Recept i ActivityHandler dla obsługi wiadomości przychodzących do Aplikacji Klienckiej. W celu rozszerzenia komunikacji o wiadomości których obecna implementacja nie przewiduje, należy stworzyć własną klasę dziedziczącą po klasie ServiceHandler i we własnej implementacji klasy YAbstractService nadpisać metodę getServiceHandler. Podobnie, aby rozszerzyć komunikację w drugą stronę należy stworzyć własną klasę dziedziczącą po ActivityHandler i użyć go do odbierania wiadomości od serwisu Recept.

5.4 Targowisko

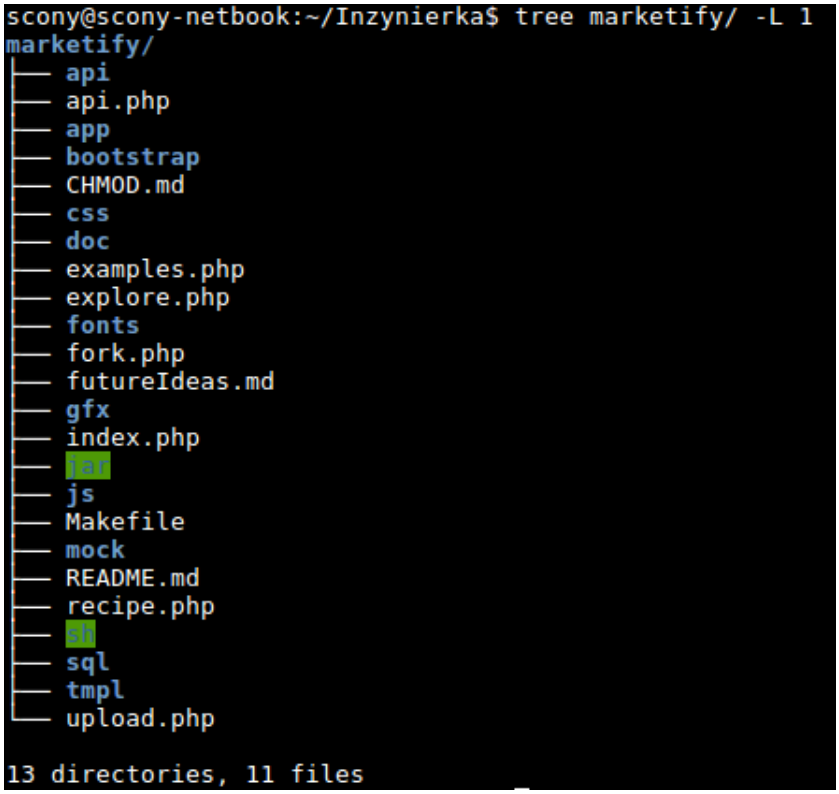
Targowisko zaimplementowano w formie aplikacji internetowej. Łączy ono w sobie technologie takie jak PHP, MySQL, HTML, CSS oraz JavaScript. Jedyną różnicą w stosunku do przeciętnych stron internetowych jest integracja z systemem operacyjnym poprzez skrypty powłoki BASH.

Targowisko ma trzy cele:

1. Zapewnić prosty w użyciu interfejs użytkownika.
2. Udostępnić zunifikowany interfejs programowania aplikacji (API) który pozwoli uzyskać dostęp do wszystkich informacji z bazy danych.
3. Zapewnić dostęp do skryptów powłoki BASH oraz zagwarantować im bezpieczne wykonanie.

W kontekście celów oraz technologii, jako wzór postępowania przyjęto wzorzec projektowy Model-Widok-Kontroler (MVC). Z racji swoich organizacyjnych właściwości, jest on najlepszym wyborem pozwalającym oddzielić logikę od interfejsu.

5.4.1 Wzorzec MVC



Rysunek 5.3: Struktura katalogowa Targowiska

Poprzez użycie Wzorca MVC w Targowisku uniemożliwiono technologiom mieszanie się ze sobą. Zjawisko takie jest częstym błędem, popełnianym przez twórców aplikacji internetowych. Utrudnia ono w znacznym stopniu rozwój oraz pielęgnację kodu. Konsekwencje użycia wzorca MVC widać najwyraźniej w strukturze katalogowej. Prezentuje ją rysunek 5.3. W głównym katalogu Targowiska znajdują się pliki o rozszerzeniu *php*. Są to kontrolery przetwarzające dane pochodzące od użytkownika. Pozostałe kontrolery znajdują się w podkatalogu *api*. Mają one na celu przetwarzanie żądań kierowanych do API.

Widoki znajdują się w katalogu *tmpl*. Ich poprawne działanie jest jednak gwarantowane poprzez style oraz skrypty zgromadzone kolejno w podkatalogach *css* oraz *js*.

Modele znaleźć można w podkatalogach *app* oraz *sql*.

5.4.2 Interfejs graficzny

Do realizacji graficznego interfejsu użytkownika użyto darmowy framework; Twitter Bootstrap. Jest to obszerny zbiór stylów CSS wraz z rzetelną dokumentacją. Pozwala on osobie, nie będącej uzdolnioną artystycznie, tworzyć dobrze wyglądające widoki aplikacji.

Framework jest bardzo prosty w użyciu. Po pierwsze, należy dołączyć jego pliki do struktury katalogowej swojej aplikacji. Po drugie, do kodu HTML należy dodać następującą linię:

```
<link href="./css/bootstrap.css" rel="stylesheet">
```

Aby przykładowo utworzyć efektywne menu, do znacznika UL będącego kontenerem dla hiperłączy wystarczy dodać odpowiednie klasy tak jak w przykładzie poniżej:

```
<ul class="nav nav-pills pull-right">
<li><a href="explore.php">Recipes</a></li>
<li><a href="examples.php">Examples</a></li>
<li><a href="upload.php">Upload</a></li>
<li><a href="doc" target="_blank">Doclava</a></li>
<li><a href="api.php">API</a></li>
</ul>
```

Dla lepszego wyobrażenia możliwości oferowanych przez framework, wystarczy spojrzeć na rysunek 5.4.

5.4.3 Edytor Ace

W celu umożliwienia pisania kodu Recept z poziomu Targowiska, skorzystano z darmowego edytora Ace. Daje on możliwości znane z popularnych narzędzi takich jak Vim czy Eclipse (rysunek 5.5). Napisano je w JavaScript, przez co osadzenie go we własnej aplikacji internetowej ogranicza się wyłącznie do modyfikacji widoków.

Kod edytora należy najpierw dołączyć do kodu widoku. Można to zrobić dwojako; poprzez dołączenie plików edytora do struktury katalogowej lub poprzez dołączenie kodu edytora ze zdalnej lokalizacji. W Targowisku skorzystano z drugiej możliwości:

```
<script src="http://ace.c9.io/build/src-min/ace.js" type="text/javascript"">
</script>
<script src="http://ace.c9.io/build/src-min/ext-language_tools.js" type="text/javascript">
</script>
```

