INTELIGENTNE USŁUGI INFORMACYJNE

„Wyszukiwarka zdjęć”

Karol Czop gr. 1ID21A

Kinga Krawczyk gr. 1ID22B

Andrzej Żak gr. 1ID22B

26 stycznia 2018

1. Wstęp

Zadaniem projektu jest implementacja systemu będącego wyszukiwarką zdjęć. Aplikacja umożliwia tworzenie galerii - dodawanie i opisywanie zdjęć za pomocą tagów, a także wyszukiwanie zdjęć po formacie, rozmiarze, kategorii i treści (tagach). Projekt został napisany w technologii Java oraz Angular. Dane przechowywane są w bazie danych MySQL.

2. Opis użytych technologii

2.1. REST

REST (Representational state transfer) to styl architektury, który zaproponował Roy Fielding w swojej rozprawie doktorskiej w 2000 roku [1]. Jest to opis jak, według niego, powinna działać sieć Internet. W architekturze REST dane są traktowane jako zasoby do których dostęp jest możliwy dzięki identyfikatorom URI (zazwyczaj hiperłącza). Działania na zasobach są wykonywane przy pomocy prostych, jasno zdefiniowanych operacji. Architektura ta wymusza wykorzystanie architektury klient-serwer oraz preferuje wykorzystanie bezstanowego protokołu komunikacji (najczęściej HTTP). Zasady ułatwiające tworzenie aplikacji REST [2]:

- identyfikacja zasobów poprzez URI - każdy zasób powinien mieć swój adres, być jednoznacznie identyfikowanym przez URI,
- jednolity interfejs - operacje wykonywane na zasobie są jasno określone poprzez stosowanie odpowiedniej metody HTTP:
  - GET - pobranie zasobu,
  - PUT - stworzenie lub nadpisanie zasobu,
  - POST:
    - stworzenie zasobu podrzędnego(gdy to serwer decyduje pod jakim URI będzie dostępny zasób),
    - dodanie danych do istniejącego zasobu,
  - DELETE - usunięcie zasobu,
  - HEAD - pobranie metadanych zasobu,
  - OPTIONS - pozwala poznać, co można zrobić z zasobem tzn. jakich metod HTTP można na nim użyć,

- samoopisujące się komunikaty - metadane zasobu są dostępne, na ich podstawie można określić sposób reprezentacji zasobu,
- samowystaczalne żądania - interakcja z zasobem jest bezstanowa, wszystkie potrzebne informacje znajdują się w żądaniu.

Obecnie zastosowanie mają głównie metody GET, która jest wykorzystywana zgodnie z założeniami oraz POST która zastąpiła inne metody. W takim przypadku sposób wykorzystania zasobu zakodowany jest wewnątrz zapytania. Często jest to prostszy i wygodniejszy sposób reprezentowania skomplikowanych operacji na wielu zasobach.

2.2. Java Enterprise Edition

Java Enterprise Edition (Java EE, JEE) to platforma stworzona w celu ułatwienia tworzenia aplikacji typu enterprise [3]. Głównym celem Javy EE jest uproszczenie procesu wytwarzania oprogramowania przez dostarczenie programistom rozbudowanego API zapewniającego m.in. przenośność i bezpieczeństwo. W platformie JEE znajduje zastosowanie model aplikacji wielowarstwowej. Aplikacja składa się z komponentów, które mogę być instalowane na różnych maszynach w zależności od warstwy do której należą. Te warstwy to:

- warstwa klienta działająca na maszynie klienta, uogólniając jest to klient dostarczający interfejsu użytkownika,
- warstwa sieciowa działająca na serwerze JEE, odpowiada za generowanie dynamicznych stron internetowych, warstwa ta występuje wtedy, gdy klientem może być przeglądarka internetowa,
- warstwa logiki biznesowej działająca na serwerze JEE, odpowiada za wykonywanie logiki aplikacji,
- warstwa danych działająca na serwerze EIS (Enterprise Information System), udostępnia warstwie biznesowej dane, które mogą się znajdować np. w bazie danych.

Ze względu na to że warstwy sieciowa i logiki biznesowej często działają na tym samym serwerze JEE, architektura ta jest nazywana architekturą trójwarstwową. Specyfikacja Javy EE definiuje takie komponenty JEE jak:

- aplikacje klienckie uruchamiane po stronie klienta,
- komponenty Java Servlet, JavaServer Faces (JSF), JavaServer Pages (JSP), działające w warstwie sieciowej,
- komponenty Enterprise JavaBeans (EJB), działające w warstwie logiki biznesowej.

Komponenty Javy EE są tworzone podobnie jak inne programy w języku Java. Istotną różnicą jest to że są one składane w aplikacje Javy EE, wdrażane na serwerze Javy EE i przez niego zarządzane.

2.2.1. Kontener Javy Enterprise Edition

Kontener Javy EE jest interfejsem pośredniczącym pomiędzy komponentami aplikacji Javy EE, a funkcjami specyficznymi dla platformy, na której uruchomiana jest ta aplikacja. Dzięki niemu jest ona niezależna od szczegółów interfejsu systemu operacyjnego, czy platformy sprzętowej.

2.2.2. Enterprise JavaBeans

Komponenty EJB są komponentami realizującymi logikę biznesową aplikacji tzn. jej funkcjonalność. Komponenty EJB działają wewnątrz kontenera EJB, dzięki czemu ich twórca może skupić się na implementowaniu rozwiązań dotyczących konkretnej aplikacji. Usługi, takie jak zarządzanie transakcjami czy autoryzacja są realizowane przez kontener. Rodzaje komponentów EJB:

- sesyjne, zawierające logikę, która może zostać wywołana programistycznie przez klienta
  - stanowe, zawierające stan sesji pomiędzy klientem a komponentem, dostęp do takiego komponentu ma tylko jeden klient,

  - bezstanowe, nie zawierające stanu sesji, dane dotyczące klienta są w nim przechowywane tylko na czas obsługi żądania,
  - singleton, posiada tylko jedną instancję w aplikacji,

- sterowane komunikatami, które pozwalają na asynchroniczne przetwarzanie komunikatów.

