Złożenie pracy online:
2013-07-13 19:08:26
Kod pracy:
9940
Kod załącznika:
9941

Przemysław Michalik
(nr albumu: 18756*INF/INŻ)

Praca inżynierska

System składania zamówień online - serwer aplikacji mobilnej

Online ordering system - a mobile application server

Wydział: Nauk Społecznych i Informatyki
Kierunek: Informatyka
Specjalność: inżynieria oprogramowania
Promotor: dr Włodzimierz Moczurad

Nowy Sącz, 2013

SPIS TREŚCI

1. WSTĘP

Według powszechnie panujących stereotypów, ulubionym pożywieniem programistów jest okrągły placek zwany “pizza”. Biorąc pod uwagę specyfikę pracy w tej branży, wymagającą długotrwałego przebywania w skupieniu, łatwo odgadnąć genezę tego trendu. Pionierami w dowozie gotowych dań do klienta były właśnie firmy gastronomiczne specjalizujące się w pieczeniu pizzy. Programista będący w trakcie analizy nurtującego go problemu zdecydowanie chętniej skorzysta z usług takiej firmy, niż zajmie się przygotowaniem posiłku, narażając się na dekoncentrację. W firmie w której odbywałem staż, zwyczaj zamawiania pizzy był również bardzo mocno zakorzeniony. Po pewnym czasie zdałem sobie sprawę że mimo iż owy placek zamawiamy czasem nawet kilka razy w tygodniu, zawsze na to samo miejsce, za każdym razem należy podać komplet informacji takich jak adres, numer telefonu, a także ilości, rozmiary sosy, składniki itp. Często trzeba coś powtórzyć, bo Pani przyjmująca zamówienie nie dosłyszała, rozpraszając w ten sposób zamyślonych kolegów. Może się rownież zdarzyć że nie zostaniemy poinformowani o dodatkowych kosztach związanych z dowozem. Po pewnym czasie rutynowe składanie zamówień staję się coraz bardziej frustrujące i coraz trudniej o ochotnika do dokonania tej procedury wśród współpracowników.

Całe to zamieszanie stało się dla mnie i mojego kolegi siedzącego przy sąsiednim biurku inspiracją do napisania aplikacji która być może w przyszłości stanie się standardem i zrewolucjonizuje wymianę informacji pomiędzy restauracjami a klientami. Chodzi o wykorzystanie możliwości współczesnych urządzeń mobilnych. W zasadzie prawie każdy z nas w dzisiejszych czasach ma przy sobie urządzenie pełniące wciąż rolę telefonu, jednak znacznie bardziej uniwersalne niż telefony sprzed kilku lat i dysponujące mocą o jakiej nawet nie mogliśmy marzyć w domowych komputerach sprzed dekady.

Bardzo popularne stały się “markety” z których niemalże jednym kliknięciem możemy pobrać na nasz smartfon dowolną aplikację. Pozwala to na bardzo łatwe dotarcie do ogromnej grupy, nawet średniozaawansowanych technologicznie osób. Praktycznie każdy współczesny smartfon pozwala na dostęp do internetu, a także umożliwia określenie naszej pozycji na ziemi, na przykład za pomocą systemu GPS (*Global Positioning System*), czyli systemu nawigacji satelitarnej stworzonego przez Departament Obrony Stanów Zjednoczonych. W zasadzie technicznie to wszystko co było nam potrzebne - pozostało tylko napisanie odpowiedniego oprogramowania. W skrócie, cel był bardzo prosty: umożliwienie każdemu posiadaczowi smartfona wybranie dowolnych dań z menu

automatycznie zlokalizowanej najbliższej restauracji. Reszta miała dziać się bez ingerencji użytkownika, aż do momentu gdy dostawca zapuka do drzwi. Sposób w jaki udało nam się ten cel osiągnąć, w kontekście serwera aplikacji którego utworzenie spoczęło na moich barkach, zawarłem w kolejnych rozdziałach.

2. TECHNOLOGIA

2.1. Wprowadzenie

Aplikacja mobilna powstała w oparciu o platformę Android. Wybór technologii dla części serwerowej projektu padł na inny produkt firmy Google o nazwie **Google App Engine**. Powodem była deklarowana przez producenta dobra współpraca obydwu technologii, a także fakt, że wszystkie potrzebne narzędzia jak również późniejsze utrzymanie serwera nie wymagało poniesienia żadnych kosztów. W pojedynku języków programowania oferowanych przez GAE (Python, Java, GO) zwyciężyła **Java**, również dla utrzymania możliwie jak największej spójności z systemem Android. Pierwszym krokiem umożliwiającym rozpoczęcie pracy była instalacja i konfiguracja stosownego środowiska programistycznego.

2.2. Środowisko

Tu wybór padł na rekomendowane przez Google narzędzie: **Eclipse** w wersji **Indigo**.
Eclipse został napisany w Javie przez firmę IBM a następnie udostępniony na zasadach otwartego oprogramowania. Do samego uruchomienia go potrzebujemy wirtualnej maszyny Javy wchodzącej w skład pakietu JRE (Java Runtime Environment), lecz aby móc tworzyć własne aplikację należy zainstalować pakiet JDK (Java Development Kit), w którego skład wchodzi między innymi kompilator i debugger. Eclipse daję możliwość doinstalowania dowolnej liczby dedykowanych dodatków, których w sieci można znaleźć bardzo dużo, ze względu na popularność tego niezawodnego i stabilnego środowiska. W przypadku tworzenia aplikacji na platformą GAE, będzie to dodatek “Google Plugin for Eclipse”. Aby go zainstalować wystarczy wybrać z menu *Help* pozycję *Install New Software...,* a następnie w zależności od wersji środowiska (Indigo - 3.7) podać ścieżkę dostępu do katalogu z dodatkiem: https://dl.google.com/eclipse/plugin/3.7 .
Przez kolejne kroki prowadzi nas instalator.
Aby nasza aplikacja mogła być dostępna z każdego miejsca na ziemi, musimy ją opublikować w sieci. Do tego potrzebujemy własnego konta w usłudze.

2.3. Konto GAE

Aby móc korzystać z usługi wystarczy dysponować ogólnym kontem w usługach google (gmail, drive, docs, play). Otwieramy stronę https://appengine.google.com/ i podajemy swoje dane logowania.

Teraz wystarczy wybrać opcję *Create Application*, i podać jej nazwę oraz ścieżkę dostępu w subdomenie appspot.com (w przypadku InstantPizza jest to instantpizza1.appspot.com).

Panel administraycjny usług oferuje wiele możliwości. Opiszę kilka moim zdaniem najistotniejszych :

Dashboard

Tutaj znajdzie my podgląd statystyk odwiedzin naszej strony/aplikacji. Jest to dział szczególnie istotny ze względu na to, że pozwala nam monitorować jak szybko zbliżamy się do granicy wykorzystania miesięcznych zasobów w obrębie naszego abonamentu. Dla potrzeb niekomercyjnej aplikacji InstantPizza wystarczy darmowa opcja usługi. Jeżeli jednak aplikacja osiągnie popularność Google oferuje przejście na wyższą przepustowość bez konieczności przerwania dostępu do usługi. Jest to możliwe dzięki temu, że App Engine działa w tzw. piaskownicy, czyli **Sandbox**. “Piaskownica” pozwala na izolację bytu jakim jest aplikacja od fizycznego sprzętu. Jako użytkownik usługi nie jesteśmy w stanie określić z jakich zasobów korzysta nasza aplikacja, ale wykresy w panelu sterowania pozwalają nam kontrolować monitorować jej wydajność. Google oferuje też opcję automatycznego podniesienia parametrów usługi w przypadku wykrycia wzmożonej aktywności. Rozwiązanie to jest także bardzo bezpieczne - nawet jeżeli na skutek ataku, uszkodzona zostanie jedna z aplikacji, inne aplikację nie odczują skutków tego ataku ponieważ działają we własnych izolowanych środowiskach uruchomieniowych (*Runtime Environment*).

Logs

Na tej podstronie znajdziemy listę wszystkich komunikatów które zarchiwizowała nasza aplikacja. Jako że nie mamy dostępu do konsoli uruchomionej aplikacji, często jest to jedyna możliwość zlokalizowania przyczyn błędu spowodowanego w trakcie jej działania.

Versions

W tym dziale mamy możliwość zarządzaniem wersjami naszej aplikacji.

Wersje aplikacji uruchomione na serwerze

Jest to funkcjonalność szczególnie przydatna gdy wykryjemy błąd w naszej aplikacji i pracujemy nad jego rozwiązaniem, jednak chcemy aby stara wersja aplikacji wciąż działała w sieci. Jeżeli nowa poprawiona wersja zostanie ukończona, możemy ją aktywować jednym kliknięciem - zajmuje ona wówczas miejsce starej wersji, a proces przełączenia jest całkowicie transparentny dla użytkowników "frontend" i aplikacji klienckich.

Datastore Viewer

Dzięki tej funkcjonalności możemy łatwo sprawdzić czy informacje zapisywane przez naszą aplikację w bazie danych są poprawne. Datastore Viewer daję możliwość wyświetlenia wszystkich rekordów dla wybranej tabeli. Możemy także sami dodawać i usuwać rekordy bezpośrednio za pomocą panelu.