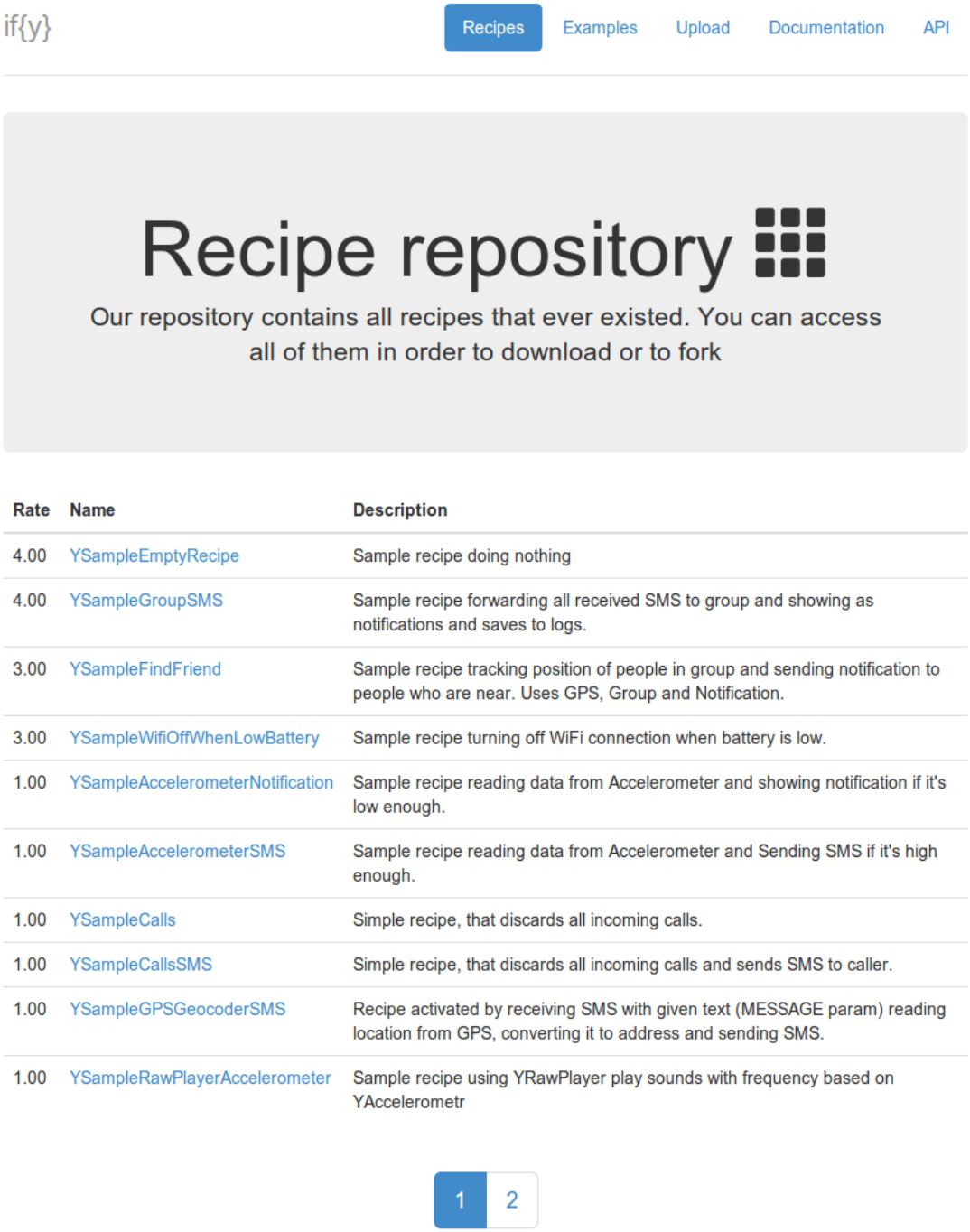
Pierwsze dwie linie dotyczą samego edytora. Pozostałe dwie dołączają rozszerzenie używane w Targowisku celem uzyskania autouzupełniania widocznego na rysunku 5.5.

Samo dołączenie kodu nie wystarcza do działania edytora. W treści strony trzeba dodać znacznik *pre* o odpowiednim parametrze id:

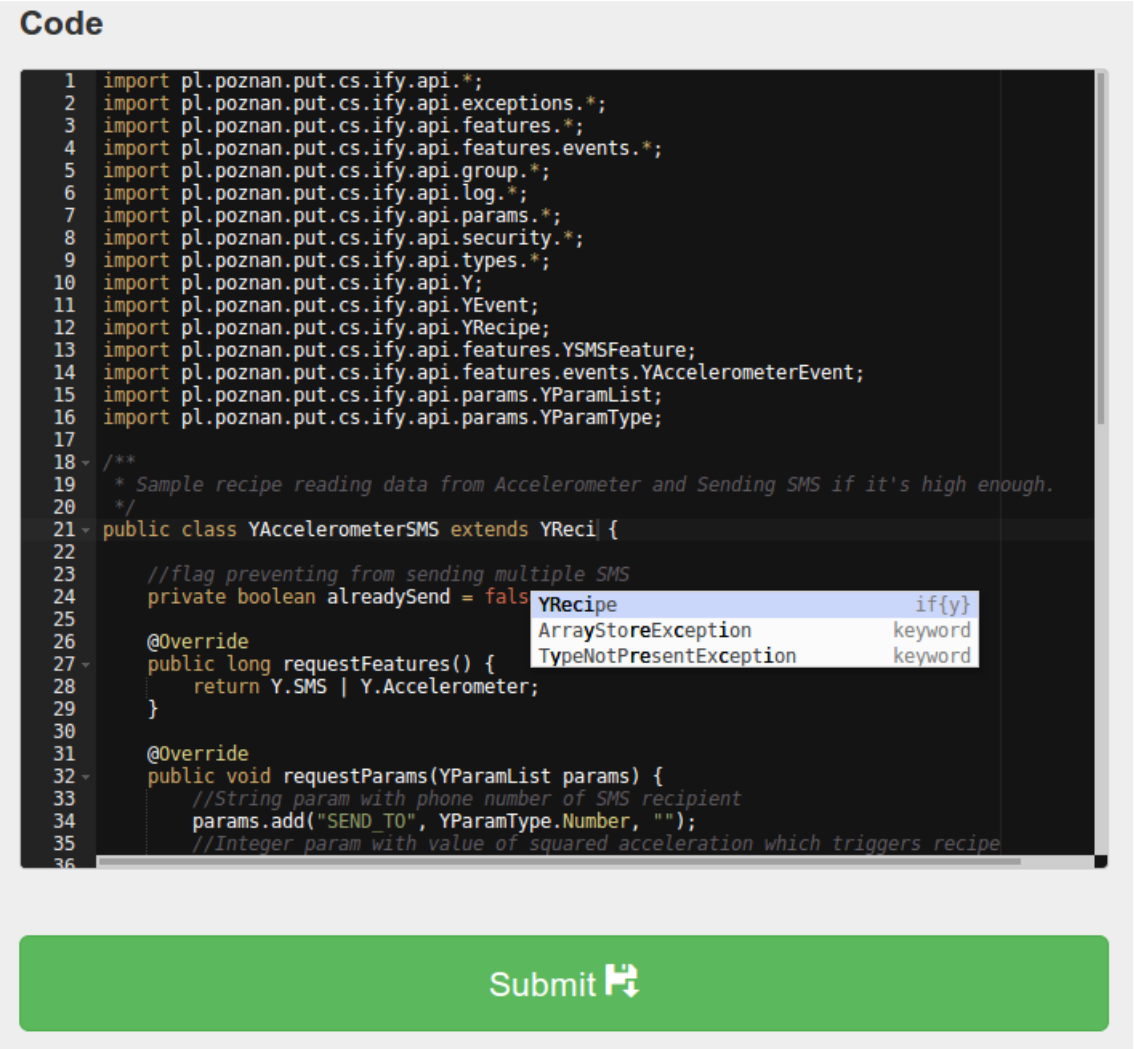
```
<pre id="editor"></pre>
```

Na koniec wykonać należy prosty skrypt JavaScript który uruchomi edytor wraz z rozszerzeniem:

```
var langTools = ace.require("ace/ext/language_tools");
var editor = ace.edit("editor");
editor.setOptions({enableBasicAutocompletion: true});
editor.setTheme("ace/theme/twilight");
```



Rysunek 5.4: Framework Twitter Bootstrap zastosowany do widoku listy Recept Targowiska



Rysunek 5.5: Edytor Ace oferuje między innymi kolorowanie składni oraz autouzupełnianie

```
editor.session.setMode("ace/mode/java");
var ifyCompleter = {
    getCompletions: function(editor, session, pos, prefix, callback) {
        $.getJSON(
          "./api/completer.php" + (prefix.length === 0 ? "" : "?prefix=" + prefix),
          function(wordList) {
      callback(null, wordList.map(function(ea) {
          return {name: ea.word, value: ea.word, score: ea.score, meta: "if{y}"}
                ;}));
          });
    }
}
langTools.addCompleter(ifyCompleter);
```

5.4.4 API

Jako alternatywną dla interfejsu użytkownika metodę dostępu do danych, w Targowisku zaimplementowano interfejs API. Jest on dedykowany aplikacjom na urządzenia mobilne. Pracuje w trybie tylko do odczytu, dzięki czemu nie występuje ryzyko ingerencji w dane.