2.2.3. Java Persistance API

Java Persistance API to standard Javy dla warstwy danych. Wykorzystuje on mapowanie obiektowo-relacyjne w celu stworzenia połączenia między modelem obiektowym a relacyjną bazą danych. Na model obiektowy składają się klasy encji, które reprezentują tabele z bazy danych. Obiekt klasy encji odpowiada wierszowi w tabeli. Encje to zwykłe obiekty POJO (Plain Old Java Object), które spełniają dodatkowo następujące warunki:

- posiadają adnotacje `@Entity (javax.persistance.Entity)`,
- posiadają publiczny lub chroniony konstruktor bezargumentowy,
- nie mogą być zadeklarowane jako `final`,
- implementują interfejs `Serializable`,
- udostępniają publicznie zmienne trwałości jedynie poprzez metody.

JPA może być także wykorzystywane w aplikacjach Javy SE.

2.2.4. Java API for RESTful Web Services

Java API for RESTful Web Services (JAX-RS) to kolejny standard obecny w Javie EE [4]. Dostarcza on narzędzi ułatwiających tworzenie usług internetowych zgodnych z architekturą REST. JAX-RS wykorzystuje adnotacje, które oznaczają klasy i metody które mają być elementem interfejsu REST. Do tych adnotacji należą:

- @Path oznaczająca pod jaką ścieżką będzie widoczna dana metoda, służy do oznaczania metod oraz klas (właściwa ścieżka powstanie z połączenia ścieżki do klasy oraz ścieżki metody),
- @Consumes oznaczająca rodzaj danych jakie przyjmuje dana klasa/metoda, wykorzystuje typy MIME,
- @Produces oznaczająca rodzaj danych jakie zwraca dana klasa/metoda, wykorzystuje typy MIME,
- @PathParam służąca do oznaczenia parametru funkcji, oznacza że parametr ten zostanie odczytany ze ścieżki (w ścieżce jest on oznaczony za pomocą nawiasów klamrowych)
- @QueryParam służąca do oznaczenia parametru funkcji, który będzie podany jako parametr zapytania,
- @GET, @POST, @PUT oraz inne odzwierciedlające metody HTTP, służą one do oznaczenia jakie zapytania będą obsługiwane przez daną metodę,
- @Provider służąca do oznaczenia m.in. filtrów, przez które będą przechodziły zapytania.

2.3. Angular

Angular to framework do budowania aplikacji klienckich w HTML i JavaScript lub języka podobnego do TypeScript, który kompiluje do JavaScript. Struktura składa się z kilku bibliotek, niektóre z nich są podstawowe, a niektóre opcjonalne.

Aplikacje w Angularze składają się z szablonów (*templates*) HTML, komponentów (*components*), które zarządzają tymi szablonami, usług (*services*) zawierających logikę aplikacji oraz modułów (*modules*) opakowujących komponenty i usługi. [6]

1. **Moduły**

   Angular posiada własny system modułów *NgModules*. Każda aplikacja w Angularze ma co najmniej jedną klasę *NgModule*. *NgModule* to funkcją dekorującą, która pobiera pojedynczy obiekt metadanych, którego właściwości opisują moduł. Najważniejsze właściwości to:

   - deklaracje (*declarations*) - klasy widoku (*view classes*) należące do tego modułu. Angular ma trzy rodzaje klas widoku: komponenty, dyrektywy i potoki.
   - eksporty (*exports*) - podzbiór deklaracji, które powinny być widoczne i możliwe do wykorzystania w szablonach komponentów innych modułów.
   - importy (*imports*) - inne moduły, których wyeksportowane klasy są wymagane przez szablony komponentów zadeklarowane w tym module.
   - dostawcy (*providers*) - twórcy usług, które ten moduł udostępnia do globalnego zbioru usług. Są dostępne we wszystkich częściach aplikacji.
   - bootstrap - główny widok aplikacji, zwany komponentem głównym, który obsługuje wszystkie inne widoki aplikacji.

2. **Komponenty**

   Komponent steruje działaniem widoków - w klasie definiowana jest logika komponentu aplikacji. Klasa komunikuje się z widokiem za pośrednictwem API. Angular tworzy, aktualizuje i niszczy komponenty, w trakcie gdy użytkownik porusza się po aplikacji.

3. **Szablony**

   Widoki komponentu są definiowane za pomocą szablonów. Szablon jest formą HTML, która mówi Angularowi, jak renderować komponent.

2.4. MySQL

MySQL jest popularnym systemem do zarządzania bazą danych typu open source. Jest on rozwijany i wspierany przez Oracle Corporation. Baza danych MySQL jest relacyjną

bazą danych, która wykorzystuje język SQL. [7]

- działa na wielu różnych platformach
- zawiera system uprawnień i haseł, który jest bardzo elastyczny i bezpieczny oraz umożliwia weryfikację na podstawie hosta
- posiada wiele typów danych, m.in: INTEGER, FLOAT, DOUBLE, CHAR, VARCHAR, BINARY, VARBINARY, TEXT, BLOB, DATE, TIME, DATETIME, TIMESTAMP, YEAR, SET, ENUM
- całkowicie wspiera instrukcje SELECT, GROUP BY, ORDER BY, COUNT(), AVG(), STD(), SUM(), MAX(), MIN(), GROUP_CONCAT(), LEFT OUTER JOIN, RIGHT OUTER JOIN, DELETE, INSERT, REPLACE, UPDATE
- wspiera aliasy na tabelach i kolumnach
- przykładowe programy klienckie: mysqldump i mysqladmin (uruchamiane z konsoli) oraz MySQL Workbench.

3. Projekt aplikacji

3.1. Założenia

Założenia aplikacji:

- możliwość utworzenia konta i zalogowania się na nie,
- możliwość dodawania zdjęć z opisem (tagami),
- możliwość wyszukiwania zdjęć po formacie,
- możliwość wyszukiwania zdjęć po rozmiarze,
- możliwość wyszukiwania zdjęć po kategorii,
- możliwość wyszukiwania zdjęć po tagach.