Application Settings

Tu możemy zmieniać główne opcje aplikacji takie jak : nazwa aplikacji, identyfikator w subdomenie, czas przechowywania logów, zewnętrzne domeny, a także możemy skopiować lub usunąć aplikację.

Permissions

Czasem jest potrzeba, aby dostęp do panelu miała więcej niż jedna osoba. W przypadku InstantPizza programista pracujący nad aplikacją mobilną, często potrzebował sprawdzić czy dodane za pomocą Web Service'u dane, zostały zapisane prawidłowo, ale nie potrzebował niczego zmieniać w działaniu samego serwera. App Engine umożliwia nadanie innym użytkowników uprawnień wyłącznie przeglądania zawartości:

Google Account	Role	Status	Remove Ac
awajdans@gmail.com	Viewer	Active	Remove
pdmichalik@gmail.com — you	Owner	Active	Remove

Lista użytkowników aplikacji w panelu appengine.google.com

2.4. Baza Danych

W przypadki InstantPizza do przechowywania danych zdecydowałem się wykorzystać App Engine Datastore. Nie jest to tradycyjna SQL'owa baza danych. Datastore jest zaprojektowana z naciskiem na maksymalną skalowalność, tak by w razie gdy zajdzie taka potrzeba poradzić sobie z bardzo dużą ilością danych. Aby zapewnić maksymalną wydajność zrezygnowano między innymi z :

- operacji typu Join
- filtrowania nierównościami wielu właściwości
- filtrowania danych opartego na wyniku podwykonania

Datastore jest obiektową bazą danych, która dysponuje własnym niskopoziomowym interfejsem, jednak znacznie wygodniej jest korzystać z dostępnych technologii Java bibliotek pośredniczących JDO (*Java Data Objects*) oraz JPA (*Java Persistence API*).

JDO

Aby aplikacja mogła korzystać z JDO 3.0 należy zainstalować dodatek DataNucleus v2 - w tym celu wystarczy dołączyć do projektu odpowiednie biblioteki w formacie *.*jar* oraz plik konfiguracyjny *jdoconfig.xml*.

Niestety niektóre funkcjonalności JDO nie będa współpracować z App Engine Datastore ze względu na jego specyfikę:

- relacje typu *owned many-to-many*
- zapytania z operatorem Join
- grupowanie i agregacja JDOQL
- polimorficzne zapytania (brak możliwości pobrania instancji podklasy za pomocą zapytania klasy nadrzędnej)

JDO od wersji 3.0 umożliwia definiowanie klas danych (w SQL odpowiednikiem jest tabela), za pomocą adnotacji (Java annotations), które do języka Java zostały wprowadzone w wersji 1.5.

W poprzednich wersjach klasy były definiowane za pomocą tagów XML.

W bazie można też przechowywać beż żadnych dodatkowych konfiguracji obiekty POJO (*Plain Old Java Objects*), czyli obiekty podstawowej palety typów Java, najczęściej jednak w roli zawartości pól obiektów zdefiniowanych przez programistę.

Przykładowa definicja klasy (w InstantPizza służy ona do przechowywania jednostek miar):

```
@PersistenceCapable
public class Unit implements Serializable{

    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    @JsonIgnore
    private Key id;

    @Persistent
    private String name;
    @Persistent
    private String signature;

    // …
}
```

W tej nieskomplikowanej klasie mamy dwa pola podstawowego typu *String* a także pole id typu *Key*, z adnotacja *@PrimaryKey* która oznacza że to pole jest polem klucza głównego.
Parametr "*valueStrategy = IdGeneratorStrategy.IDENTITY*", adnotacji *@Persistent* określa natomiast sposób generowania klucza - w tym wypadku klucz jest generowany automatycznie.

Każda encja (entity), podobnie jak w SQL, musi mieć pole (kolumnę) z unikatowym kluczem. W Datastore kluczem może w być String, liczba, a nawet adres email, jednak udało mi się doświadczalnie zaobserwować, że to rozwiązanie nie radzi sobie w przypadku gdy tak zdefiniowany chcemy przechowywać w bazie jako składnik innego nadrzędnego obiektu.
W takiej sytuacji zmuszeni jesteśmy do zastosowania klucza typu *Key* z adnotacją jak w powyższym przykładzie. Powodem tego jest specyfika generowania kluczy w Datastore - klucze obiektów podrzędnych generowane są na podstawie kluczy obiektów nadrzędnych. Dodatkowym ograniczeniem jakie narzuca to rozwiązanie, jest konieczność tworzenia i zapisywania w bazie

obiektu nadrzędnego i podrzędnego jednocześnie. Jeżeli chcemy mieć możliwość edycji jakiegoś składnika w dowolnym momencie pracy programu, musimy relacje skonfigurować ręcznie. Oto w fragmenty kodu dla porównania obydwu rozwiązań:

Bezpośrednie przechowywanie referencji

```
@PersistenceCapable
public class OrderItem implements Serializable{
      @PrimaryKey
      @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
      @JsonIgnore
      private Key id;
      //...
      @Persistent
      @JsonIgnore
      private Order order;
      //...
}
```

- Pozycja zamówienia zapisywana jest do bazy w każdym przypadku razem z zamówieniem, dlatego możliwe jest zastosowanie referencji do obiektu.

Przechowywanie klucza obiektu

```
@PersistenceCapable
public class Restaurant implements Serializable{
      @PrimaryKey
      @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
      @Extension(vendorName = "datanucleus", key="gae.encoded-pk",
value="true")
       private String id;
      //...
      @Persistent
      @JsonProperty("menuId")
      private Key menuId;
      //...
}
```

-Restauracja może z upływem czasu zmienić swoje menu, dlatego przechowujemy tylko klucz obiektu typu *Menu*. Rozwiązanie to nie pozwala na proste pobieranie obiektów z klasy nadrzednej za pomocą notacji kropkowej. Aby otrzymać żądany rezultat, musimy odnaleźć obiekt o określonym kluczu, wśród wszystkich obiektów danego typu:

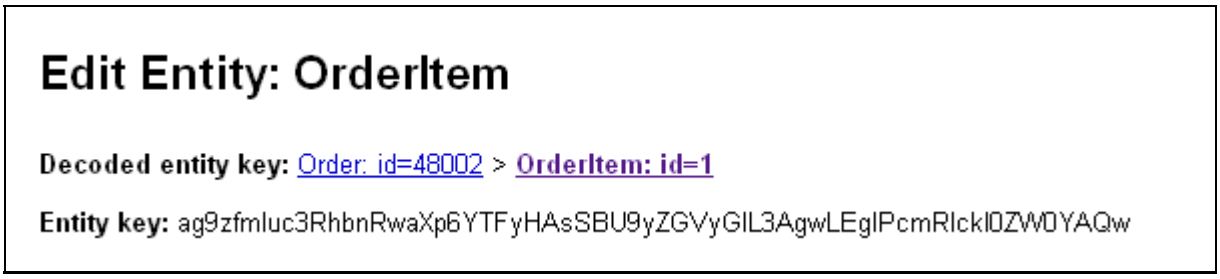
```
public static Menu getMenuById (String id) {
      Menu out = null;
      PersistenceManager pm = PMF.get().getPersistenceManager();
      try {
            out = pm.getObjectById(Menu.class, id);
            out.getSubMenusKeys();
      } catch (Exception e) {
            loger.info(e.getMessage());
      } finally {
            pm.close();
      }
      return out;
}
```

Powyższy kod to przykład pobrania z bazy obiektu typu Menu (*Menu.class* zwraca identyfikator typu), na podstawie klucza - *id*. Do tego celu wykorzystany został obiekt klasy *PersistanceManager*, z dodatku DataNucleus v2, dysponujący szablonową funkcją *getObjectById*.

Automatycznie generowany klucz składa się z:

- typu encji
- identyfikatora (String lub Integer)
- ścieżki przodków (jeżeli obiekt jest podrzędny względem innego)

Poniżej przykład klucza zakodowanego w postaci String'a, oraz zdekodowana postać:



Jak widać, klucz pozycji zamówienia zawiera także klucz swojego rodzica, czyli zamówienia.

W ten sposób tworzone są grupy encji. Datastore w wycofanej już wersji "master/slave" umożliwiał transakcyjny zapis do bazy tylko w obrębie takich grup. W nowej wersji HRD (*High Replication Datastore*), jest wprowadzono taką możliwość, ale trzeba jawnie zadeklarować że chcemy ją wykorzystać kosztem wydajności:

```
DatastoreService datastore =
DatastoreServiceFactory.getDatastoreService();
TransactionOptions options =
TransactionOptions.Builder.withXG(true);
Transaction txn = datastore.beginTransaction(options);
      try {
      MenuItemsDAO.save(mi);
      MenusDAO.save(sm);
      txn.commit();
      //...
} catch (Exception e) {
      //...
} finally{
      if (txn.isActive()) {
            txn.rollback();
      }
}
```

Ponieważ pozycje menu w obrębie danego menu mogą się zmieniać w czasie, powiązania przechowywane są w bazie w postaci kluczy. Wymusza to użycie XG(cross-group transaction), a Datastore musi pracować w trybie HRD.