Działanie API opiera się o wywołania GET protokołu HTTP. Dane, będące wynikiem wywołania, zwracane są w formie wielowymiarowej tablicy zapisanej w formacie JSON. API Targowiska wyróżnia pięć możliwych żądań:

- O najnowsze Recepty
  - http://ADRES_TARGOWISKA/api/new.php?limit={limit}
  - {limit}, liczba naturalna – ilość Recept do pobrania
- O wszystkie Recepty
  - http://ADRES_TARGOWISKA/api/recipes.php?page={page}&limit={limit}
  - {page}, liczba naturalna – numer strony z której zostaną pobrane Recepty
  - {limit}, liczba naturalna – ilość Recept przypadających na każdą stronę
- O Recepty których nazwy pasują do wzorca
  - http://ADRES_TARGOWISKA/api/search.php?phrase={phrase}
  - {phrase}, łańcuch znaków – wzorzec wyszukiwania
- O konkretną Receptę
  - http://ADRES_TARGOWISKA/api/recipe.php?name={name}
  - {name}, łańcuch znaków – pełna nazwa Recepty
- O losową Receptę
  - http://ADRES_TARGOWISKA/api/random.php

Przykładowa odpowiedź Targowiska na żądanie o losową Receptę może wyglądać następująco:

```
[
```

```
   {
      "name":"YSampleYRC",
      "forked":null,
      "description":"Simple IRC - like chat, using YTextEvent, YGroupFeature and YLog.",
      "code":"(...)",
      "ts":1391197304,
      "safe":1,
      "url":"http:\/\/ADRES_TARGOWISKA\/jar\/YSampleYRC.jar",
      "rate":null,
      "comments":[

      ]
   }
]
```

5.4.5 Kompilacja Recept

Po dodaniu Recepty do bazy danych Targowiska jest ona kompilowana oraz przekształcana do wynikowego pliku *jar* poprzez następujący skrypt powłoki BASH:

```
#!/bin/bash

if [ $# -eq 1 ];
then
    # clean
    rm -f recipe.jar
    # build
    javac -source 1.6 -target 1.6 -bootclasspath rt.jar
    -cp ify.jar:android.jar:rt.jar $1.java\
    || exit 1 # compability with jdk 1.6
    # javac -cp ify.jar:android.jar $1.java || exit 1
    jar cf tmp.jar $1.class
    ./dx --dex --output=classes.dex tmp.jar
    jar cf recipe.jar $1.class classes.dex
    # post-clean
    rm -f *.class
    rm -f classes.dex
    rm -f tmp.jar
    rm -f *.java
    exit 0
else
    echo 'Usage: ./build.sh [recipe class name without .java]'
    exit 2
fi
```

5.5 Serwer Recept grupowych

5.5.1 Komunikacja z Receptami

Komunikacja Recept z Serwerem grup odbywa się za pomocą odpowiedniego zapytania do Serwera:

- Adres docelowy: ADRES_SERWERA/rest/recipe
- Typ zapytania: POST
- Content-Type: application/json

W ciele zapytania znajduje się JSON o następującej strukturze:

```
{
    "user": {
        "username": STRING,
        "recipe": STRING,
        "group": STRING,
        "password": STRING
    },
    "event": {
        "target": STRING,
        "tag": INT
    },
    "values": {
        STRING: {
            "value": OBJECT,
            "type": STRING
        },
        STRING: {
            "value": OBJECT,
            "type": STRING
        },
       [...]
    }
}
```

Wyrazy pisane wersalikami to typy danych, których wartości są zależne od konkretnego zapytania.

Pole user ma na celu identyfikację nadawcy wiadomości - zawiera on nazwę użytkownika, Recepty i grupy. Znajduje się tutaj także hasło, a ściślej - skrót SHA-1 hasła, używający loginu jako soli.

Pole event to login użytkownika, do którego skierowany jest komunikat lub wartość specjalna BROADCAST, oznaczająca skierowanie do wszystkich oraz tag określający typ Zdarzenia.

Pole values to wartości przesłane wraz ze Zdarzeniem. Mają formę słownika indeksowanego ciągami znaków. Każdy obiekt ma typ i wartość. Typy są tożsame z typami

zdefiniowanymi w klasie YParamType, przy czym przy przesyłaniu JSON-en wartości typów innych niż Integer i Boolean są zamieniane na String.

Taki schemat danych odpowiada podstawowej operacji - sendEvent. Tag jest wówczas dodatni. Pozostałe operacje jednak posługują się tym samych schematem z drobnymi różnicami:

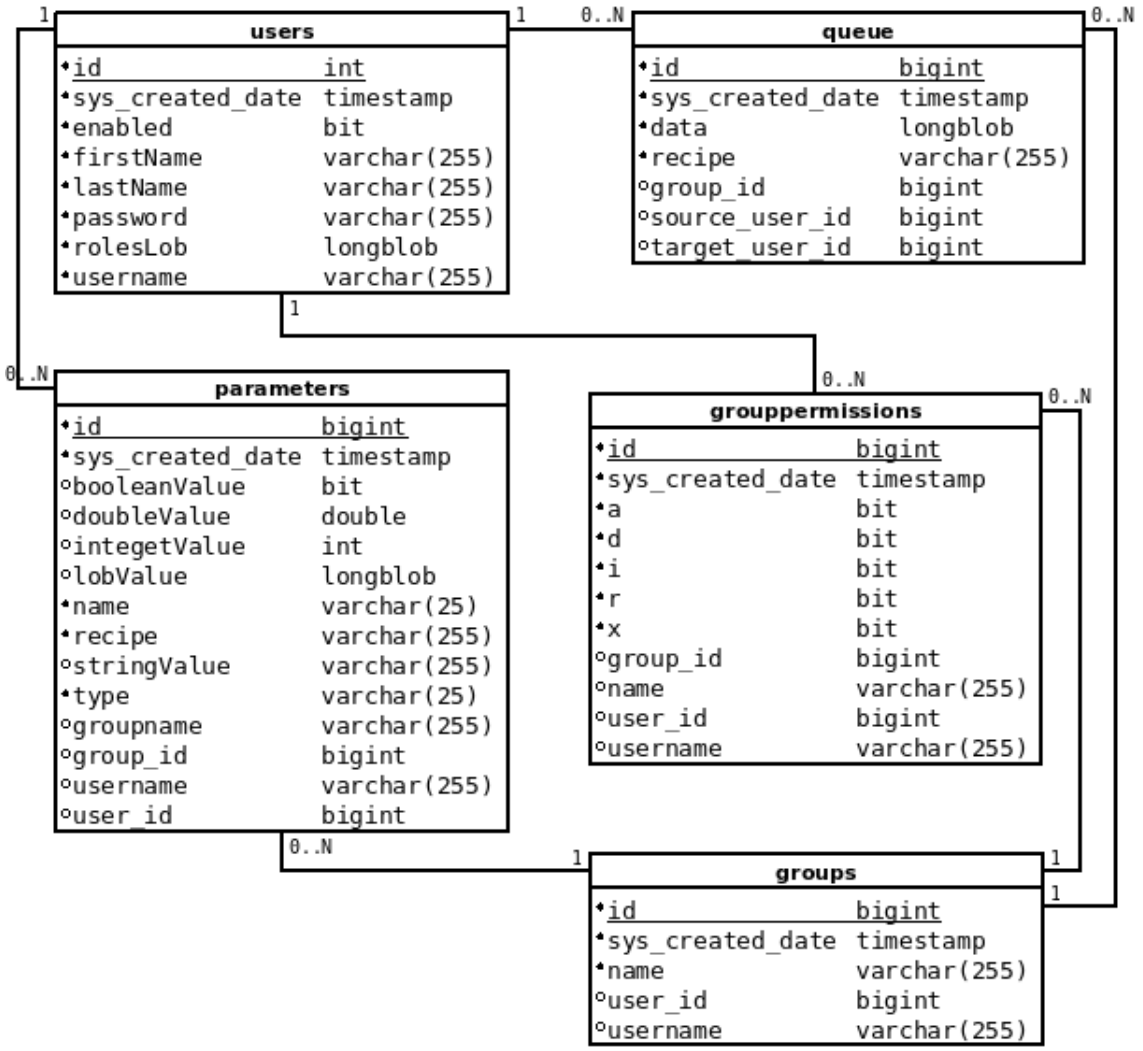
- sendValues
  Tag jest równy stałej YCommand.SEND_DATA = 0. Pole event.target jest ignorowane. Nazwy tych danych różnią się od tych określonych w Recepcie - mają formę nazwa@użytkownik, co zapewnia ich unikalność kluczy przy pobieraniu przez getAllValues. Oczywiście znak @ nie może być częścią loginu użytkownika, co jest weryfikowane przy rejestracji.
- getValues
  Tag jest równy stałej YCommand.GET_DATA = -1. Pole event.target to użytkownik, którego dane nas interesują. W przypadku pustej wartości Serwer zwraca dane całej grupy (operacja getAllValues). Pole values jest ignorowane.
- poll
  Tag jest równy stałej YCommand.POOLING = -4. Jest to techniczna operacja odpytywania Serwera o Zdarzenia kierowane do Recepty. Pola event.target oraz values są ignorowane.

Odpowiedzi na żądania, jeśli wymagają przesłania danych, są przekazywane w podobnej formie. W wypadku Zdarzeń pole user identyfikuje nadawcę. Oczywiście jest z niego usuwane pole password. Pola event i values są takie, jak w wiadomości tworzącej Zdarzenie. Przy odpowiedziach na GET_DATA pole values jest wypełnione wartościami danych zapisanymi na Serwerze. Pola user i event są kopią tych wysłanych, jednak ich wartość jest ignorowana.