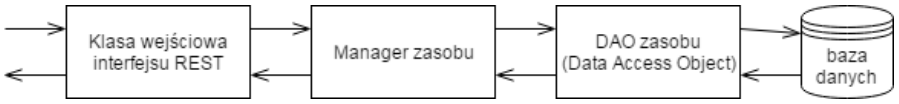
3.2. Architektura

Ze względu na specyfikę aplikacji wybrano architekturę klient-serwer. Część serwerowa umożliwia przechowywanie informacji o zdjęciach i użytkownikach, zaś część kliencka pozwala na przeglądanie zdjęć oraz wyszukiwanie ich według określonych kryteriów. Informacje o użytkownikach i zdjęciach będą zapisywane w bazie danych MySQL, znajdującej się na tej samej maszynie co część serwerowa.

Klient do działania będzie wymagał dostępu do Internetu oraz przeglądarki internetowej. Aplikacja pobierze zdjęcia z bazy danych za pośrednictwem serwera. Obrazy nie będą fizycznie przechowywane w bazie danych - w tabeli zostanie zapisany jedynie odnośnik do lokalizacji na serwerze.

Połączenie pomiędzy klientem a serwerem zostało oparte na architekturze REST. Serwer zaprojektowano i podzielono na warstwy tak, aby dostęp do niego był możliwy niezależnie od rodzaju klienta. Wydzielono warstwę danych, logiki, a także warstwę udostępniającą interfejs REST, który serwuje dane w formacie JSON. W realizowanym projekcie

komunikacja pomiędzy klientem, a serwerem będzie odbywała się poprzez ten interfejs, za pomocą żądań HTTP. Ogólny schemat przetwarzania takiego żądania przedstawiono na rysunku 1.



Rysunek 1: Przetwarzanie zapytania na serwerze
Źródło: opracowanie własne

3.3. Interfejs REST

Tworząc interfejs oparty na architekturze REST należało wyodrębnić zasoby, które będą udostępniane klientom. Wyróżniono trzy: użytkownicy, zdjęcia, tagi. Każdemu z zasobów został przypisany adres:

- `http://adres.serwera/TestProject/api/user/` udostępnia zasób „użytkownicy” (tabela 1),
- `http://adres.serwera/TestProject/api/image/` udostępnia zasób „zdjęcia” (tabela 2),
- `http://adres.serwera/TestProject/api/tag/` udostępnia zasób „tagi” (tabela 3).

Tabela 1: Adresy zasobu „użytkownicy”

adres	metoda	operacja
`/user/`	POST	rejestracja użytkownika
`/user/login`	POST	logowanie użytkownika
`/user/login`	DELETE	wylogowanie użytkownika

Tabela 2: Adresy zasobu „zdjęcia”

adres	metoda	operacja
`/image/`	GET	pobranie wszystkich zdjęć
`/image/`	POST	wgranie zdjęcia
`/image/{id}`	GET	pobranie informacji o zdjęciu o id={id}
`/image/details/{id}`	GET	pobranie dokładnych informacji o zdjęciu o id={id}
`/image/size/size`	GET	wyszukiwanie zdjęć po rozmiarze={size}
`/image/user/userID`	GET	wyszukiwanie zdjęć użytkownika o id={userID}
`/image/extension/extension`	GET	wyszukiwanie zdjęć po rozszerzeniu={extension}
`/image/tag/tag`	GET	wyszukiwanie zdjęć z tagiem={tag}
`/image/category/category`	GET	wyszukiwanie zdjęć z kategorii={category}

Tabela 3: Adresy zasobu „tagi”

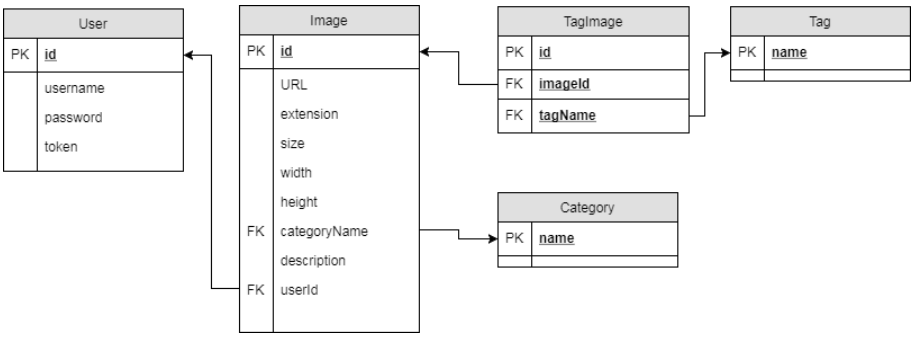
adres	metoda	operacja
`/tag/All`	GET	pobranie wszystkich tagów

4. Baza danych

W celu przechowywania danych zaprojektowano bazę danych przedstawioną na rysunku 2. Sama baza danych zostanie stworzona z klas modelu dzięki użyciu standardu JPA. Na bazę składa się pięć tabel, ich szczegółowy opis przedstawiono poniżej.

- Image - tabela przechowująca informacje o zdjęciach
  - id - identyfikator
  - URL - adres URL do lokalizacji zdjęcia na serwerze
  - extension - rozszerzenie
  - size - rozmiar zdjęcia
  - width - szerokość zdjęcia

  - height - wysokość zdjęcia
  - categoryName - klucz obcy do tabeli Category
  - description - opis zdjęcia
  - userId - klucz obcy do tabeli User
- User - tabela przechowująca dane o użytkownikach
  - id - identyfikator
  - username - login użytkownika
  - password - hash hasła użytkownika
  - token - token uwierzytelniający użytkownika
- Category - tabela przechowująca informacje o kategoriach
  - name - nazwa kategorii
- Tag - tabela przechowująca informacje o tagach
  - name - nazwa tagu
- TagImage - tabela łącząca zdjęcie z tagiem, przechowuje informacje o tagach przypisanych do zdjęć
  - id - identyfikator
  - imageId - klucz obcy do tabeli Image
  - tagName - klucz obcy do tabeli Tag



Rysunek 2: Schemat bazy danych

Źródło: opracowanie własne

5. Implementacja

5.1. Narzędzia

W celu ułatwienia pracy kilku osób nad projektem zostało wykorzystane narzędzie GitHub [8]. Zastosowano również ciągłą integrację wykorzystując Travis CI [9]. Do automatyzuji budowy oprogramowania na platformę Java użyto Apache Maven [10].