Dziedziczenie

JDO umożliwia zastosowanie mechanizmu dziedziczenia w definiowaniu klas danych. W przypadku InstantPizza użycietego mechanizmu znalazło zastocowanie w przypadku bytów *Supplier*, *CMSUser* i *Client* bazujących na klasie *User*:

```
@PersistenceCapable
@Inheritance(strategy = InheritanceStrategy.SUBCLASS_TABLE)
public abstract class User implements Serializable{
	@PrimaryKey
	private String email;

	@Persistent
	private String name;

	@Persistent
	private String phoneNumber;

	@Persistent
	private boolean isBanned;

	@Persistent
	private boolean isActive;
	//...
}
```

Ponieważ każdy użytkownik chcący korzystać z aplikacji musi mieć konto google i legitymować się adresem email, każdy typ użytkownika zawiera to pole. name, phoneNumber, isBanned, isActive to również pola wykorzystywane w przypadku wszystkich użytkowników InstantPizza. Jako że klasa jest abstrakcyjna, nie można zapisywać w bazie obiektów typu User, ale dostajemy możliwość bazowania na niej a także korzystania z mechanizmu polimorfizmu(na przykład część funkcji zarządzających użytkownikami może być wspólna):

```
void ban(User u){
	//...
}
```

Jeżeli blokowanie użytkownika sprowadza się do ustawienia flagi(true/false) w odpowiednim polu bazy danych, proces blokowania można ujednolicić. Bez względu na to jaki rodzaj użytkownika otrzymamy na wejściu funkcji, zablokowanie sprowadza się do przypisania polu *isBanned* wartości *true*.

JDO wyróżnia cztery strategie dziedziczenia:

- **subclass-table** - pola znajdują są w tabeli(encji) podklasy
- **new-table** - pola znajdują się w osobnej tabeli
- **superclass-table** - pola znajdują się w tabeli nadklasy
- **complete-table** - pola klasy i wszystkich jej nadklas znajdują się w osobnych tabelach

Przykładowa klasa dziedzicząca:

```
@PersistenceCapable
public class Supplier extends User{
	@Persistent
	private List<Key> orderKeys;

	@Persistent
	private double lastLongitude;

	@Persistent
	private double lastLatitude;

	//...
}
```

Wystarczy standardowo za pomocą słowa kluczowego *extends* określić klasę, której pola i funkcje odziedziczy nowa klasa.

2.5. MVC

Aplikacja InstantPizza od strony użytkownika panelu, zorganizowana jest oparciu o wzorzec projektowy Model-View-Controller. Wzorzec ten zakłada podział aplikacji na trzy części:

Model

Ta część odpowiada za logikę biznesową aplikacji, czyli sposób w jaki aplikacja przechowuje dane. W poprzednim rozdziale opisałem jakie mechanizmy użyte zostały w InstantPizza dla realizacji logiki biznesowej i zapewnienia persystencji danych. Model powinien być jedyną częścią aplikacji, która w trwały sposób przechowuje dane, aby założenia MVC były spełnione. Model powinien ukrywać przed resztą aplikacji szczegóły implementacyjne takie jak zapytania bazodanowe.

W InstantPizza do definiowania konkretnych struktur danych zastosowałem klasy danych JDO, natomiast interfejsy zarządzania tymi danymi zrealizowane są wyłącznie w klasach DAO (Data Access Object), pośredniczących między logiką biznesową a warstwą prezentacji i pozostałymi klasami aplikacji. Przykładowa klasa DAO może mieć następującą strukturę:

```
public class UnitsDAO {
      public static Unit getUnitById(Key id){
            //...
      }
      public static List<Unit> getUnits() {
            //...
      }
      public static void update(Unit object) {
          //...
      }
      public static void save(Unit object) {
          //...
      }
      public static void delete(Unit object) {
            //...
      }
}
```

Implementacją poszczególnych funkcji uzależniona jest od wykorzystywanej technologii.

W przypadku InstantPizza realizacja funkcji oparta jest o klasę:

```
public final class PMF {
    private static final PersistenceManagerFactory pmfInstance =

JDOHelper.getPersistenceManagerFactory("transactions-optional");
    private PMF() {}
    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }
}
```

Klasa PMF za pośrednictwem metody get() dostarcza referencji do wspólnej dla całej aplikacji instancji klasy *PersistenceManagerFactory.*

Chcąc dokonywać operacji na bazie danych musimy utworzyć obiekt *PersistanceManager*:

PersistenceManager pm = PMF.get().getPersistenceManager();

Aby pobrać z bazy tylko jeden obiekt możemy użyć funkcji:

pm.getObjectById(Unit.class, id);

Aby pobrać wszystkie obiekty danego typu z bazy możemy np. skonstruować zapytanie:

Query q = pm.newQuery(Unit.class);

q.setOrdering("name asc");

q.addExtension("datanucleus.appengine.datastoreReadConsistency", "STRONG");

"datastoreReadConsistency" ustawione na "*STRONG"* gwarantuje że pobierzemy najaktualniejsze dane z bazy. Stworzone zapytanie urochamiamy funkcją: *q.execute();*

Operacje edycji obiektu wykonujemy za pośrednictwem transakcji:

```
try {
      pm.currentTransaction().begin();
      Unit s = pm.getObjectById(Unit.class, object.getId());
            s.setName(object.getName());
```

```
            s.setSignature(object.getSignature());
        pm.currentTransaction().commit();
} catch (Exception e) {
        //...
        pm.currentTransaction().rollback();
} finally {
        pm.close();
}
```

Konstrukcja taka gwarantuje nam że system albo utrwali wszystkie zmiany, albo wszystkie odrzuci, jeżeli natrafi na jakieś problemy w trakcie wprowadzania modyfikacji.
Za każdym razem gdy korzystamy z instancji *PersistenManager'a,* po przeprowadzeniu wszystkich operacji, bez względu na rezultat ich wykonania, należy wywołać metodę *close().* Po jej wykonaniu na obiekcie nie można wykonywać już operacji, co z punktu widzenia bezpieczeństwa i wydajności jest efektem pożądanym.

Aby zapisać nowo utworzony obiekt do bazy, podajemy go jako argument funkcji:
makePersistent(object); (bez konieczności podawania identyfikatora typu jako parametru)
Aby usunąć obiekt z bazy:
deletePersistent(pm.getObjectById(Unit.class, object.getId()));

View

Widok, to część aplikacji, odpowiedzialna za warstwę prezentacji oraz interakcje z użytkownikiem. Dostarcza graficznych interfejsów służących do wprowadzania danych, nawigacji w obrębie panelu, a także wyświetla zgromadzone informacje w sposób czytelny i zrozumiały użytkownikowi. Widok może pobierać określone dane bezpośrednio z modelu, ale wszelkie modyfikacje danych a także określenie kryteriów ich doboru leżą po stronie **kontrolera**.
W InstantPizza technologią odpowiadającą za prezentacje jest język HTML wspierany przez JSP (*Java Server Pages*), który odpowiada za dynamiczne fragmenty treści.

JSP oferuje kilka rodzajów tagów dzięki którym możemy wstawiać kod JAVA, pomiędzy statyczne fragmenty kodu HTML. W chwili gdy przeglądarka internetowa wyśle do serwera żądanie dotyczące treści podstrony JSP, serwer wykonuje kod JAVA, uzupełnia ewentualne dane i przesyła gotową treść w postaci czystego kodu HTML (ewentualnie wzbogaconego o skrypty wykonywane po stronie przeglądarki takie jak JavaScript), lub zwraca kod błędu w przypadku niepowodzenia.

Tagi podstawowe:

<%@ %> - służy do importowania klas z których chcemy korzystać w obrębie dokumentu:

<%@ page import="java.util." %>* , a także określenia parametrów takich jak:

<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<% %> - tagi w których umieścić możemy dowolny kod JAVA:

<h3>Lista podmenu dla menu: <% out.print(menu.getName()) %></h3>

Powyższa konstrukcja może być też zastąpiona specjalnym tagiem odpowiedzialnym za wyświetlanie:

<h3>Lista podmenu dla menu: <%= menu.getName() %></h3>

Przykład zastosowania technologii JSP do wygenerowania wierszy tabeli z rekordami wszystkich dostawców (*Suppliers*) z bazy danych:

```
<%
      List<Supplier> suppliers = SuppliersDAO.getSuppliers();
      for (Supplier s : suppliers) {
%>
```

Powyższy fragment kodu zapisuje w zmiennej *suppliers* listę wszystkich pracowników.

Następnie otwarta zostaje pętla typu “foreach” czyli wykonująca się dla każdego elementu listy, udostępniając poprzez zmienna *s* referencję do bieżącego.