5.5.2 Baza danych

Baza Serwera Recept grupowych składa się z 5 tabel jak pokazano na rysunku 5.6. Tabela `users` przechowuje dane o użytkownikach systemu takie jak login, hasło w formie skrótu, imię, nazwisko. Tabela `groups` reprezentuje grupy użytkowników, przechowuje nazwy oraz informacje o twórcy grupy. Tabela `groupspermissions` jest tabelą łączącą użytkowników z grupami oraz przechowującą informacje o uprawnieniach, jakie posiada użytkownik w danej grupie. Wpis w tej tabeli potwierdza przynależność użytkownika do grupy. Tabela `parameters` przechowuje dane aplikacyjne utworzone przez Recepty. Dane te mogą zostać zapisane w jednej z 4 kolumn w zależności od typu, jaki reprezentuje. Możliwymi typami danych, które obsługuje baza, są Double (liczby zmiennoprzecinkowe), Integer (liczby całkowite), Boolean (zmienne typu logicznego) oraz String (zmienne znakowe). Jeżeli typ podany w kolumnie `type` nie odpowiada żadnej z tych wartości, za domyślny typ przyjmowany jest String i w takiej formie jest zapisywany parametr. Każdy z parametrów zapisywany w tej tabeli posiada takie pola jak nazwa, nazwa właściciela, nazwa grupy w obrębie której jest widziany parametr. Tabela `queue` przechowuje wiadomości wysyłane pomiędzy użytkownikami w postaci zserializowanej do ciągu bajtów. Razem z każdą krotką w tabeli `queue` zapisywana jest informacja o grupie, Recepcie oraz nadawcy i od-

biorcy informacji. Każda z tabel zawiera również takie pola jak id, które jest unikalnym, autoinkrementowanym numerem w całej bazie danych, oraz czas utworzenia krotki.



Rysunek 5.6: Schemat bazy danych Serwera grup

5.5.3 Grupy i użytkownicy

Użytkownicy są identyfikowani przez login. Oprócz tego przechowywane jest ich imię, nazwisko i skrót hasła.

Łączenie użytkowników w grupy realizowane jest za pomocą relacji w bazie danych. Każda z grup jest opisana przez swoją unikalną nazwę, dzięki której może być w łatwy sposób wyszukana. Przy próbie utworzenia grupy o tej samej nazwie zostanie zwrócony wyjątek a z poziomu interfejsu błąd walidacji. Tabele reprezentujące grupy i użytkowników połączone są za pomocą tabeli pośredniej `groupspermmisions`, w której zapisane są informacje o uprawnieniach, jakie posiada użytkownik w grupie. Uprawnienia nadawane są dla każdej grupy indywidualnie. Członkowie grupy posiadają prawo do pobrania listy członków grupy oraz prawo do wykonywania, czyli możliwość wysyłania wiadomości do innych użytkowników w ramach tej grupy oraz zapisywania własnych danych na Serwerze.

Twórca grupy dodatkowo posiada wpis z uprawnieniem do dodawania nowych członków.

5.6 Uwierzytelnianie, autoryzacja, obsługa błędów

Po odebraniu każdej wiadomości od użytkownika, jest on uwierzytelniany za pomocą loginu i hasła, podanych w przypadku połączeń typu HTTP w JSONie lub dla połączeń typu REST w ścieżce URL. W celu zapobiegnięcia przesyłaniu haseł w postaci jawnej, szyfrowane są one funkcją skrótu SHA-1.

Autoryzacja użytkownika odbywa się poprzez sprawdzenie uprawnień użytkownika do grupy, którą podał w danych do komunikacji. Jako pierwsze sprawdzane jest, czy użytkownik jest członkiem grupy, w ramach której wysyła wiadomość. Jeżeli użytkownik nie należy do grupy, nie posiada żadnych uprawnień w ramach tej grupy. Dostępnych jest 5 uprawnień, jakie może posiadać użytkownik dla każdej grupy do której należy:

- d (ang. delete) pozwala usuwać użytkowników z grupy
- a (ang. add) pozwala dodawać nowych użytkowników
- r (ang. read) pozwala odczytywać członków grupy oraz nie widzi grupy na swojej liście grup, do których należy.
- x (ang. execute) umożliwia wysyłanie i odbieranie wiadomości w ramach grupy.
- i (ang. invitation) oznacza, że użytkownik został zaproszony do grupy i nie potwierdził zaproszenia. Jeżeli ustawiona jest ta flaga, pozostałe uprawnienia traktowane są tak, jakby nie były ustawione. Użytkownik może tylko potwierdzić lub odrzucić zaproszenie do grupy.

Dla zapytań typu REST w przypadku nieuwierzytelnionej, nieautoryzowanej komunikacji lub w przypadku błędu zwracana jest wartość false. Przy komunikacji z użyciem protokołu HTTP zostaje wysłana odpowiedź ze statusem błędu.

5.6.1 Kolejka komunikatów

Serwer pośredniczący w komunikacji odbiera wiadomości, które do czasu wysłania do klienta docelowego są zapisywane w bazie danych. Wiadomość w bazie jest w formie zserializowanej do ciągu bajtów. Z każdej wiadomości, która zostanie odebrana pobierane są informacje niezbędne do jej zapisania oraz późniejszego zidentyfikowania, takie jak nazwa grupy, Recepty, użytkownika docelowego i nadawcy. W szczególności odbiorcą może być BROADCAST - specjalny użytkownik, oznaczający wiadomość rozgłoszeniową, która jest zapisywana w bazie tyle razy, ilu jest członków danej grupy, z wpisanym odpowiednim użytkownikiem docelowym.

5.6.2 API zarządzania grupami

Działanie API opiera się o wywołania GET lub POST protokołu REST. Dane, będące wynikiem wywołania, zwracane są w formacie JSON. API Serwera umożliwia następujące akcje:

- Lista grup do których należy użytkownik
  - GET
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/groups/user/password
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika w postaci skrótu SHA-1
- Lista grup do których użytkownik został zaproszony
  - GET
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/invitations/user/password
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika w postaci skrótu SHA-1
- Akceptacja zaproszenia przez użytkownika
  - POST
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/invitations/accept/group/user/password
  - {group}, nazwa grupy,do której akceptowane jest zaproszenie
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika w postaci skrótu SHA-1
- Wysłanie zaproszenia do użytkownika
  - POST
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/invite/member/group/user/password
  - {group}, nazwa grupy, do której wysyłane jest zaproszenie
  - {member}, nazwa użytkownika, który jest zapraszany
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika w postaci skrótu SHA-1
- Wysłanie zaproszenia do użytkownika
  - POST
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/invite/member/group/user/password
  - {group}, nazwa grupy, do której wysyłane jest zaproszenie
  - {member}, nazwa użytkownika, który jest zapraszany
  - {user}, nazwa użytkownika uprawnionego do wysyłania zaproszeń w tej grupie
  - {password}, hasło użytkownika w postaci skrótu SHA-1
- Lista użytkowników należących do grupy
  - GET
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/groups/members/group/user/password
  - {group}, nazwa grupy
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika w postaci skrótu SHA-1
- Rejestracja nowego użytkownika

  - POST
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/register/user/password/firstName/secondName
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika
  - {firstName}, imię użytkownika
  - {secondName}, nazwisko użytkownika
- Tworzenie nowej grupy
  - POST
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/groups/create/group/user/password
  - {group}, nazwa grupy
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika
- Weryfikacja hasła
  - GET
  - http://ify.cs.put.poznan.pl/Webif{y}-1.0/rest/login/user/password
  - {group}, nazwa grupy
  - {user}, nazwa użytkownika
  - {password}, hasło użytkownika

5.7 Użyte technologie

W tej części zaprezentowano opis technologii użytych bezpośrednio w implementacji składowych platformy.

- Android
  System operacyjny z rodziny Linux przeznaczony dla urządzeń mobilnych. Aktualnie rozwijanej przez sojusz biznesowy Open Handset Alliance.
- Android SDK
  Platforma programistyczna umożliwiająca tworzenie aplikacji dla systemu Android. Zawiera wtyczkę do środowiska Eclipse, narzędzia wspierające prace programisty, emulator i biblioteki potrzebne do zbudowania aplikacji. Programy dedykowane platformie pisane są w języku Java i uruchamiane na maszynie wirtualnej Dalvik.
- Apache Commons
- Apache HTTP Server
- Git
  Rozproszony oraz wieloplatformowy system kontroli wersji będący wolnym oprogramowaniem.