5.2. Środowisko

Do stworzenia klienta aplikacji użyto środowiska Visual Studio Code. Część serwerowa powstała w środowisku Eclipse EE z wykorzystaniem serwera aplikacyjnego WildFly [5] oraz bazy danych MySQL. Tabele bazy danych zostaną utworzone przy pierwszym uruchomieniu serwera. Należy skonfigurować serwer WildFly tak, aby mógł korzystać z baz danych MySQL.

5.3. Serwer

Do zadań serwera należy zarządzanie przechowywanymi informacjami oraz ich udostępnianie. Został on stworzony w technologii Java EE. Kod podzielono na warstwy: danych (model), logiki (managery), oraz interfejs wejściowy (REST). Warstwa danych odpowiada za przechowywanie informacji w bazie danych oraz dostęp do nich. Warstwa logiki realizuje funkcje takie jak np. wgrywanie zdjęcia, czy rejestracja użytkownika. Interfejs REST przyjmuje żądania HTTP, wyodrębnia z nich dane i przekazuje je do warstwy logiki. Generując odpowiedź tworzy on odpowiedź HTTP, ustalając dla niej m.in. odpowiedni kod odpowiedzi oraz treść.

5.3.1. Model

W celu stworzenia bazy danych przedstawionej w projekcie aplikacji utworzono klasy modelowe. Są to zwykłe klasy Javy opatrzone specjalnymi adnotacjami. Umieszczone

one zostały w pakiecie photoGallery.model. Przykład takiej klasy przedstawiono na listingu 1. Za pomocą adnotacji są w niej tworzone klucze główne (adnotacja @Id oraz @GeneratedValue do automatycznego generowania wartości) oraz klucze obce (adnotacje @ManytoOne oraz @JoinColumn). Dodatkowo definiowane są także zapytania, które będą mogły być później wykorzystane (@NamedQuery).

Listing 1: Fragment klasy modelu Image, pominięto metody oraz pola nie posiadające adnotacji

```
1  @Entity
2  @NamedQueries({ @NamedQuery(name = "Image.findAll", query = "SELECT
       i FROM Image i"),
3      @NamedQuery(name = "Image.findBySize", query = "SELECT i FROM
          Image i WHERE i.size >= :minSize AND i.size <= :maxSize"),
4      @NamedQuery(name = "Image.findByExtension", query = "SELECT i
          FROM Image i WHERE i.extension = :extension"),
5      @NamedQuery(name = "Image.findByTag", query = "SELECT i FROM
          Image i, TagImage ti WHERE ti.tag = :tag AND ti.image = i"),
6      @NamedQuery(name = "Image.findByUser", query = "SELECT i FROM
          Image i WHERE i.user = :user"),
7      @NamedQuery(name = "Image.findByCategory", query = "SELECT i
          FROM Image i WHERE i.category = :category"),
8      @NamedQuery(name = "Image.findAllNames", query = "SELECT concat(
          i.url,'.',i.extension) FROM Image i") })
9  public class Image {
10   @Id
11   @GeneratedValue(strategy = GenerationType.IDENTITY)
12   private long id;
13   private String url;
14   private String extension;
15   private int size;
16   private int width;
17   private int height;
18   @ManyToOne
19   @JoinColumn(name = "categoryName")
20   private Category category;
```

```
21    private String description;
22    @ManyToOne
23    @JoinColumn(name = "userId")
24    private User user;
25    @OneToMany(mappedBy = "image")
26    private List<TagImage> tagImages;
27    ...
28 }
```

Do wykonywania operacji na bazie danych wykorzystano klasy DAO. Każda klasa modelu posiada swoją własną klasę DAO. Klasy te można znaleźć w pakiecie photoGallery.dao. Ze względu na to, że wszystkie posiadają podobne funkcje stworzono abstrakcyjną klasę generyczną GenericDAO. Dostarcza ona podstawowych operacji takich jak wyszukanie w bazie obiektu po identyfikatorze, jego tworzenie, modyfikacja oraz usuwanie, a także pobieranie wszystkich obiektów. Klasa ta została przedstawiona na listingu 2. DAO konkretnej klasy rozszerza generyczne DAO oraz, jeśli to potrzebne, dodaje dodatkowe metody. Przykład klasy DAO pokazano na listingu 3. Rozszerza ona klasę GenericDAO podając typ danych na jakich będzie operowała oraz implementuje metodę GetClassType().

Listing 2: Klasa GenericDAO

```
1  public abstract class GenericDAO<T> {
2    @PersistenceContext
3    protected EntityManager entityManager;
4
5    public T create(T entity){
6      entityManager.persist(entity);
7      return entity;
8    }
9
10   public void remove(T entity) {
11     entityManager.remove(entity);
12   }
13
14   public void update(T entity) {
15     entityManager.merge(entity);
16   }
17
18   public T find(Object entityId) throws EntityNotFound{
19     T tmp=(T) entityManager.find(getClassType(), entityId);
20     if(tmp==null){
21       throw new EntityNotFound();
22     }
23     return tmp;
24   }
25   public List<T> getAll(){
26     Query q = entityManager.createNamedQuery(getClassType().
         getSimpleName()+".findAll", getClassType());
27     List<T> resultList = q.getResultList();
28     return resultList;
29   }
30   protected abstract Class<?> getClassType();
31 }
```

Listing 3: Klasa UserDAO

```
1  @Stateless
2  public class UserDAO extends GenericDAO<User> {
3
4    @Override
5    protected Class<?> getClassType() {
6      return User.class;
7    }
8
9    public User getByUsername(String username) throws EntityNotFound {
10     Query q = entityManager.createNamedQuery("User.findByUsername",
           User.class);
11     q.setParameter("username", username);
12     User user = null;
13     try {
14       user = (User) q.getSingleResult();
15     } catch (NoResultException e) {
16       throw new EntityNotFound();
17     }
18     return user;
19   }
20 }
```