Teraz należy odpowiednio dobrać i sformatować dane wyjściowe w postaci wierszy tabeli:

```
<tr class="tableRow">
      <td>
            <%= s.getName() %> <br/>
            <!-- ... -->
      </td>
      <td>
            <%= s.getEmail() %>
      </td>
      <td><!-- ... --></td>
</tr>
<%  } %>
```

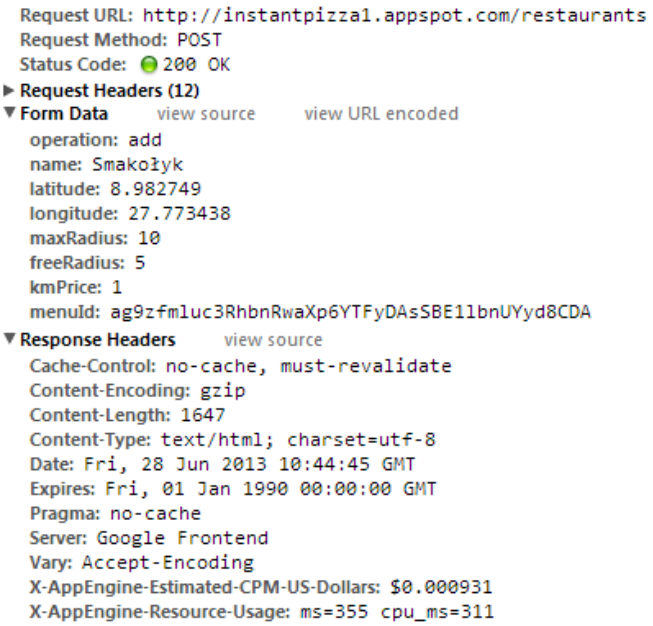
Należy pamiętać o zamykającym nawiasie na końcu, ponieważ literówki i niedomknięte nawiasy są często przyczyną trudnych do wykrycia błędów - pliki *.jsp, w przeciwieństwie do serwletów, nie są kompilowane przed umieszczeniem ich na serwerze.

Powyższy kod odpowiada za wyświetlanie danych. Teraz pora na edycję danych.

Do tego posłużymy się formularzami HTML:

```
<form action="/restaurants" method="post">
<input type="hidden" name="operation" value="add">
//...
<input type="text" size="25" name="name">
//...
<input type="text" size="25" name="latitude">
//...
<input type="text" size="25" name="longitude">
//...
<input type="text" size="25" name="maxRadius">
//...
<input type="text" size="25" name="freeRadius">
//...
<input type="text" size="25" name="kmPrice">
//...
<select name="menuId">
<option value="">Wybierz menu z listy</option>
<%
List<Menu> menus = MenusDAO.getMenus();
for (Menu menu : menus) {
%>
<option value="<%=KeyFactory.keyToString(menu.getId())%>">
<%=menu.getName() %>
</option>
<% } %>
</select>
//...
<input type="submit" value="DODAJ" />
//...
</form>
```

Większość pól to standardowe pola tekstowe typu *input*. Nieco ciekawszym elementem tego formularza jest pole typu *select,* którego wartości ustawiane są dynamicznie, w zależności od obiektów Menu które znajdują się w bazie w chwili odczytu. Zasada jest podobna jak w przypadku listy dostawców przedstawionej we wcześniejszym przykładzie, jednak tu musimy też zapewnić możliwość poźniejszej identyfikacji w serwlecie wybranego Menu. W tym celu w parametrze *value* podajemy Id, czyli klucz menu. Formularze mają to ograniczenie, że fizycznie mogą przekazywać tylko ciągi znaków i aby przesłać jakiś abstrakcyjny byt, musimy go najpierw sprowadzić do postaci tekstu, na bazie którego odbierający go serwlet będzie mógł przywrócić obiekt do pierwotnej postaci. Służy nam do tego obiekt *KeyFactory*, dysponujący statyczną metodą *keyToString* a także umożliwiającą odtworzenie obiektu metodą *stringToKey*.



Dane przesłane za pomocą formularza

Powyżej dane wysłanego formularza przechwycone za pomocą narzędzi programistycznych przeglądarki Google Chrome.

Request URL to adres na jaki zostało wysłane żądanie z formularza, określony w polu *action.*

Request Method to metoda przesyłania danych w obrębie protokołu HTTP. W tym wypadku użyta jest metoda POST, która w przeciwnieństwie do metody GET nie dołącza danych do adresu URL żądania, dzięki czemu czemu umożliwia przesłanie znacznie większej ilości danych i nie zostawia w historii przeglądarki śladów swojego wykonania.

Status Code to rezultat przesłania formularza - kod 200 według definicji statusów HTTP oznacza że operacja zakończyła się sukcesem(od strony wykonania żądania, co jeszcze nie jest gwarantem

osiągnięcia celów biznesowych). Pole *operation* w mojej realizacji MVC identyfikuje akcje danego kontrolera. Przesłany formularz z powyższego przykładu trafia do odpowiedniego kontrolera.

Kontroler (sterownik)

Kontroler jest odpowiedzialny ze wykonywanie zleceń z widoku lub z zewnątrz aplikacji. Wywołuje on metody wprowadzające zmiany w bazie danych, a także decyduje o tym jaki widok wyświetlić po wykonaniu operacji. W przypadki InstantPizza za zadania te odpowiedzialne są klasy typu *Servlet,* zwane dalej serwletami.

Serwlety rozszerzają możliwości serwera, w modelu żądanie-odpowiedź. Odbierają i analizują żądania wysyłane głównie za pomocą formularzy. Cykl życia serwletu składa się z trzech podstawowych etapów:

1. utworzenie i inicjalizacja - na tym etapie tworzona jest instancja serwletu a następnie wywoływana metoda *init()* - wykonywana jest ona tylko raz w całym cyklu życia serwletu:
    a. jeżeli użyjemy tagu *load-on-startup* w pliku konfiguracyjnym (opiszę go nieco dalej), wywołana zostanie zaraz po starcie serwera
    b. zaraz przed pierwszym wywołaniem metody *service()* odpowiedzialnej za obsługę żądania
    c. jeśli administrator zdecyduje się zainicjować serwlet bezpośrednio
2. nasłuchiwanie - za każdym razem gdy na adres serwletu przekazywane jest żądanie, uruchamiana jest metoda service(), która następnie przepina żądanie do funkcji obsługujących konkretnie metody HTML
    a. *doGet()*
    b. *doPost()*
    c. *doXx()* - inne metody HTML, z zachowaniem konwencji nazewnictwa
3. likwidacja - przed zakończeniem pracy serwletu wywoływana jest funkcja *destroy(),* która służy zwolnieniu zasobów wykorzystywanych przez serwlet, które nie będą już potrzebne po zakończeniu działania jego instancji, lub nie mogły by być zwolnione przez Garbage Collector(mechanizm automatycznego zarządzania pamięcią, wchodzący w skład środowiska uruchomieniowego JAVA).

Nazwę i lokalizację definicji klasy każdego serwletu należy podać w osobnym węźle pliku *web.xml.* Aby można było odwoływać się do serwletu poprzez adres URL, należy także utworzyć węzeł z zapisem mapowania na niego konkretnego adresu:

```
<servlet>
      <servlet-name>RestaurantsServlet</servlet-name>
      <servlet-class>pl.instantpizza.srv.servlets.RestaurantsServlet
      </servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>RestaurantsServlet</servlet-name>
      <url-pattern>/restaurants</url-pattern>
</servlet-mapping>
```

Powyższy zapis oznacza że żądania kierowane na adres URL /restaurants (instantpizza1.appspot.com/restaurants), przekierowane zostaną do instancji klasy *RestaurantsServlet.*

Przykładowo, dane z formularza zaprezentowanego w poprzednim rozdziale, natrafią na następującą akcję w kontrolerze *RestaurantsServlet,* w metodzie *doPost():*

```
if (operation.compareTo("add") == 0) {
      String name = req.getParameter("name");
      if (name != null) {
            Restaurant r = new Restaurant();
            r.setName(name);
            String longitude = req.getParameter("longitude");
            String latitude = req.getParameter("latitude");
            //...
            if (longitude != null && latitude != null /* ... */) {
                  try {
                        r.setLongitude(new Double(longitude));
                        r.setLatitude(new Double(latitude));
                        //...
                        RestaurantsDAO.save(r);
                        req.setAttribute("info","Restauracja " +
r.getName() + " została dodana.");
                  } catch (Exception e) {
                        loger.log(Level.WARNING , e.getMessage());
                        req.setAttribute("info","Wystąpił problem przy
```

```
dodawaniu restauracji "
                        + r.getName());
                }
            }
      }
}
```

Zmienne *req* oraz *resp* to parametry funkcji *doPost:*
public void doPost(HttpServletRequest req, HttpServletResponse resp){...}

HttpServletRequest to żądanie skierowane do serwletu, natomiast *HttpServletResponse* reprezentuje klasę obiektów odpowiedzialnych za formułowanie odpowiedzi.

HttpServletRequest udostępnia między innymi metodę *getParameter(String)* która zwraca wartość pola o nazwie podanej jako parametr:
req.getParameter("name");

HttpServletResponse możemy użyć na przykład do przesłania "zleceniodawcy" informacji o błędzie podczas przetwarzania żądania:

```
public void doPost(HttpServletRequest req, HttpServletResponse
resp) throws IOException {
      //...
      String operation = req.getParameter("operation");
      if (operation == null){
            resp.sendError(400);
            return;
      }
      //...
}
```

W sytuacji gdy do serwletu trafi zestaw danych, w którym brakuje informacji o tym jakiej akcji kontrolera dotyczy, przetwarzanie żądania zostanie zatrzymane, a serwer odeśle informacje zwrotną ze statusem 400, czyli kodem błędu informującym o nieprawidłowym żądaniu.