- HTML 5

- Hibernate
  Narzędzie odwzorowań obiektowo-relacyjnych (ang. object-relation mapping, ORM) rozwijany na zasadzie wolnego oprogramowania. Umożliwia odwzorowania obiektowo-relacyjne, pamięć podręczną, leniwe (ang. Lazy loading), chciwe pobieranie oraz rozproszoną pamięć podręczną.
- JSON
  Skrót od JavaScript Object Notation. Jest to lekki, tekstowy format wymiany danych niezależny od języka programowania. Został wybrany ze względu na swoją czytelność i wsparcie ze strony bibliotek programistycznych.
- Java

- JavaScript
  Skryptowy język oprogramowania stosowany na stronach internetowych.
- Apache Maven
  Narzędzie automatycznego budowania oprogramowania dla języka JAVA. Głównymi problemami, jakie rozwiązuje Maven przy budowaniu aplikacji, są: zarządzanie zależnościami, możliwość operowaniawieloma modułami, wsparcie dla testów.
- MySQL
  System zarządzania relacyjnymi bazami danych. Jest to wolne oprogramowanie szczególnie upodobane przez twórców aplikacji internetowych. Bardzo dobrze współpracuje z językami takimi jak PHP czy Java
- PHP
  Obiektowy język programowania dedykowany generowaniu stron internetowych w czasie rzeczywistym. Szczególnie użyteczny w przypadku tworzenia prototypów tudzież niewielkich projektów, wymagających stosunkowo niskiego poziomu abstrakcji.
- RESTeasy
  Framework oprogramowania służący do tworzenia aplikacji rozproszonych, oparty na wzorcu architektury oprogramowania Representational State Transfer(REST).
- SpringFramework
  Framework(Szkielet) tworzenia aplikacji w języku Java a w szczególności JavaEE. Do najważniejszych funkcji Springa zalicza się wstrzykiwanie zależności (ang. dependency injection, DI) oraz programowanie aspektowe (ang. aspect-oriented programming, AOP).
- Vaadin
  Framework sieciowy służący do tworzenia aplikacji sieciowych, w szczególności interfejsu użytkownika, w oparciu o Google Web Toolkit (GWT) w języku JAVA.
- JUnit
  Biblioteka służąca do tworzenia testów jednostkowych w języku Java.

5.8 Użyte narzędzia

- Apache Tomcat
  Kontener aplikacji sieciowych.
- Eclipse
  Popularne zintegrowane środowisko programistyczne (IDE) wspierające głównie język Java (wtyczki pozwalają obsługiwać inne języki).
- Android developer tools
  Wtyczka do Eclipse pozwalająca tworzyć aplikacje androidowe. Dodaje takie funkcjonalności, jak edycja plików XML odpowiadających za wygląd aplikacji (również w trybie graficznym) czy debugowanie na telefonach oraz emulatorze.
- String Tool Suite
  Zintegrowane środowisko programistyczne oparte o Eclipsa, dostosowane do SpringFramework.
- Emacs
  Popularny, w pełni rozszerzalny edytor tekstowy spotykany głównie w systemach operacyjnych z rodziny Unix.
- Git bash for windows
  Narzędzie umożliwiające używanie Gita z linii poleceń w systemie Windows poprzez wbudowane środowisko MinGW.
- Github
  Serwis internetowy gromadzący społeczność programistów z całego świata. Służy jako hosting dla otwartoźródłowych projektów, zarządzanych za pomocą systemu Git. Udostępnia szereg narzędzi wspierających - system śledzenia zadań, budowę statystyk.
- Latex
  Język znaczników oraz zbiór oprogramowania do zautomatyzowanego składania tekstu.
- Linux
  Rodzina systemów operacyjnych, będących wolnym oprogramowaniem oraz używających jądra Linux.
- Notepad++
  Prosty edytor tekstowy umożliwiający kolorowanie składni w wielu językach.
- Przeglądarki internetowe
  Google Chrome, Mozilla Firefox, Opera
- Windows

5.9 Urządzenia mobilne

Aplikacja była testowana na następujących urządzeniach mobilnych:

- LG Swift GT540
  Procesor: Qualcomm MSM7227 600 MHz Pamięć RAM: 256 MB System operacyjny: Android 4.0.1 (Cyanogen mod)
- Media-Droid IMPERIUS EN3RGY MT7013 Procesor: dwurdzeniowy, 1GHz ARM7 MTK6577 Pamięć RAM: 256 MB System operacyjny: Android 4.1.2
- Motorola Defy MB525
  Procesor: TI OMAP3610 800 MHz Pamięć RAM: 512 MB System operacyjny: Android 4.3.1 (Cyanogen mod)
- Sony LT18 Xperia Arc S
  Procesor: Qualcomm MSM8255T 1,40 GHz Pamięć RAM: 512 MB System operacyjny: Android 4.0.4
- Samsung Galaxy Mini GT-S5570
  Procesor: Qualcomm MSM7227 600 MHz Pamięć RAM: 384 MB System operacyjny: Android 2.2

5.10 Opis pakietów

5.10.1 Pakiety Aplikacji

pl.poznan.put.cs.ify.app - główny pakiet Aplikacji.

pl.poznan.put.cs.ify.jars - pakiet odpowiedzialny za zarządzanie plikami .jar zawierającymi Recepty pobrane z Targowiska.

pl.poznan.put.cs.ify.core - pakiet odpowiedzialny za zarządzanie dostępnymi i aktywowanymi Receptami.

pl.poznan.put.cs.ify.appify.receipts - pakiet zawierający Recepty wbudowane w Aplikację.

pl.poznan.put.cs.ify.app.ui - pakiet zawierający kontrolki interfejsu użytkownik.

pl.poznan.put.cs.ify.app.ui.params - pakiet zawierający kontrolki interfejsu użytkownika wykorzystywane do wprowadzania parametrów przy inicjalizacji Recepty.

pl.poznan.put.cs.ify.app.market - pakiet odpowiedzialny za pobieranie danych z Targowiska i wyświetlanie ich.

pl.poznan.put.cs.ify.app.fragments - pakiet zawierający widoki ekranów aplikacji.

5.10.2 Pakiety Biblioteki

pl.poznan.put.cs.ify.api - pakiet główny Biblioteki.

pl.poznan.put.cs.ify.api.exceptions - pakiet zawierający wyjątki, które mogą być rzucane przez metody z Biblioteki.

pl.poznan.put.cs.ify.api.features - pakiet zawierający Podfunkcjonalności i Zdarzenia.

pl.poznan.put.cs.ify.api.group - pakiet odpowiedzialny za obsługę Recept Grupowych.

pl.poznan.put.cs.ify.api.log - pakiet odpowiedzialny za obsługę logowania i domyślny widok logów.

pl.poznan.put.cs.ify.api.params - pakiet zawierający typy parametrów wykorzystywanych przez Recepty.

pl.poznan.put.cs.ify.api.security - pakiet odpowiedzialny za moduł uprawnień Biblioteki.

pl.poznan.put.cs.ify.api.types - pakiet zawierający typy danych wykorzystywanych przez Bibliotekę.

5.10.3 Pakiety Serwera

pl.poznan.put.cs.ify.webify - pakiet główny Serwera.

pl.poznan.put.cs.ify.webify.data.dao - pakiet zawierający warstwę dostępu do danych.

pl.poznan.put.cs.ify.webify.data.entity - pakiet zawierający klasy odwzorowywane na bazę danych.

pl.poznan.put.cs.ify.webify.data.enums - pakiet zawierający potrzebne w bazie danych typy wyliczeniowe(np. lista ról).

pl.poznan.put.cs.ify.webify.gui - pakiet główny graficznego interfejsu użytkownika.

pl.poznan.put.cs.ify.webify.gui.windows - pakiet zawierający wszystkie okna aplikacji sieciowej.

pl.poznan.put.cs.ify.webify.gui.components - pakiet zawierający komponenty użyte w aplikacji.

pl.poznan.put.cs.ify.webify.gui.session -

pl.poznan.put.cs.ify.webify.service - pakiet zawierający logikę.

pl.poznan.put.cs.ify.webify.rest - pakiet zawierający obsługę zapytań typu REST.

pl.poznan.put.cs.ify.webify.utils - pakiet, w którym przechowywane są funkcje pomocnicze używane w całym projekcie.