5.3.2. Logika aplikacji

Logika aplikacji, realizująca jej funkcjonalności po stronie serwera, została umieszczona w klasach Managerów (znajdującą się one w pakiecie photoGallery.manager). Są to bezstanowe komponenty sesyjne EJB. Każda ich metoda realizuje jedno polecenie, które jest niezależne od innych. Managery posiadają dostęp do DAO dzięki wstrzykiwaniu zależności (ang. dependency injection). Na listingu 4 przedstawiono klasę UserManager, która zajmuje się operacjami na użytkownikach - procesem rejestracji i logowania. Adnotacja w wierszu pierwszym informuje o tym, że klasa ta jest bezstanowym EJB. Wiersze 4-5 odpowiadają za wstrzyknięcie DAO odpowiedzialnego za operacje na tabeli przechowującej dane użytkowników. Metoda loginUser() (wiersze 9-21) przyjmuje jako parametry

login oraz hasło użytkownika. Po udanym logowaniu zwracany jest obiekt użytkownika, w przeciwnym wypadku zgłaszany jest wyjątek logowania. Występuje on w przypadku gdy hasło jest niepoprawne, a także gdy nie istnieje użytkownik o podanym loginie. Proces logowania przebiega następująco:

- pobranie użytkownika o podanym loginie, jeśli nie istnieje zgłoszenie wyjątku (wiersze 10-12),
- wygenerowanie hashu z hasła (wiersz 13. oraz metoda generateHash() w wierszach 27-42),
- porównanie wygenerowanego hashu z zapisanym w bazie oraz zgłoszenie wyjątku jeśli nie są identyczne (wiersze 13-16),
- wygenerowanie tokena i zapisanie go w bazie (wiersze 17-19 oraz metoda generateToken() w wierszach 23-25),
- czyszczenie pola zwierającego hash hasła, modyfikacja ta dotyczy tylko obiektu zwracanego przez funkcje, nie ma wpływu na dane w bazie danych (wiersz 19.).

Listing 4: Klasa UserManager

```
1  @Stateless
2  public class UserManager {
3
4    @EJB
5    UserDAO userDAO;
6
7    public User registerUser(String login, String password) throws
         EntityAlreadyExist {
8      User user = null;
9      try {
10       user = userDAO.getByUsername(login);
11       throw new EntityAlreadyExist();
12     } catch (EntityNotFound e) {
```

```
13          System.out.println(e.getMessage());
14          user = new User();
15          user.setUsername(login);
16          user.setPassword(password);// TODO: HASH!
17          user = userDAO.create(user);
18        }
19        return user;
20      }
21
22      public User loginUser(String login, String password) throws
            WrongPassword, EntityNotFound {
23        User user = null;
24        user = userDAO.getByUsername(login);
25        if (password.equals(user.getPassword())) {// TODO: HASH!
26          user.setToken(login + "|" + password);
27        } else {
28          throw new WrongPassword();// wrong password
29        }
30        return user;
31      }
32
33  }
```

5.3.3. Interfejs wejściowy

Dostęp do logiki aplikacji został udostępniony poprzez interfejs REST stworzony przy użyciu standardu JAX-RS. W celu kontroli dostępu do aplikacji (ochrona przed niezalogowanymi użytkownikami) został utworzony filtr przechwytujący wchodzące zapytania (listing 5). Pobiera on zawartość nagłówka *AUTHORIZATON*, a następnie, dzięki wstrzykniętemu managerowi użytkowników, decyduje czy żądanie może zostać przepuszczone dalej. W nagłówku znajduje się login użytkownika oraz token wygenerowany w trakcie procesu logowania. W przypadku, gdy token będzie niepoprawny, żądanie zostanie przerwane z kodem odpowiedzi *UNAUTHORIZED*.

Listing 5: Klasa LoginFilter

```
1  @Secured
2  @Provider
3  @Priority(Priorities.AUTHENTICATION)
4  public class LoginFilter implements ContainerRequestFilter{
5
6
7    @Inject
8    Authenticator auth;
9
10   @Override
11   public void filter(ContainerRequestContext requestContext) throws
         IOException {
12     String token=requestContext.getHeaderString(HttpHeaders.
           AUTHORIZATION);
13     if(!auth.validateToken(token)){
14       requestContext.abortWith(Response.status(Status.UNAUTHORIZED).
             build());
15     }
16
17   }
18 }
```

Nie każda metoda jest chroniona przed dostępem niezalogowanych użytkowników. Metody chronione są oznaczone adnotacją, którą jest też oznaczony filtr (adnotacja @Secured przedstawiona na listingu 6).

Listing 6: Adnotacja @Secured

```
1  @Target({ ElementType.TYPE, ElementType.METHOD })
2  @Retention(value=RetentionPolicy.RUNTIME)
3  @NameBinding
4  public @interface Secured {
5
6  }
```

Na listingu 7 został przedstawiony fragment klasy będącej punktem wejścia do serwera odpowiedzialnym za zdjęcia. Za pomocą adnotacji oznaczono ścieżkę oraz typ danych, które przyjmują oraz zwracają metody danej klasy. Dodatkowo każda metoda jest opatrzona adnotacją z konkretną ścieżką oraz z metodą HTTP jaką obsługuje. Każda metoda zwraca obiekt typu Response, który reprezentuje odpowiedź HTTP. Poprzez ten obiekt ustawiany jest kod odpowiedzi oraz jej treść w formacie JSON.

Listing 7: Fragment klasy ImageRestEndpoint

```
1  @Path("image")
2  @Consumes({ "application/json" })
3  @Produces({ "application/json" })
4  public class ImageRestEndpoint {
5
6    @EJB
7    ImageManager imageManager;
8
9    @Path("/")
10   @GET
11   public Response getAllImageNames() {
12     List<String> names = imageManager.getAllImageNames();
13     return Response.ok(names).build();
14   }
15
16   @Path("/{id}")
17   @GET
18   @Produces(MediaType.APPLICATION_OCTET_STREAM)
19   @Secured
20   public Response getImage(@PathParam("id") String id) {
21     File image = null;
22     try {
23       image = imageManager.loadImage(id);
24       return Response.ok(image, MediaType.APPLICATION_OCTET_STREAM)
25           .header("Content-Disposition", "attachment; filename=\"" +
                 image.getName() + "\"").build();
26     } catch (IOException e) {
```

```
27          return Response.status(Status.NOT_FOUND).build();
28        }
29      }
30      ...
31  }
```

5.4. Klient

Aplikacja kliencka została wykonana...