HttpServletRequest możemy użyć do przekazywania do widoków (JSP) komunikatów na poziomie logiki aplikacji:

req.setAttribute("info","Restauracja " + r.getName() + " została usunięta.");

Za pomocą metody *setAttribute* ustalamy dodajemy kolejne parametry, jako atrybuty podając kolejno nazwę parametru i jego wartość.

Informacja przekazywana jest po na końcu metody *doPost,* po wykonaniu wszystkich innych operacji:

```
RequestDispatcher reqDispatcher =
getServletConfig().getServletContext().getRequestDispatcher("/jsp/r
estaurantsList.jsp");
try {
     reqDispatcher.forward(req,resp);
} catch (ServletException e) {
     e.printStackTrace();
}
```

Ewentualne komunikaty wyświetlane są na widoku (restaurantsList.jsp) dzięki wsparciu JavaScript i biblioteki jQuery:

```
$(document).ready(function(){
   if("${info}") {
     alert("${info}");
}});
```

Ten krótki fragment kodu w zdarzeniu *ready* obiektu *document* (a więc w momencie kompletnego załadowania podstrony) sprawdza czy istnieje parametr o nazwie *info*. Jeżeli tak, wyświetla jego treść jako komunikat przeglądarki internetowej.

Oprócz przesyłania komunikatów użytkownikowi, ewentualne informacje o błędach warto także archiwizować, dla ułatwienia administracji aplikacją i lokalizowania błędów.

Służy do tego system **logów**. Są ta specjalne rekordy, przechowujące rodzaj błędu, jego treść, a często także stos wywołań. Poniżej przykładowy wpis z zakładki **Logs** w panelu appengine.com.

```
2013-05-20 04:22:09.981  /panelUsers  404  214ms  0kb  Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.17 (KHTML
178.37.77.107 - pdmichalik [20/May/2013:04:22:09 -0700] "POST /panelUsers HTTP/1.1"
"http://instantpizza1.appspot.com/jsp/panelUsersList.jsp" "Mozilla/5.0 (X11; Linux i
Chrome/24.0.1312.56 Safari/537.17" "instantpizza1.appspot.com" ms=215 cpu_ms=42 cpm_
instance=00c61b117cc82acc001c2b9222d7533679f86dc4

W 2013-05-20 04:22:09.935
  /panelUsers
  java.lang.NullPointerException
          at com.google.appengine.api.datastore.KeyFactory.stringToKey(KeyFactory.ja
          at pl.instantpizza.srv.servlets.PanelUsersServlet.doPost(PanelUsersServlet
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:637)
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
          at org.mortbay.jetty.servlet.ServletHolder.handle(ServletHolder.java:511)
```

Fragment treści błędu zarchiwizowanego na serwerze (log)

Ten log został dodany automatycznie, jednak możemy także dodawać własne.

Potrzebujemy do tego instancji klasy *java.util.logging.Logger:*

private static final Logger loger = Logger.getLogger(PanelUsersServlet.class.getName());

W przyjętej przeze mnie konwencji nazwą logera jest nazwa klasy którą obsługuje:

PanelUsersServlet.class.getName()

Podczas edycji danych użytkownika, żaden błąd nie może zostać niezauważony:

```
try {
      CMSUsersDAO.update(u);
      req.setAttribute("info","Użytkownik " + u.getName() + " został
zablokowany.");
} catch (Exception e) {
      loger.log(Level.WARNING , e.getMessage());
      req.setAttribute("info","Wystąpił problem przy próbie
zablokowania użytkownika"
+ u.getName());
}
```

Pogrubiona linijka to fragment kodu który w przypadku złapania wyjątku przez blok *catch,* zapisze w logach aplikacji treść tego wyjątku. Klasa *Level* udostępnia następujące poziomy określające jak ważną informacje przechowuje dany wpis(w kolejności malejącej):

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

2.6. Komunikacja z urządzeniami mobilnymi - Restlet

Podstawowa funkcjonalność InstantPizza, czyli składanie zamówień za pomocą urządzeń mobilnych, wymaga określenia spójnego protokołu komunikacyjnego.
Do zapewnienia uniwersalnego interfejsu wymiany zastosowałem **Restlet -** framework dla języka Java implementujący protokół **REST** (representational state transfer).
Twórcą REST'a jest Roy Fielding, jeden z głównych współtwórców protokołu HTTP, na którym właśnie REST się opiera (wykorzystuje metody GET, POST, PUT oraz DELETE). Konkretne zasoby, dla których interfejsy chcemy udostępniać, identyfikowane są przez adresu URL, a dane mogą być zakodowane w dowolnym formacie(np. XML, JSON). W InstantPizza RESTful Web Service dostępny jest pod adresem URL */ws* (*instantpizza1.appspot.com/ws*).
Wpis w pliku *web.xml*:

```
<servlet>
        <servlet-name>RestletServlet</servlet-name>
        <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-class>
        <init-param>
                <param-name>org.restlet.application</param-name>
                <param-value>pl.instantpizza.srv.ws.WebServiceApplication</param-value>
```

```
        </init-param>
</servlet>
<servlet-mapping>
        <servlet-name>RestletServlet</servlet-name>
        <url-pattern>/ws/*</url-pattern>
</servlet-mapping>
```

Zapis */ws/** oznacza że wszelkie odwołania pod adresy podrzędne względem */ws* zostaną za pośrednictwem serwletu *org.restlet.ext.servlet.ServerServlet* przesłane do instancji klasy *pl.instantpizza.srv.ws.WebServiceApplication.* Klasa ta zajmuje się przede wszystkim routingiem, czyli mapowaniem adresów URL na odpowiednie klasy zasobów (*ServerResource)*:

```
public Restlet createInboundRoot() {
    Router mainRouter = new Router(getContext());
    mainRouter.attach("/restaurant/{restaurantId}",
RestaurantResource.class);
    mainRouter.attach("/restaurants", RestaurantsResource.class);
    mainRouter.attach("/menu/{menuId}", MenuResource.class);
    mainRouter.attach("/menus", MenusResource.class);
    mainRouter.attach("/subMenu/{subMenuId}",
SubMenuResource.class);
    mainRouter.attach("/subMenus", SubMenusResource.class);
    mainRouter.attach("/menuItem/{menuItemId}",
MenuItemResource.class);
    //...
    return mainRouter;
}
```

Dodatkowo zapis z użyciem nawiasów klamrowych pozwala na przekazywanie parametrów w ścieżce URL. Najczęściej wykorzystuje się tę funkcjonalność przy pobieraniu pojedynczych obiektów o zadanym kluczu, lub kolekcji obiektów spełniających jakieś kryterium:

```
mainRouter.attach("/menu/{menuId}", MenuResource.class);
```

Powyższa linijka to reguła routera przekierowująca do obiektu klasy *MenuResource*, umożliwiającego pobranie z bazy zasobu typu *Menu* o kluczu *menuId.*

Możemy również dla jednego typu zasobu utworzyć dwie reguły routingu - z parametrem i bez:

```
mainRouter.attach("/supplierPosition",
SupplierPositionResource.class);
mainRouter.attach("/supplierPosition/{email}",
SupplierPositionResource.class);
```

Pozwala to na używanie tej samej nazwy dla zasobu, zarówno w przypadku gdy chcemy pobierać, jak i edytować obiekty tupu który reprezentuje. Powyższy przykład to reguły odnoszące się do zasobu reprezentującego pozycje GPS dostawcy pizzy. Ścieżka bez parametru dotyczy metody *Put,* która w tym wypadku obsługuje zmianę aktualnej pozycji GPS dostawcy, więc adres email stanowiący identyfikator użytkowników aplikacji, przekazywany jest wraz z danymi o szerokości i długości geograficznej. W przypadku gdy chcemy otrzymać aktualną pozycję dostawcy, korzystamy z metody *Get* za pośrednictwem tej samej ścieżki, jednak na końcu podając adres email (identyfikator dostawcy). Klasa obsługująca powyższe funkcjonalności:

```
public class SupplierPositionResource extends ServerResource{
      private static class SupplierPosition{ /* ... */ }
      String email;

      @Override
      protected void doInit() throws ResourceException { /* ... */ }

      @Get("JSON")
      public SupplierPosition retrieve() { /* ... */ }

      @Put("JSON")
      public void store(SupplierPosition pos) { /* ... */ }
}
```

Przeładowana funkcja *doInit*, obiektu typu *ServerResource,* uruchamiana jest na moment przed dowolną z funkcji reprezentujących metody HTTP.

Funkcje z adnotacjami *@Get("JSON")* oraz *@Put("JSON")* to kolejno funkcje odpowiadające za obsługę metod GET i PUT, natomiast parametr *"JSON"* określa format przesyłanych danych.

W poprzednim rozdziale zaznaczyłem że REST umożliwia przesyłanie danych w dowolnej postaci. Wybrałem JSON do serializacji obiektów JAVA ze względu na czytelność zapisu i łatwość operowania na nim z poziomu dowolnej platformy. Aby obiekt można było wysyłać przez sieć musi on implementować interfejs *Serializable*. Jeżeli warunek ten jest spełniony, wszelkie obiekty wysyłane za pomocą metod z typem danych ustawionym na "JSON", będą automatycznie konwertowane.