Rozdział 6

Wyniki

6.1 Testy akceptacyjne

22 stycznia odbyły się testy akceptacyjne rozwiązania. Zostały przetestowane następujące scenariusze:

1. Ściągnięcie kodu źródłowego Serwera i wykonanie załączonych instrukcji pozwala na bezbłędne jego uruchomienie i późniejsze działanie.
   - **Pobierz kod Serwera z repozytorium**
     Kod źródłowy jest kompletny
   - **W celu uruchomienia Serwera wykonaj instrukcje zamieszczone w pliku README**
     Serwer jest uruchomiony i gotowy na obsługiwanie Zdarzeń.
2. Utworzenie nowej Recepty w serwisie Marketify (+ opcja fork)
   - **Wejdź na serwis Marketify**
     Strona internetowa uruchamia się poprawnie
   - **Wybierz Receptę na której chcesz bazować i kliknij 'Fork'**
     Utworzona zostaje nowa Recepta na podstawie kodu wybranej Recepty
   - **Zmodyfikuj Receptę dla swoich potrzeb i zatwierdź**
     Nowa, zmodyfikowana Recepta jest dostępna w serwisie
3. Instalacja aplikacji mobilnej.
   - **W ustawieniach systemu Android włącz zezwolenie na instalowanie aplikacji z nieznanych źródeł**
   - **Pobierz aplikację AppIfy i zainstaluj na swoim urządzeniu mobilnym**
     Aplikacja jest dostępna w telefonie
   - **Uruchom aplikację**
     Aplikacja działa poprawnie i nie zamyka się bez wyraźnego żądania
4. Instalacja Recepty indywidualnej z Marketify
   - **Uruchom aplikację AppIfy**

- **Wybierz z menu pozycję 'Market' i wybierz Receptę z listy**
  Wyświetlone zostają dokładne informacje dotyczące wybranej Recepty
- **Pobierz Receptę**
  Recepta zostaje dodana do listy dostępnych Recept
- **Przejdź do listy dostępnych Recept i aktywuj Receptę ew. ustawiając wymagane parametry**
  Recepta została poprawnie aktywowana

5. Instalacja Recepty grupowej z Marketify

   - **Uruchom aplikację AppIfy**

   - **Wybierz z menu pozycję 'Market' i wybierz Receptę z listy**
     Wyświetlone zostają dokładne informacje dotyczące wybranej Recepty
   - **Pobierz Receptę grupową**
     Recepta zostaje dodana do listy dostępnych Recept
   - **Przejdź do listy dostępnych Recept i aktywuj Receptę ustawiając grupę oraz ew. dodatkowe parametry**
     Recepta grupowa została poprawnie aktywowana

6. Implementacja i instalacja Recepty skutkuje jej wykonaniem tylko i wyłącznie po zajściu zdefiniowanych wyzwalaczy (Zdarzeń).

   - **Zainstaluj i aktywuj Receptę**
     Recepta oczekuje na Zdarzenia
   - **Wyzwól Zdarzenie zdefiniowane w Recepcie**
     Akcja zdefiniowana w Recepcie zostaje wykonana, akcje z pozostałych Recept nie są wykonywane

7. Serwis Serverify daje możliwość rejestracji i logowania użytkowników w celu przypisywania ich do grup

   - **Zarejestruj się w serwisie**
     Zostaje utworzone nowe konto użytkownika
   - **Zaloguj się do serwisu**
     Użytkownik jest zalogowany
   - **Utwórz nową grupę lub wybierz grupę do której chcesz się dołączyć**
     Użytkownik zostaje przypisany do grupy

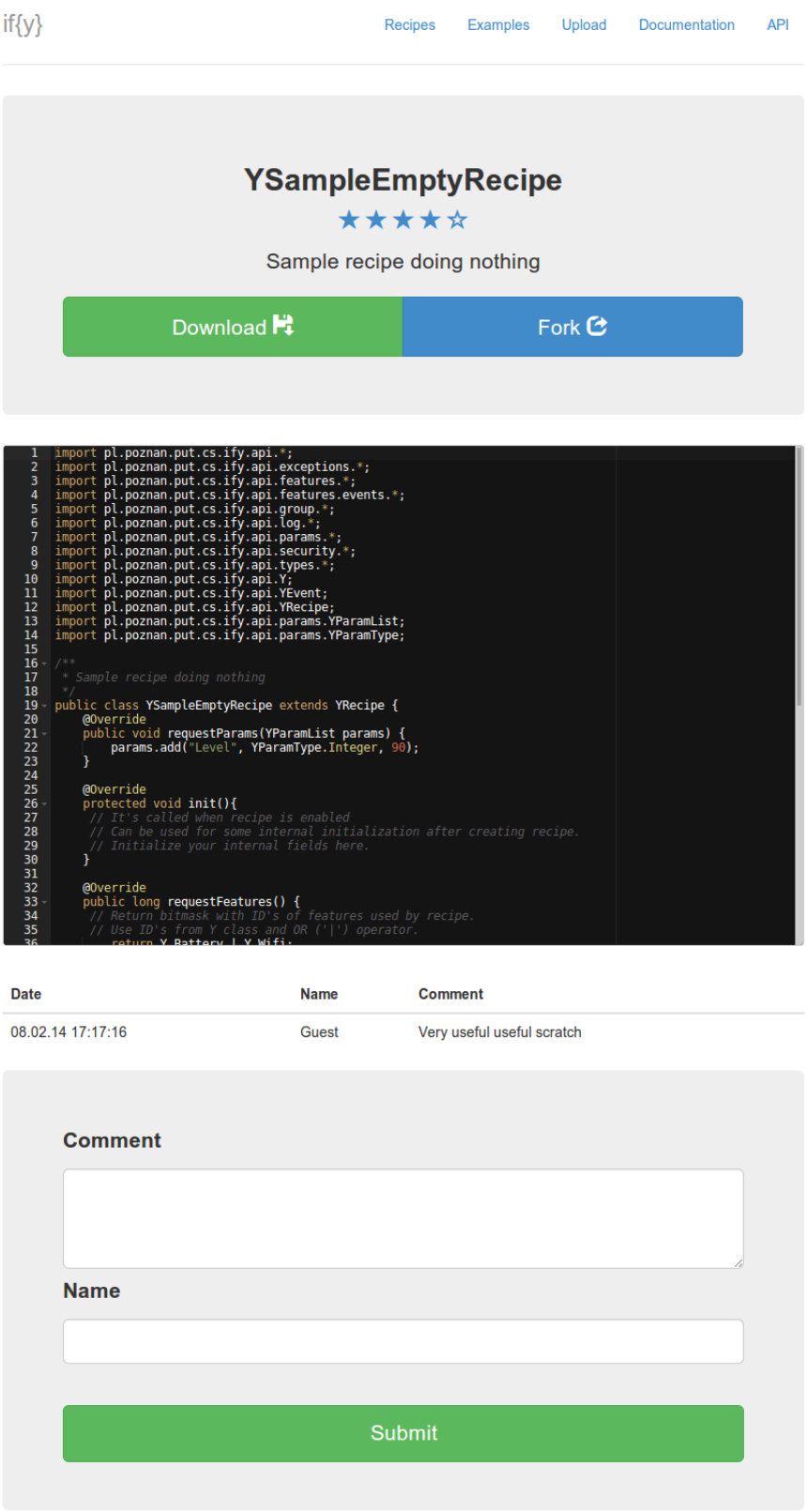
6.2 Cykl życia Recepty w praktyce

W efekcie ukończenia prac nad wszystkimi modułami platformy, możliwym stało się odtworzenie cyklu życia Recepty w praktyce. Użytkownik kompetentny do pisania kodu w języku Java, jest w stanie od tego miejsca tworzyć Recepty, ograniczając się wyłącznie interfejsem biblioteki oraz własną wyobraźnią.

W dalszej części rozdziału przedstawiona zostanie Recepta, informująca o wzajemnej bliskości osób należących do grupy, oraz przeanalizowany zostanie każdy etap jej cyklu

życia.