5.4.1. Połączenie z serwerem

Komunikacja z serwerem w całości realizowana jest poprzez protokół HTTP.

Całość komunikacji zawarta została w klasie RestClient. Posiada ona metodę odpowiedzialną za tworzenie połączenia (listing 9) oraz metody realizujące konkretne zadania, takie jak utworzenie wycieczki, czy aktualizację informacji o użytkowniku na serwerze. Przebieg realizacji pojedynczego zadania po stronie klienta przedstawiono na rysunku 3. Większość metod może zakończyć się na 3 sposoby:

- operacja wykonana pomyślnie, zwrócona oczekiwana wartość,
- operacja się nie powiodła, zwrócona zostaje wartość `null` (w przypadku gdy wystąpi nieoczekiwany błąd po stronie serwera. Dzięki temu, gdy niepowodzenie nie wpływa na działanie aplikacji, użytkownik nie będzie o tym poinformowany) lub komunikat błędu,
- nastąpił błąd połączenia, zgłoszony zostaje wyjątek.

Do konwersji obiektów Javy na format JSON i odwrotnie użyto biblioteki Jackson. Udostępnia ona klasę ObjectMapper, która posiada metody umożliwiające taką konwersje. Przykład użycia przedstawiono na listingu 8.

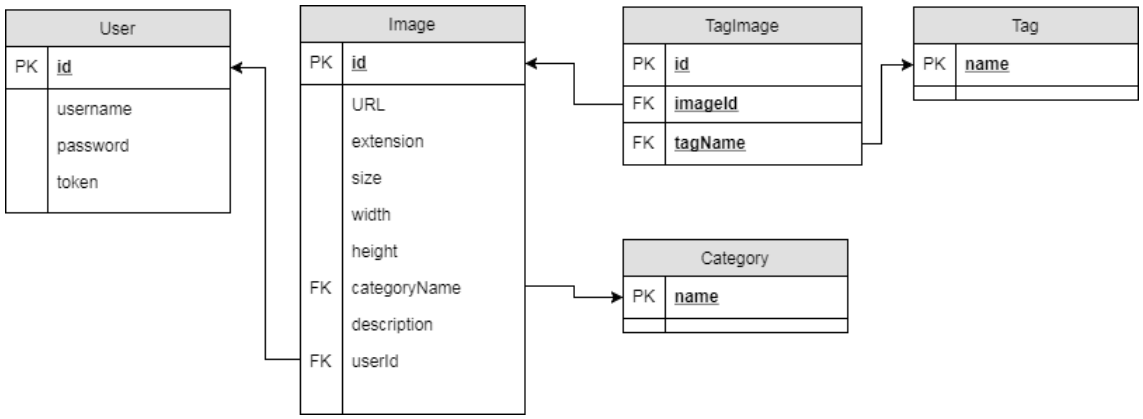
Listing 8: Przykład użycia obiektu ObjectMapper

```
1  @Entity
2  @NamedQueries({ @NamedQuery(name = "Image.findAll", query = "SELECT
       i FROM Image i"),
3      @NamedQuery(name = "Image.findBySize", query = "SELECT i FROM
           Image i WHERE i.size >= :minSize AND i.size <= :maxSize"),
4      @NamedQuery(name = "Image.findByExtension", query = "SELECT i
           FROM Image i WHERE i.extension = :extension"),
5      @NamedQuery(name = "Image.findByTag", query = "SELECT i FROM
           Image i, TagImage ti WHERE ti.tag = :tag AND ti.image = i"),
6      @NamedQuery(name = "Image.findByUser", query = "SELECT i FROM
           Image i WHERE i.user = :user"),
7      @NamedQuery(name = "Image.findByCategory", query = "SELECT i
           FROM Image i WHERE i.category = :category"),
8      @NamedQuery(name = "Image.findAllNames", query = "SELECT concat(
           i.url,'.',i.extension) FROM Image i") })
9  public class Image {
10   @Id
11   @GeneratedValue(strategy = GenerationType.IDENTITY)
12   private long id;
13   private String url;
14   private String extension;
15   private int size;
16   private int width;
17   private int height;
18   @ManyToOne
19   @JoinColumn(name = "categoryName")
20   private Category category;
21   private String description;
22   @ManyToOne
23   @JoinColumn(name = "userId")
24   private User user;
25   @OneToMany(mappedBy = "image")
26   private List<TagImage> tagImages;
27   ...
28 }
```

Listing 9: Metoda przygotowująca nagłówek zapytania HTTP

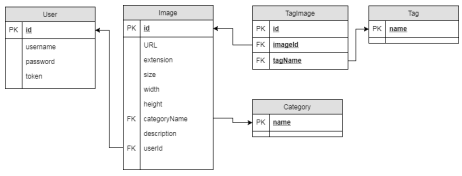
```
@Entity
@NamedQueries({ @NamedQuery(name = "Image.findAll", query = "SELECT
    i FROM Image i"),
    @NamedQuery(name = "Image.findBySize", query = "SELECT i FROM
        Image i WHERE i.size >= :minSize AND i.size <= :maxSize"),
    @NamedQuery(name = "Image.findByExtension", query = "SELECT i
        FROM Image i WHERE i.extension = :extension"),
    @NamedQuery(name = "Image.findByTag", query = "SELECT i FROM
        Image i, TagImage ti WHERE ti.tag = :tag AND ti.image = i"),
    @NamedQuery(name = "Image.findByUser", query = "SELECT i FROM
        Image i WHERE i.user = :user"),
    @NamedQuery(name = "Image.findByCategory", query = "SELECT i
        FROM Image i WHERE i.category = :category"),
    @NamedQuery(name = "Image.findAllNames", query = "SELECT concat(
        i.url,'.',i.extension) FROM Image i") })
public class Image {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;
  private String url;
  private String extension;
  private int size;
  private int width;
  private int height;
  @ManyToOne
  @JoinColumn(name = "categoryName")
  private Category category;
  private String description;
  @ManyToOne
  @JoinColumn(name = "userId")
  private User user;
  @OneToMany(mappedBy = "image")
  private List<TagImage> tagImages;
  ...
```

```
28 }
```