Dodatkowo wykorzystując możliwości biblioteki Jackson (JSON processor), za pomocą adnotacji nad definicjami pól klas danych, możemy między innymi:

- pomijać niektóre z pól w JSON'owej reprezentacji obiektu:

```
@JsonIgnore
private Key restaurantId;
```

- dodawać pola o wartościach generowanych przez funkcje:

```
@JsonProperty("restaurantIdStringValue")
public String getRestaurantIdStringValue(){
if(getRestaurantId()!=null)
     return KeyFactory.keyToString(getRestaurantId());
return null;
}
```

Powyższy przykład przedstawia przykład rozwiązania problemu na który na który zwróciłem uwagę przy okazji opisywania sposobu przesyłania klucza z widoku do kontrolera. W tym przypadku również lepiej jest operować na kluczu zakodowanym w postaci ciągu znaków *String,* niż pozwalać na serializację typu *Key*. Urządzeniom mobilnym zdecydowanie łatwiej jest operować na typie *String*, a większość urządzeni, przy założeniu że Web Service ma być uniwersalny, nie byłoby w stanie przywrócić obiektu do właściwej mu postaci sprzed serializacji. Adnotacji @JsonProperty możemy użyć także do określenia sposobu deserializacji obiektów z zewnątrz:

```
@JsonProperty("menuItemSizeKeyStringValue")
public void setMenuItemSizeKey(String menuItemSizeKeyStringValue) {
     this.menuItemSizeKey =
     KeyFactory.stringToKey(menuItemSizeKeyStringValue);
}
```

Powyższy fragment kodu odpowiada za to by wartość pola zakodowanego w JSON jako *menuItemSizeKeyStringValue* została przekonwertowana na typ *Key,* a następnie zapisana w polu klasy o nazwie *menuItemSizeKey*.

Przykład funkcji dla metody PUT:

```
@Put("JSON")
public void store(Order order) {
if (order != null) {
      for(OrderItem oi : order.getOrderItems()){

if(!SizesDAO.exists(oi.getMenuItemSizeKeyStringValue())){
setStatus(org.restlet.data.Status.CLIENT_ERROR_NOT_ACCEPTABLE);
                  return;
            }
      }

if(!RestaurantsDAO.exists(order.getRestaurantKeyStringValue())){

setStatus(org.restlet.data.Status.CLIENT_ERROR_NOT_ACCEPTABLE);
            return;
      }
      order.setDate(new Date()); order.setStatus(Status.PLACED);
      if(!ClientsDAO.exists(order.getClientEmail())){
            Client c = new Client(order.getClientEmail().getEmail(),
order.getPhoneNumber());
            ClientsDAO.save(c);
      }
      //ustalenie ceny dostawy
      order.calculateAndSetDeliveryPrice();
      OrderDAO.save(order);
}
else{ /* ... */ }
}
```

Funkcja jako argument przyjmuje obiekt klasy Order. W praktyce oznacza to że metoda Put dla zasobu o adresie URL /order akceptuje wyłącznie dane JSON, które jest w stanie zdeserializować do postaci obiektu klasy Order. W przypadku gdy deserializacja się nie powiedzie serwer zwraca kod błędu 415, czyli "Unsupported media type" . Funkcja sprawdza czy restauracja oraz produkty, których identyfikatory znajdują się na zamówieniu, istnieją w bazie danych. Jeżeli nie uda się ich odnaleźć, ustawiany jest status:

setStatus(org.restlet.data.Status.CLIENT_ERROR_NOT_ACCEPTABLE);

Obsługa żądania jest następnie przerywana a serwer zwraca powyższy status - kod błędu 406.

Wychwycenie komunikatu serwera o błędzie i weryfikacja przesłanych danych leży po stronie aplikacji na urządzeniu mobilnym.

Przykład funkcji dla metody GET oraz nadpisanej metody *doInit*:

```
Menu menu;
String id;

@Override
protected void doInit() throws ResourceException {
      this.id = (String) getRequest().getAttributes().get("menuId");
      this.menu = MenusDAO.getMenuById(id);
      //...
}

@Get("JSON")
public List<Menu> retrieve() {
      ArrayList<Menu> menusList = new ArrayList<Menu>();
      menusList.add(menu);
      return menusList;
}
```

Funkcja *doInit* odczytuje atrybut *menuId* z URL zasobu */menu,* a następnie na jego podstawie pobiera z bazy obiekt *Menu* i zapisuje go w zmiennej *menu.*

Funkcja *retrieve* tworzy jednoelementową listę (*ArrayList*) obiektów typu *Menu* zawierającą

obiekt do którego referencja przechowywana była w zmiennej *menu*. Na końcu lista ta jest zwracana przez funkcje, konwertowana na zapis w JSON i odsyłana jako odpowiedź serwera na żądanie. Typ *ArrayList* jest w JSON przekształcany na tablice, ale jeżeli chcemy wymusić inne struktury, możemy posłużyć się następującymi zależnościami:

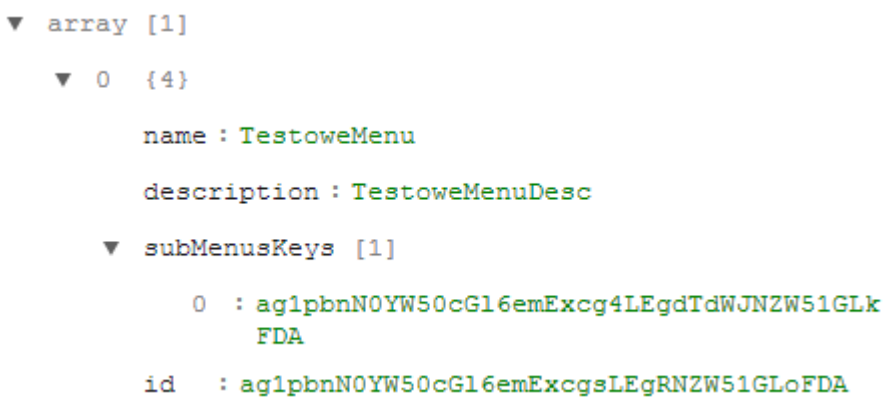
JSON Type	Java Type
`object`	`LinkedHashMap<String,Object>`
`array`	`ArrayList<Object>`
`string`	`String`
`number (no fraction)`	`Integer`, `Long` or `BigInteger` (smallest applicable)
`number (fraction)`	`Double` (configurable to use `BigDecimal`)
`true|false`	`Boolean`
`null`	`null`

Tabela przedstawia odpowiedniki typów JAVA w JSON (JavaScript) - serializacja obiektów

Do przetestowania metody GET wystarczy nam przeglądarka internetowa.
Na moim lokalnym serwerze testowym, wpisanie w przeglądarce adresu URL http://localhost:8888/ws/menu/ag1pbnN0YW50cGl6emExcgsLEgRNZW51GLoFDA (wcześniej skopiowałem z bazy danych klucz losowej restauracji) skutkuje wyświetleniem w oknie przeglądarki następującego wyniku:

[{"name":"TestoweMenu","description":"TestoweMenuDesc","subMenusKeys":["ag1pbnN0YW50cGl6emExcg4LEgdTdWJNZW51GLkFDA"],"id":"ag1pbnN0YW50cGl6emExcgsLEgRNZW51GLoFDA"}]

Aby zapis JSON był bardziej czytelny warto posłużyć się aplikacjami analizującymi składnie JSON takimi jak aplikacja online ze strony www.jsoneditoronline.org . JSON Editor Online przekształca kod do bardziej przyjaznej człowiekowi formy:



Obiekt typu *Menu* zakodowany w JSON

Przetestowanie metody PUT z poziomu przeglądarki niestety nie jest możliwe, ponieważ przeglądarki standardowo obsługują jedynie metody GET oraz POST. Możemy natomiast skorzystać z biblioteki cURL, którą możemy pobrać ze strony http://curl.haxx.se/ , a następnie zainstalować w w swoim systemie operacyjnym. W systemie Windows 7, po dodaniu położenia cURL do zmiennej systemowej PATH możemy bardzo łatwo uruchamiać polecenia z poziomu terminala. Załóżmy że chcemy dodać nowe zamówienie. Wartości niektórych pól obiektu zamówienia wyliczane są automatycznie:

```
Float deliveryPrice; //wysokość opłaty za dowóz w zależności od
                     //odległości, na podstawie współrzędnych GPS

enum Status {PLACED, SENDED, SERVED, APPROVED};
Status status; //po złożeniu zamówienia wartość ustawiana na PLACED


private Date date; //na podstawie aktualnego czasu serwera
```

Pola klas *Order* oraz *OrderItem* wymagane przez serwer zestawie danych dostarczanym przez urządzenie mobilne, oraz ich reprezentacje JSON z przykładowymi wartościami:

<table>
<tr><th>JAVA</th><th>JSON</th></tr>
<tr><td>Email clientEmail</td><td>"clientEmail":"a@op.pl"</td></tr>
<tr><td>List<OrderItem> orderItems</td><td>
<table>
<tr><td>Key menuItemSizeKey
---------->
@JsonProperty
("menuItemSizeKeyString
Value")
public void
setMenuItemSizeKey
(String misksv) { ... }</td><td>"menuItemSizeKeyStringValue":"ag9zfmluc3RhbnRwaXp6YTFyGgsSCE1lbnVJdGVtGMG7AQwLEgRTaXplGAEM"</td></tr>
<tr><td>int quantity</td><td>"quantity":5</td></tr>
</table>
"orderItems":[{"menuItemSizeKeyStringValue":"ag9zfmluc3RhbnRwaXp6YTFyGgsSCE1lbnVJdGVtGMG7AQwLEgRTaXplGAEM","quantity":5}]
</td></tr>
<tr><td>String phoneNumber</td><td>"phoneNumber":"555666444"</td></tr>
<tr><td>double longitude</td><td>"longitude":20.693368</td></tr>
<tr><td>double latitude</td><td>"latitude":49.621679</td></tr>
<tr><td>@JsonIgnore
Key restaurantKey;
---------->
@JsonProperty("restaurantKeyStringValue")
void setRestaurantKey(String rksv) { ... }</td><td>"restaurantKeyStringValue":"ag9zfmluc3RhbnRwaXp6YTFyEgsSClJlc3RhdXJhbnQY8asBDA"</td></tr>
</table>

Połączenie powyższych węzłów w jeden obiekt JSON (czyli nieuporządkowany zbiór par w postaci *nazwa : wartość,* gdzie wartość może być liczbą, ciągiem znaków, tablicą, obiektem, wartością logiczną lub nullem), daje następujący kod:

```
{"longitude":20.693368,"latitude":49.621679,"clientEmail":"a@op.pl"
,"phoneNumber":"555666444","restaurantKeyStringValue":"ag9zfmluc3Rh
bnRwaXp6YTFyEgsSClJlc3RhdXJhbnQY8asBDA","orderItems":[{"menuItemSiz
eKeyStringValue":"ag9zfmluc3RhbnRwaXp6YTFyGgsSCE1lbnVJdGVtGMG7AQwLE
gRTaXplGAEM","quantity":5}]}
```

Teraz wygenerowany obiekt wystarczy przesłać za pomocą cURL na adres URL reprezentujący zasób Order czyli w przypadku lokalnego serwera testowego: http://localhost:8888/ws/order

```
curl -i -H "Content-type:application/json" -X PUT --data
{\"longitude\":20.693368,\"latitude\":49.621679,\"clientEmail\":\"a
@op.pl\",\"phoneNumber\":\"555666444\",\"restaurantKeyStringValue\"
:\"ag9zfmluc3RhbnRwaXp6YTFyEgsSClJlc3RhdXJhbnQY8asBDA\",\"orderItem
s\":[{\"menuItemSizeKeyStringValue\":\"ag9zfmluc3RhbnRwaXp6YTFyGgsS
CE1lbnVJdGVtGMG7AQwLEgRTaXplGAEM\",\"quantity\":5}]}
localhost:8888/ws/order
```

Powyższy przykład uruchamiany był w linii komend systemu Windows, dlatego cudzysłowie poprzedzone są ukośnikami wstecznymi. Zabieg ten sprawia że cudzysłów przestaje być znakiem specjalnym konsoli i jest traktowany jako zwykły element ciągu znaków. Uruchamiając tę samą komendę w systemie Linux, ukośniki należy pominąć. Parametr *-i* oznacza że po wykonaniu komendy, wyświetlony zostanie nagłówek odpowiedzi serwera - pozwala to na odczyt statusu żądania:

```
HTTP/1.1 406 Not Acceptable
Content-Type: text/html; charset=UTF-8
Date: Thu, 04 Jul 2013 16:34:19 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.2snapshot
Content-Length: 574
```

Opowiedź z serwera - kod błędu 406

Parametr *-H* pozwala na dodanie pól do nagłówka żądania - `"Content-type:application/json"` określa typ MIME przesyłanych danych. MIME to standard dwuczęściowych identyfikatorów dla typów danych przesyłanych w sieci internet.
W przypadku powyższej komendy typem danych jest obiektowa notacja języka JavaScript - JSON.
Parametr *-X* służy do określenia metody HTTP. W naszym przypadku wartość tego parametru występująca tuż za jego symbolem to `PUT`. Ostatni opcjonalny parametr *--data* poprzedza dane które chcemy przesłać - w powyższym przykładzie reprezentacje obiektu Order (zamówienie).
Na samym końcu należy dopisać adres URL na jaki chcemy przesłać żądanie.
Po dodaniu zamówienia powinno być ono widoczne na liście zamówień użytkownika dostępnej z poziomu *http://localhost:8888/ws/order/a@op.pl :*

[{"clientEmail":{"email":"a@op.pl"},"supplierEmail":null,"orderItems":[{"quantity":5,"menuItemSizeKey":"ag1pbnN0YW50cGl6emExchoLEghNZW51SXRlbRi3BQwLEgRTaXplGMoGDA"}],"date":1372976494654,"phoneNumber":"555666444","longitude":20.693368,"latitude":49.621679,"deliveryPrice":14502.0,"status":"PLACED","clientEmailString":"a@op.pl","totalPrice":14562.0,"restaurantKey":"ag1pbnN0YW50cGl6emExchELEgpSZXN0YXVyYW50GMcGDA","date_day":"05-07-2013"}]

2.7. Transfer plików

Platforma-usługa GAE działa w chmurze - w izolowanym środowisku uruchomieniowym zwanym “piaskownicą”. Fakt ten niesie za sobą kilka ograniczeń, które jednak z uwagi na profil zastosowań usługi i pakiet zamiennych funkcjonalności nie są uciążliwe. Pierwsza ograniczenie to brak możliwości uzyskania bezpośredniego dostępu do struktury plików na serwerze na którym uruchomiona jest nasza aplikacja - wynika to ze specyfiki chmury - nie da się określić gdzie fizycznie znajdują się pliki aplikacji, ani jaka konkretnie maszyna wykonuje jej kod w danej chwili. W tej sytuacji na myśl przychodzi niezależny serwer FTP - nie jest to jednak rozwiązanie optymalne, gdyż do jego obsługi potrzebowalibyśmy jeszcze usługi third-party (np. PHP Web Service), obsługującej operacje protokołu transferu plików FTP. Wynika to z tego, że aplikacja GAE działające w izolowanej “piaskownicy”, ze względów bezpieczeństwa dopuszcza wyłącznie komunikację opartą o protokoły HTTP oraz HTTPS (zaszyfrowana wersja HTTP wykorzystującą protokół SSL).

Jeżeli założenia projektu nie wymuszają użycia konkretnego serwera do przechowywania plików, warto rozważyć wykorzystanie Blobstore. Za pośrednictwem Blobstore API, platforma oferuje możliwość przechowywania obiektów (data objects) znacznie większych niż za pośrednictwem Datastore. InstantPizza wymaga przechowywania obrazków prezentujących wygląd poszczególnych potraw z karty dań, lecz przechowywanie tak dużych porcji danych w bazie byłoby bardzo nieefektywne. Z tego względu zdecydowałem wykorzystać Blobstore do zaimplementowania interfejsu CRUD (create, read, update, delete) dla zdjęć potraw.

Zapis

Na początek widok:

```
<form action="<%= blobstoreService.createUploadUrl("/menuItems")
%>"
method="post" enctype="multipart/form-data" accept-charset="utf-8">
<!-- ... -->
<table>
<!-- ... -->
<tr>
    <td> Miniatura: </td>
    <td>
      <input required="required" type="file" name="thumbnail"/>
    </td>
</tr>
<!-- ... -->
</table></form>
```

Funkcja *createUploadUrl* tworzy adres upload'u obrazka:

```
<form accept-
charset="utf-8" enctype="multipart/form-
data" method="post" action="http:
//instantpizza1.appspot.com/_ah/upload
/AMmfu6YJ_u0GktWoWUAuG3nTkogCH2w5D6V1vD1
j44s8krA-
RjrU706_5mML2EcpJ0x0gTaNV9D_32fnWdO7ZrGs
e_HyX05h-fT4MGVwvzjikB-
dt1z4rKWxW4TvWIosAAUJrnusqwdyPWqKr5FGaAZ
NkpwRblUDpQ
/ALBNUaYAAAAAUdXQK2qFKAPtlWKWhBAKiseTtDR
sGKWL/">
```

Po zapisaniu obrazka na serwerze obsługa żądania z formularza przekazywana jest do servletu którego adres URL podany był jako argument funkcji.

Zapis *enctype="multipart/form-data"* jest konieczny, jeżeli wraz z danymi chcemy za pośrednictwem formularza przesyłać pliki.

<input required="required" type="file" name="thumbnail"/> - pole formularza umożliwiające otwarcie systemowego okna wyboru pliku - jedyna możliwość dostępu do systemu plików z poziomu przeglądarki internetowej.