6.2.1 Pisanie kodu



Rysunek 6.1: Widok przykładowej Recepty w Targowisku

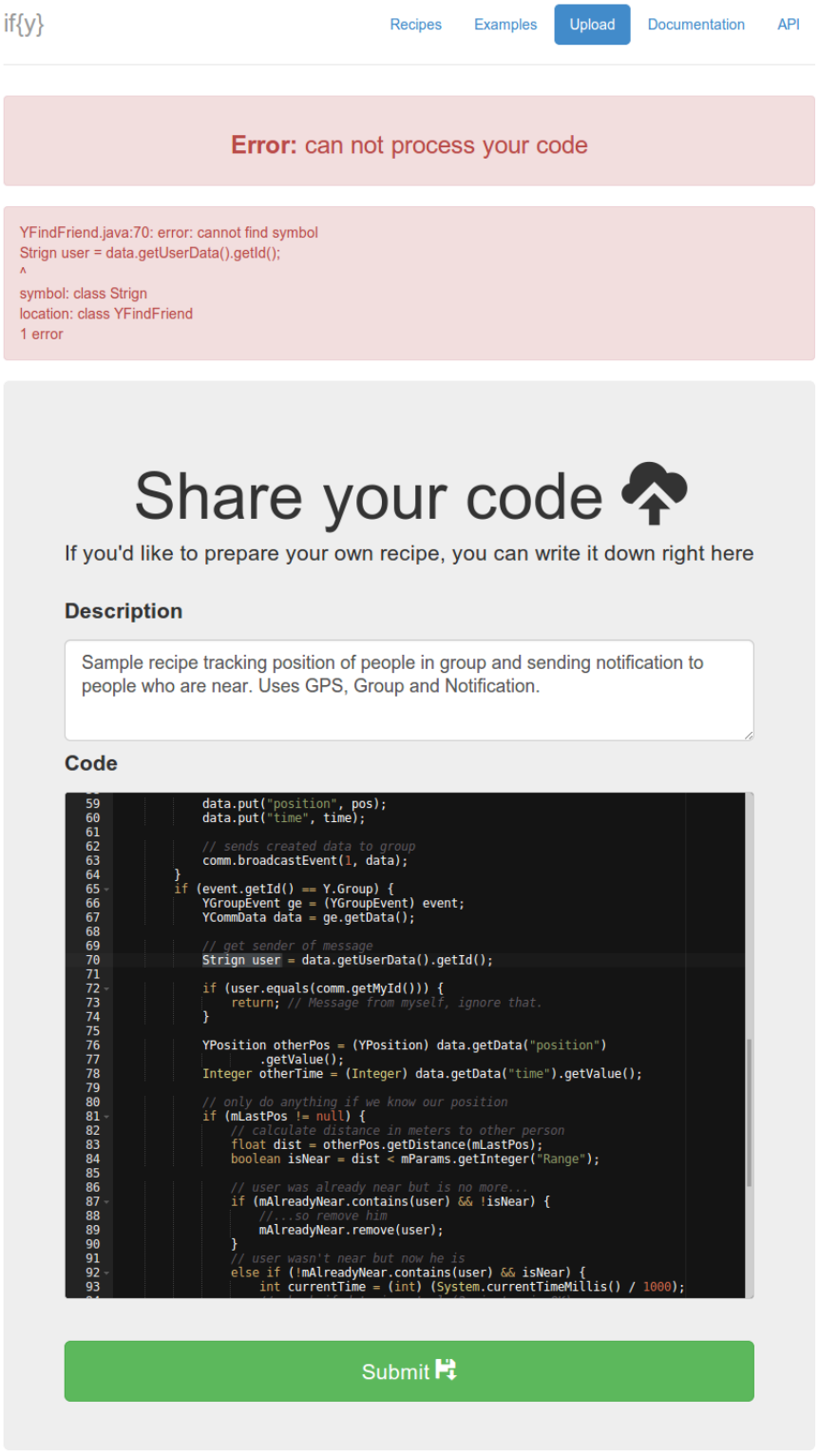
Pierwszym krokiem do napisania kodu własnej Recepty jest zapoznanie się z przykładami dostępnymi na stronie Targowiska w dziale *examples*. Przykładowa Recepta może wyglądać jak na rysunku 6.1. Każda przykładowa Recepta to działający kod, zawierający przydatne informacje zawarte w komentarzach. Gdy komentarze i przykłady nie wystarczą, sięgnąć należy do dokumentacji dostępnej w Targowisku w dziale *documentation*.

Po nabyciu niezbędnej wiedzy, najprościej zacząć od rozwidlenia gotowej Recepty. Sprowadza się to do kliknięcia przycisku *fork* z poziomu widoku Recepty oraz wprowadzenia nowej nazwy. Alternatywną drogą jest rozpoczęcie pisania Recepty od podstaw. W tym celu należy przejść do działu Targowiska o nazwie *upload*.

Pisanie kodu odbywa się w edytorze który szerzej opisany został w rozdziale 5.5.3. W celu wywołania okna autouzupełniania należy użyć skrótu klawiszowego *CTRL+SPACJA*.

6.2.2 Kompilacja i budowa

Po zakończeniu pisania kodu wystarczy nacisnąć przycisk *submit* i odczekać kilka sekund.



Rysunek 6.2: Błąd w trakcie kompilacji kodu Recepty

W przypadku niepowodzenia etapu kompilacji czy budowy, użytkownik dostaje pełną informację o błędzie jak na rysunku 6.2. Po przeanalizowaniu komunikatu o błędzie, użytkownik może wprowadzić niezbędne zmiany oraz ponownie nacisnąć przycisk *submit*.

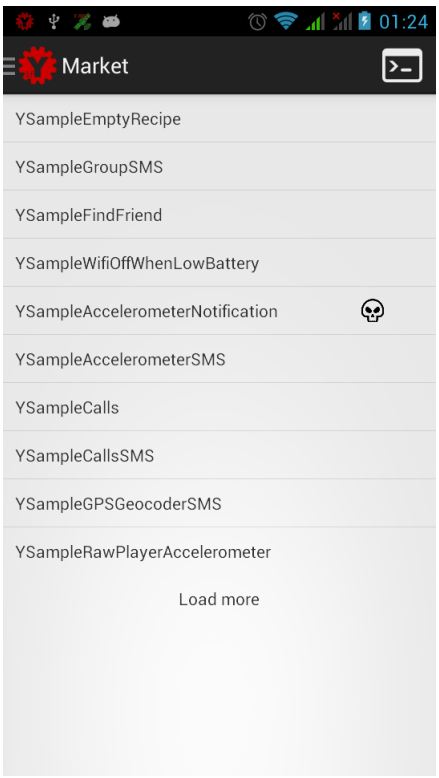
W przypadku poprawnego przetworzenia kodu Recepty, użytkownik zostaje przekierowany bezpośrednio do jej widoku.

6.2.3 Dystrybucja

Zbudowana poprawnie Recepta dostępna jest natychmiastowo zarówno w Targowisku jak i module Targowiska aplikacji if{y} w urządzeniu mobilnym. Aby Recepta zdobyła popularność i uznanie wśród społeczności, warto aby jej kod był możliwie wysokiej jakości oraz z dużą ilością komentarzy.

O sukcesie Recepty decydują oceny nadawane przez użytkowników oraz pozostawione przez nich komentarze.

6.2.4 Pobranie na urządzenie mobilne



Rysunek 6.3: Lista Recept dostępnych w Aplikacji Klienckiej if{y} za pośrednictwem Targowiska

Aby pobrać Receptę na urządzenie mobilne należy wybrać w appIFY zakładkę Market. Pojawi się wówczas lista dostępnych Recept podobna do tej na rysunku 6.3. Przy niektórych pojawiają się czaszki – oznacza to, że odwołują się do bibliotek Androida lub mechanizmu refleksji i są potencjalnie niebezpieczne – mogą uzyskać dostęp do innych funkcji telefonu niż wynika z zadeklarowanych Podfunkcjonalności. Po wybraniu Recepty wystarczy użyć przycisku Download.

6.2.5 Uruchomienie

Aby uruchomić Receptę należy wybrać zakładkę Available recipes a następnie wybrać Receptę z listy. Pojawi się okno w które należy określić parametry, a następnie użyć przycisku Start. W przypadku Recept grupowych użytkownik musi być zalogowany aby uruchomić Receptę.

6.2.6 Działanie

Po aktywacji Recepty działa ona w tle, nasłuchując Zdarzeń i reagując na nie. Działanie Recepty kończy się po wybraniu zakładki *Active recipes* z listy i kliknięciu przycisk *Disable*. Praca Recepty może się też zakończyć automatyczne w wyniku wyrzucenia przez nią wyjątku lub wylogowania się użytkownika w przypadku Recepty grupowej.

Rozdział 7

Zakończenie

7.1 Podsumowanie

W wyniku realizacji pracy inżynierskiej stworzono w pełni funkcjonującą platformę if{y}. W skład platformy wchodzą biblioteka oraz aplikacja dla systemu Android, Serwer Recept grupowych, Targowisko oraz dokumentacja techniczna.

7.2 Dalszy rozwój

Po zakończeniu prac projekt będzie dalej rozwijany, jako projekt open source. Najtrudniejszym zadaniem będzie zdobycie bazy użytkowników, którzy umożliwią wyznaczenie dobrego kierunku rozwoju. Prawdopodobnie będzie to wymagało znalezienia grafika i dopracowaniu interfejsu użytkownika. Poza tym widać co najmniej trzy kierunki, w jakich można rozwinąć projekt.

7.2.1 Rozwój Podfunkcjonalności

W pracy zaimplementowano dosyć mało Podfunkcjonalności w porównaniu z pełnym API Androida. Wynikało to z trzech faktów – po pierwsze skupiono się na ogólnym szkielecie rozwiązania oraz Receptach Grupowych, po drugie nie ma sensu tworzyć na każdą opcję w systemie nakładek, z których większość nie zostanie użyta. Tworzono więc głównie Podfunkcjonalności do których istniały przypadki użycia. Gdy pojawią się pomysły na Recepty, których obecnie nie można zrealizować, można będzie dodawać nowe Podfunkcjonalności.