Rysunek 3: Schemat generowania zapytania po stronie klienta
Źródło: opracowanie własne

System Android zabrania wykonywania operacji korzystających z sieci w wątku głównym aplikacji. W związku z tym metody klasy RestClient są wywoływane z klas dziedziczących po klasie AsyncTask. Dzięki temu komunikacja z serwerem odbywa się w tle, nie blokując interfejsu graficznego. Dodatkowo została utworzona klasa WaitingAsyncTask dziedzicząca po klasie AsyncTask. Zawiera ona kod odpowiedzialny za wyświetlanie ikony ładowania informującą użytkownika, że właśnie przeprowadzana jest jakaś operacja. Klasa ta wykorzystywana jest, gdy aplikacja kliencka oczekuje na odpowiedź serwera. Efekt widoczny dla użytkownika pokazano na rysunku 4.



Rysunek 4: Ekranu oczekiwania na wynik operacji wykonującej się w tle
Źródło: opracowanie własne

5.4.2. Śledzenie pozycji

Aby móc śledzić aktualną pozycję użytkownika utworzono klasę GPSListener implementującą interfejs LocationListener. Jest ona rejestrowana jako obserwator zdarzeń dotyczących systemu GPS. Informacje o postępach użytkownika są przechowywane w klasie UserTripManager (której ważniejsze metody wypisano na listingu 11). Zawiera ona metody odpowiedzialne za aktualizowanie pozycji użytkownika, obliczanie przebytej drogi, prędkości średniej oraz chwilowej. W chwili otrzymania z systemu GPS nowej lokalizacji wywoływana jest metoda onLocationChange() (przedstawiona na listingu 10). Wywoływane są w niej metody odpowiedzialne za zaktualizowanie pozycji użytkownika na mapie oraz aktualizacje postępów. Dodatkowo sprawdzany jest czas od wysłania ostatniej aktualizacji do serwera. Jeśli upłynął ustalony okres, to aktualizacja zostanie wysłana.

Listing 10: Metoda onLocationChange klasy GPSListener

```
1  @Entity
2  @NamedQueries({ @NamedQuery(name = "Image.findAll", query = "SELECT
       i FROM Image i"),
3      @NamedQuery(name = "Image.findBySize", query = "SELECT i FROM
           Image i WHERE i.size >= :minSize AND i.size <= :maxSize"),
4      @NamedQuery(name = "Image.findByExtension", query = "SELECT i
           FROM Image i WHERE i.extension = :extension"),
5      @NamedQuery(name = "Image.findByTag", query = "SELECT i FROM
           Image i, TagImage ti WHERE ti.tag = :tag AND ti.image = i"),
6      @NamedQuery(name = "Image.findByUser", query = "SELECT i FROM
           Image i WHERE i.user = :user"),
7      @NamedQuery(name = "Image.findByCategory", query = "SELECT i
           FROM Image i WHERE i.category = :category"),
8      @NamedQuery(name = "Image.findAllNames", query = "SELECT concat(
           i.url,'.',i.extension) FROM Image i") })
9  public class Image {
10   @Id
11   @GeneratedValue(strategy = GenerationType.IDENTITY)
12   private long id;
13   private String url;
```

```
14    private String extension;
15    private int size;
16    private int width;
17    private int height;
18    @ManyToOne
19    @JoinColumn(name = "categoryName")
20    private Category category;
21    private String description;
22    @ManyToOne
23    @JoinColumn(name = "userId")
24    private User user;
25    @OneToMany(mappedBy = "image")
26    private List<TagImage> tagImages;
27    ...
28 }
```

Listing 11: Nagłówki ważniejszych metod klasy UserTripManager

```
1  @Entity
2  @NamedQueries({ @NamedQuery(name = "Image.findAll", query = "SELECT
       i FROM Image i"),
3      @NamedQuery(name = "Image.findBySize", query = "SELECT i FROM
          Image i WHERE i.size >= :minSize AND i.size <= :maxSize"),
4      @NamedQuery(name = "Image.findByExtension", query = "SELECT i
          FROM Image i WHERE i.extension = :extension"),
5      @NamedQuery(name = "Image.findByTag", query = "SELECT i FROM
          Image i, TagImage ti WHERE ti.tag = :tag AND ti.image = i"),
6      @NamedQuery(name = "Image.findByUser", query = "SELECT i FROM
          Image i WHERE i.user = :user"),
7      @NamedQuery(name = "Image.findByCategory", query = "SELECT i
          FROM Image i WHERE i.category = :category"),
8      @NamedQuery(name = "Image.findAllNames", query = "SELECT concat(
          i.url,'.',i.extension) FROM Image i") })
9  public class Image {
10   @Id
11   @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
12    private long id;
13    private String url;
14    private String extension;
15    private int size;
16    private int width;
17    private int height;
18    @ManyToOne
19    @JoinColumn(name = "categoryName")
20    private Category category;
21    private String description;
22    @ManyToOne
23    @JoinColumn(name = "userId")
24    private User user;
25    @OneToMany(mappedBy = "image")
26    private List<TagImage> tagImages;
27    ...
28 }
```

Metoda updateLocation() klasy UserTripManager odświeża aktualnie przechowywaną pozycje oraz przebyty dystans. Przewidywany pozostały czas zwracany jest w minutach, prędkości (chwilowa i średnia) w km/h, a dystans w kilometrach. Zarówno przebyty dystans, jak i prędkości obliczane są na podstawie pozycji uzyskanych z systemu GPS.