7.2.2 Inne platformy

Powstała platforma ify jest obecnie dedykowana tylko części rynku związanej z systemem Android. Rodzi to potencjalną możliwość ekspansji na inne systemy operacyjne. Jeżeli platforma zostanie pozytywnie odebrana przez społeczność korzystającą z urządzeń mobilnych, w przyszłości powstaną implementacje biblioteki if{y} na systemy takie jak iOS czy Windows Phone. Niestety, wersje na inne systemy prawdopodobnie będą miały mniejszą

funkcjonalność – nie da się tam na przykład nasłuchiwać połączeń czy SMSów, nie zbadano także możliwości ładowania kodu w czasie wykonania. Architektura platformy nie wyklucza także wsparcia dla systemów, które jeszcze nie istnieją.

7.2.3 if{y} jako biblioteka

Przy tworzeniu obecnej wersji platformy skupiano się na wykorzystywaniu jej do tworzenia Recept. Znajduje się w niej jednak wiele reużywalnych fragmentów kodu, które mogą się przydać gdzie indziej. Po wykonaniu większej liczby korzystających z niej aplikacji na pewno pojawią się uwagi, co należy poprawić, aby użycie biblioteki było łatwiejsze.

Dodatek A

Recipe creation guide

Best way to create recipe is forking existing one from the market - just choose one from Examples or Recipes tab. Then you can edit it's code in browser. You don't have to care about ify imports - they will be added automatically when uploading recipe. More information is available in Doclava tab in market.

Currently market is available here:

http://ify.cs.put.poznan.pl/~scony/marketify

You can access some fields and methods generated by system which can be useful:

getParams()
: List of params requested in requestParams. It's populated with values when init() and handleEvent() code is called. Use getString(String name), getInteger(String name), etc. to get param with given name depanding what type param is.

getFeatures()
: Contains required features. Use specified getters like getSMS(), getWifi(), etc. or use general method get(Long id) inserting id from Y and cast returned feature to right type.

Log (it's field)
: Use if to create logs with methods wtf(String), e(String), w(String), i(String), d(String), v(String). These logs will be visible on recipe screen and saved to Android logs (use LogCat to see it)

Here is quick overview of methods you need to implement in your recipe, you can find same information in YEmptyRecipe example.

protected void init()
: It's called when recipe is enabled, can be used for some internal initialization after creating recipe. Initialize your internal fields here.

public long requestFeatures()
: Return bitmask with ID's of features used by recipe. Use ID's from Y class and OR ('|') operator.

public void requestParams(YParamList params)

Populate params object with definitions of needed params using params.add(String paramName, YParam param)

protected void handleEvent(YEvent event)

It's place for main recipe logic. It's called on event connected with requested feature occurs. Check what event is that by checking event.getId() and compare it to id's from Y

public String getName()

Return String same like your recipe class name here. It's needed to avoid using reflection which is slow.

public YReceipt newInstance()

Just call default constructor and return recipe instance. Also needed to avoid reflection.

Dodatek B

Server setup guide

In order to setup server it's necessary to:

1. Install MySQL database and MySQL client
2. Run MySQL: `mysql -u YOUR_MYSQL_USER -password=YOUR_MYSQL_PASSWORD`
3. Evaluate following commands:

```
CREATE USER 'malinka'@'localhost' IDENTIFIED BY 'St3g0zaur';

GRANT ALL PRIVILEGES ON *.* TO 'malinka'@'localhost';

CREATE DATABASE IF NOT EXISTS `ify` DEFAULT CHARACTER SET utf8 COLLATE utf8_polish_ci;

USE `ify`;

CREATE TABLE IF NOT EXISTS `grouppermissions` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sys_created_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `a` bit(1) NOT NULL,
  `d` bit(1) NOT NULL,
  `r` bit(1) NOT NULL,
  `x` bit(1) NOT NULL,
  `group_id` bigint(20) DEFAULT NULL,
  `name` varchar(255) COLLATE utf8_polish_ci DEFAULT NULL,
  `user_id` bigint(20) DEFAULT NULL,
  `username` varchar(255) COLLATE utf8_polish_ci DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_id` (`user_id`,`name`,`username`),
  UNIQUE KEY `group_id` (`group_id`,`name`,`username`),
  UNIQUE KEY `name` (`name`,`username`),
  UNIQUE KEY `group_id_2` (`group_id`,`user_id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_polish_ci AUTO_INCREMENT=123 ;

CREATE TABLE IF NOT EXISTS `groups` (
```

```
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sys_created_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `name` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  `user_id` bigint(20) DEFAULT NULL,
  `username` varchar(255) COLLATE utf8_polish_ci DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`),
  UNIQUE KEY `name_2` (`name`,`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_polish_ci AUTO_INCREMENT=95 ;

CREATE TABLE IF NOT EXISTS `parameters` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sys_created_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `booleanValue` bit(1) DEFAULT NULL,
  `device` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  `doubleValue` double DEFAULT NULL,
  `integerValue` int(11) DEFAULT NULL,
  `lobValue` longblob,
  `name` varchar(25) COLLATE utf8_polish_ci NOT NULL,
  `recipe` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  `stringValue` varchar(255) COLLATE utf8_polish_ci DEFAULT NULL,
  `type` varchar(25) COLLATE utf8_polish_ci NOT NULL,
  `groupname` varchar(255) COLLATE utf8_polish_ci DEFAULT NULL,
  `group_id` bigint(20) DEFAULT NULL,
  `username` varchar(255) COLLATE utf8_polish_ci DEFAULT NULL,
  `user_id` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK1B57C1EADF8E90FE` (`username`,`user_id`),
  KEY `FK1B57C1EA329EA100` (`groupname`,`group_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_polish_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `queue` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sys_created_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `data` longblob NOT NULL,
  `source_user_id` bigint(20) DEFAULT NULL,
  `target_user_id` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK66F19111934FF7A` (`target_user_id`),
  KEY `FK66F191126D4CC04` (`source_user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_polish_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `users` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sys_created_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
  `enabled` bit(1) NOT NULL,
  `firstName` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  `lastName` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  `password` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  `rolesLob` longblob NOT NULL,
  `username` varchar(255) COLLATE utf8_polish_ci NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `username` (`username`),
  UNIQUE KEY `username_2` (`username`,`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_polish_ci AUTO_INCREMENT=171 ;

ALTER TABLE `parameters`
  ADD CONSTRAINT `FK1B57C1EA329EA100` FOREIGN KEY (`groupname`, `group_id`)
  REFERENCES `groups` (`name`, `id`),
  ADD CONSTRAINT `FK1B57C1EADF8E90FE` FOREIGN KEY (`username`, `user_id`)
  REFERENCES `users` (`username`, `id`);

ALTER TABLE `queue`
  ADD CONSTRAINT `FK66F19111934FF7A` FOREIGN KEY (`target_user_id`)
  REFERENCES `users` (`id`),
  ADD CONSTRAINT `FK66F191126D4CC04` FOREIGN KEY (`source_user_id`)
  REFERENCES `users` (`id`);
```

4. Install tomcat 6
5. Identify tomcat's webapp directory (OpenSUSE: `/srv/tomcat/webapps/`)
6. `git clone` https://github.com/patdab90/ServerIfy.git
7. Copy **.war* file form `target/` to the tomcat's webapp directory

Dodatek C

Market setup guide

In order to setup market it's necessary to:

1. Install apache2
2. Start apache2
3. Change directory to `htdocs/`
4. git clone https://github.com/Scony/marketify.git
5. Install MySQL database & MySQL client
6. Start *mysqld*
7. `mysql -u YOUR_MYSQL_USER -password=YOUR_MYSQL_PASSWORD`
8. Evaluate commands from `htdocs/maretify/sql/db.sql`

Bibliografia

[1] E. Gamma. *Design Patterns, First edition*. Person Education, Inc, 1995.

[2] C. Walls. *Spring in action, 3rd edition*. Manning Publication Co, 2011.

[3] Vaadin https://vaadin.com/book/vaadin6/-/page/preface.html Ostatnio odwiedzone 31.01.2014.

[4] Projekt on{X} http://www.onx.ms/#!findOutMorePage Ostatnio odwiedzone 31.01.2014.

[5] The Reflection API http://docs.oracle.com/javase/tutorial/reflect/index.html Ostatnio odwiedzone 31.01.2014.

[6] Android API Guide - Service http://developer.android.com/guide/components/services.html Ostatnio odwiedzone 31.01.2014.

[7] Android API Guide - Messenger http://developer.android.com/guide/components/bound-services.html Ostatnio odwiedzone 31.01.2014.

[8] Android API Guide - Intents and Intent Filters http://developer.android.com/guide/components/intents-filters.html Ostatnio odwiedzone 31.01.2014.

[9] Android API Guide - Reading and Writing Logs http://developer.android.com/tools/debugging/debugging-log.html Ostatnio odwiedzone 31.01.2014.

[10] Takser na Google Play https://play.google.com/store/apps/details?id=net.dinglisch.android.taskerm&hl=pl Ostatnio odwiedzone 31.01.2014.