5.4.3. Interfejs graficzny

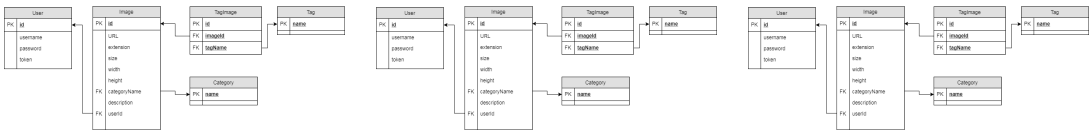
Interfejs graficzny stworzono wykorzystując powiązane ze sobą aktywności. Po uruchomieniu aplikacji, jeśli użytkownik nie był wcześniej zalogowany, ukaże się ekran logowania (rysunek 5a). Po zalogowaniu użytkownik uzyska dostęp do menu głównego (rysunek 5b). Pod przyciskiem "Opcje" dostępne są ustawienia dotyczące rodzaju map oraz częstotliwości komunikowania się z serwerem (rysunek 5c). Ekrany tworzenia wycieczki, dołączania do niej oraz podglądu trwającej aktualnie wycieczki wykorzystują zakładki w celu podziału widoku. Ekran tworzenia wycieczki podzielono na mapę na której można dodać do niej punkty kontrolne (rysunek 6a) oraz szczegóły, gdzie należy ustalić m.in.

nazwę i opis wycieczki (rysunek 6b). Dołączając do wycieczki trzeba wybrać ją z listy (rysunek 6c), a następnie można podejrzeć jej szczegóły oraz przebieg na mapie. Po dołączeniu do wycieczki, pod opcją "Aktualna wycieczka" będzie dostępna podobna mapa, na której dodatkowo zaznaczeni są uczestnicy wycieczki. Ponadto, znajdują się tam także zakładki z wynikami użytkownika, listą innych uczestników wraz z ich prędkością (rysunek 7b) oraz z rozmowami z innymi uczestnikami. Po wybraniu uczestnika na liście można spróbować wysłać do niego wiadomość (rysunek 7c). Zakładki można zmienić za pomocą gestów przesunięcia (w prawo lub lewo). W tym celu stworzono abstrakcyjną klasę SlideTabActivity. Zawiera ona kod inicjujący detektor gestów oraz metody odpowiedzialne za reakcje na wybrane gesty. Rozszerzając taką klasę należy przypisać odpowiednie wartości do pól tabNumber i currentTab. Dodatkowo należy też w klasach odpowiadających za same zakładki w metodzie onResume() ustawić pole currentTab. Jest to wymagane gdy do zmiany ekranów wykorzystuje się zarówno gesty, jak i zakładki.

Listing 12: Klasa SwipeTabActivity

```
1  @Entity
2  @NamedQueries({ @NamedQuery(name = "Image.findAll", query = "SELECT
       i FROM Image i"),
3      @NamedQuery(name = "Image.findBySize", query = "SELECT i FROM
           Image i WHERE i.size >= :minSize AND i.size <= :maxSize"),
4      @NamedQuery(name = "Image.findByExtension", query = "SELECT i
           FROM Image i WHERE i.extension = :extension"),
5      @NamedQuery(name = "Image.findByTag", query = "SELECT i FROM
           Image i, TagImage ti WHERE ti.tag = :tag AND ti.image = i"),
6      @NamedQuery(name = "Image.findByUser", query = "SELECT i FROM
           Image i WHERE i.user = :user"),
7      @NamedQuery(name = "Image.findByCategory", query = "SELECT i
           FROM Image i WHERE i.category = :category"),
8      @NamedQuery(name = "Image.findAllNames", query = "SELECT concat(
           i.url,'.',i.extension) FROM Image i") })
9  public class Image {
10   @Id
11   @GeneratedValue(strategy = GenerationType.IDENTITY)
```

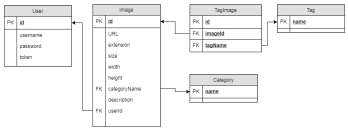
```
12    private long id;
13    private String url;
14    private String extension;
15    private int size;
16    private int width;
17    private int height;
18    @ManyToOne
19    @JoinColumn(name = "categoryName")
20    private Category category;
21    private String description;
22    @ManyToOne
23    @JoinColumn(name = "userId")
24    private User user;
25    @OneToMany(mappedBy = "image")
26    private List<TagImage> tagImages;
27    ...
28 }
```



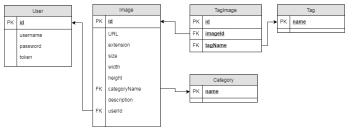
(a) Ekran logowania (b) Ekran menu (c) Ekran ustawień

Rysunek 5: Interfejs graficzny

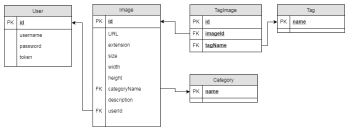
Źródło: opracowanie własne



(a) Tworzenie wycieczki - mapa



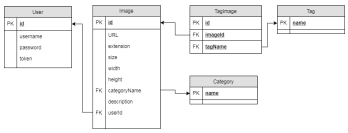
(b) Tworzenie wycieczki - szczegóły



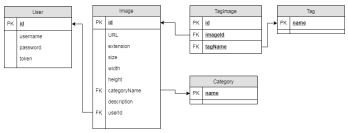
(c) Dołączanie do wycieczki - lista

Rysunek 6: Ekrany tworzenia oraz dołączania do wycieczki

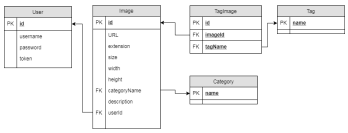
Źródło: opracowanie własne



(a) Nasze wyniki



(b) Uczestnicy wycieczki



(c) Ekran rozmowy z innym uczestnikiem

Rysunek 7: Interfejs graficzny

Źródło: opracowanie własne

6. Wnioski

Celem pracy było zaprojektowanie i stworzenie aplikacji służącej do wyszukiwania zdjęć. Stworzono aplikację o architekturze klient-serwer. Serwer powstał przy wykorzystaniu technologii Java Enterprise Edition.

Literatura

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (dostęp 10.01.2017)

[2] Leonard Richardson, Sam Ruby, *RESTful web services*, O'Reilly Media, Farnham 2007

[3] Eric Jendrock, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa, *Java EE 6 : przewodnik*, Wydawnictwo Helion, Gliwice 2012

[4] http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html (dostęp 02.02.2017)

[5] https://docs.jboss.org/author/display/WFLY10/Documentation (dostęp 20.02.2017)

[6] https://angular.io/guide/architecture

[7] https://dev.mysql.com/doc/refman/8.0/en/

[8] https://guides.github.com

[9] https://docs.travis-ci.com

[10] https://maven.apache.org