Wysłanie formularza powoduje automatyczny zapis obrazka na serwerze, jednak jeżeli chcemy mieć do niego dostęp w przyszłości musimy zapisać jego położenie w bazie danych.
Służy do tego pole typu *BlobKey:*

```
@PersistenceCapable
public class MenuItem implements Serializable{
	//...
	@Persistent
	@JsonIgnore
	private BlobKey thumbnailBlobKey;
	//...
}
```

W pliku kontrolera (servlet) obsługa żądania z dołączonym obrazkiem wygląda następująco:

Najpierw należy zadeklarować zmienna *BlobstoreService* w klasie obsługującego serwletu:
private BlobstoreService blobstoreService = BlobstoreServiceFactory.getBlobstoreService();
Następnie w funkcji *doPost* definiujemy kolekcje do przechowywania "blobów" i inicjalizujemy ją null'em: *Map<String, List<BlobKey>> blobs = null;*
Kolejnym krokiem jest pobranie kluczy wszystkich przesłanych obrazków (Blobs):
blobs = blobstoreService.getUploads(req);
Należy zauważyć że klucze te przechowywane są w zmiennej *HttpServletRequest,* czyli zmiennej przechowywującej parametry żądania.
Na końcu zapisujemy tylko pobrany *BlobKey* w polu *ThumbnailBlobKey* obiektu MenuItem:

```
MenuItem m = new MenuItem();
//...
for (BlobKey key : blobs.get("thumbnail")){
	m.setThumbnailBlobKey(key);
}
```

I zapisujemy zmiany:

MenuItemsDAO.save(m);

Odczyt

Aby uzyskać dostęp do obrazków za pośrednictwem adresu URL, najlepiej utworzyć do tego celu osobny kontroler:

```
public class ImagesServlet extends HttpServlet {
   private BlobstoreService blobstoreService =
       BlobstoreServiceFactory.getBlobstoreService();
   public void doGet(/* ... */) throws IOException {
       BlobKey blobKey = new BlobKey(req.getParameter("blob-key"));
       blobstoreService.serve(blobKey, resp);
   }
   //...
}
```

Serwlet wykorzystując obiekt typu *BlobstoreService* tworzy na bazie reprezentacji klucza w postaci ciągu znaków obiekt typu *BlobKey.* Klucz ten jest następnie przekazywany do funkcji *serve* wraz z referencją do obiektu odpowiedzialnego za odpowiedź na żądanie czyli obiektu typu *HttpServletResponse*. Funkcja umieszcza dane obrazka w ciele odpowiedzi HTTP.

Na widoku panelu administracyjnego możemy funkcjonalność oferowana przez serwlet wykorzystać w następujący sposób (zmienna *m* przechowuje obiekt typu *MenuItem* pobrany wcześniej z bazy za pośrednictwem klasy *MenuItemsDAO*):

```
<img src="<% if(m.getThumbnailBlobKey()!=null)
out.print("/img?blob-key="+m.getThumbnailBlobKey().getKeyString());
%>" />
```

Analogicznie obrazki mogą być pobierane przez urządzenia mobilne. Klasa *MenuItem* implementuje metodę, która za pomocą adnotacji *@JsonProperty* dynamicznie generuje wartość dla pola *thumbnailUrl* w pobieranej z serwera reprezentacji obiektu w JSON:

```
@JsonProperty("thumbnailUrl")
public String createThumbnailUrl() {
      return "/img?blob-key="+
      this.getThumbnailBlobKey().getKeyString();
}
```

Przykładowo efektem uruchomienia w przegądarce adresu dla zasobu *menuItem* o *id=ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGJHIAgw,* czyli *http://instantpizza1.appspot.com/ws/menuItem/ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGJHIAgw ,* jest następujący obiekt JSON*:*

[{"name":"Pączek","description":"z_czekoladą","id":"ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGJHIAgw","thumbnailUrl":"/img?blob-key=AMIfv955Lnj9YtZ9IJRm5aFkcFHOBKmv2swtaH0V36Zjge9YBcVP1huIPYJWRWeTqs4WRrRN7jkA0_w7SKk3dIYeJBPLY9u8clVBaPPTs8qgssUKPMgXXJEVANBdMmk-ftBSmNzjxOBlvvIqAzrgQ0dbUJA1JHhMjmvZleobMawAt52lgbxQJHA"}]

Wpisanie w przeglądarce internetowej adresu *http://instantpizza1.appspot.com/img?blob-key=AMIfv955Lnj9YtZ9IJRm5aFkcFHOBKmv2swtaH0V36Zjge9YBcVP1huIPYJWRWeTqs4WRrRN7jkA0_w7SKk3dIYeJBPLY9u8clVBaPPTs8qgssUKPMgXXJEVANBdMmk-ftBSmNzjxOBlvvIqAzrgQ0dbUJA1JHhMjmvZleobMawAt52lgbxQJHA*
spowoduje wyświetlenie w oknie obrazka, któremu przypisany jest klucz z parametru *blob-key.*

2.8. Ograniczenia dostępu do panelu administracyjnego InstantPizza

Google App Engine udostępnia trzy podstawowe, wbudowane metody kontroli dostępu do treści:

- Google Accounts API - korzysta z uniwersalnego systemu logowania Google, wystarczy być zalogowanym do dowolnej usługi Google, aby zalogować się do aplikacji - decyzja jakie zasoby udostępniać konkretnym użytkownikom leży po stronie aplikacji
- Google Apps Domain - umożliwia dostęp tylko użytkownikom usług Google, znajdujących się w określonej domenie (Google oferuje płatną usługę Google Apps for Business, umożliwiającą podpięcie zewnętrznej domeny)
- Federated Login (w fazie eksperymentalnej) - umożliwia logowanie za pośrednictwem OpenID, który udostępniają m.in. następujące serwisy:
    - yahoo.com
    - myspace.com
    - aol.com
    - myopenid.com

Mój wybór padł na metodę najbardziej uniwersalną czyli **Google Accounts API**.
Każdy z użytkowników panelu administracyjnego identyfikuje się unikalnym adresem email:

```
public abstract class User implements Serializable{
      @PrimaryKey
      private String email;
      //...
}
public class CMSUser extends User{
      public enum Role {ADMIN, RESTAURATEUR};
      @Persistent
      private Role role=null;
      @Persistent
      @JsonIgnore
      private Key restaurantId;
      //...
}
```

Dodatkowo użytkownik może mieć przypisaną jedną z dwóch ról (administrator lub restaurator), oraz identyfikator restauracji(założenie: każde konto restauratora może być przypisane tylko do jednej restauracji).

Dostosowanie mechanizmu oferowanego przez Google do własnych potrzeb wymagało napisania klasy implementującej interfejs *UserService*:

```
class AdminService implements UserService {
	UserService service = null;
	public AdminService() {
		service = UserServiceFactory.getUserService();
	}
	//...
	@Override
	public User getCurrentUser() {
		//...
	}
	//...
	@Override
	public boolean isUserLoggedIn() {
		//...
	}
}
```

Konstruktor klasy do zmiennej *service* przypisuje referencję do *UserService* pobiera ona z sesji przeglądarki informacje na temat użytkownika zalogowanego obecnie w usługach google.

Implementacja metody *getCurrentUser:*

```
@Override
public User getCurrentUser() {
	User usr = service.getCurrentUser();
	if (usr == null) return null;
	String email = usr.getEmail();
	//...
	if(CMSUsersDAO.existsAndEnabled(new Email(email))){
```

```
            return usr;
      }
      return null;
}
```

Do sprawdzenia czy dany użytkownik Google, istnieje również w InstantPizza i nie jest zablokowany wykorzystywana jest funkcja *existsAndEnabled* obiektu *CMSUsersDAO*:

```
public static boolean existsAndEnabled(Email email){
      CMSUser u = CMSUsersDAO.getCMSUserByEmail(email);
      if(u != null){
            if(u.isActive() && !u.isBanned()) return true;
      }
      return false;
}
```

Implementacja funkcji isUserLoggedIn, wygląda natomiast następująco:

```
@Override
public boolean isUserLoggedIn() {
      User usr = service.getCurrentUser();
      //...
      if(CMSUsersDAO.existsAndEnabled(new Email(usr.getEmail())))
return true;
      return false;
}
```

Wykorzystuje również metodę *existAndEnabled* do sprawdzenia czy użytkownik istnieje w systemie.
Przykład wykorzystania utworzonej klasy do ograniczenia dostępu niepowołanym użytkownikom:

```
<%
    UserService userService = AdminServiceFactory.getUserService();
    User user = userService.getCurrentUser();
%>
```

Powyższy kod zapisuje w zmiennej *user* dane obecnie zalogowanego użytkownika Google. Wykorzystuje do tego klasę *AdminService*, uzyskiwaną za pośrednictwem:

```
public class AdminServiceFactory {
      public static UserService getUserService() {
          return new AdminService();
      }
}
```

Jest to fabryka abstrakcyjna, oferująca metodę *getUserService,* tworząca instancje klasy AdminService i zwracająca je jako wynik.

Zgodnie ze zdefiniowanymi zasadami, jeżeli użytkownika nie istnieje w systemie, lub został celowo zablokowany przez administratora, zmienna *user* przyjmie wartość *null.*

```
<%  if (user != null) {  %>
      Ta treść będzie widoczna tylko dla zalogowanych użytkowników.
<% } %>
      Ta treść będzie widoczna dla wszystkich.
<%  if (user == null) {  %>
      Użytkownicy niezalogowani mogą użyć następującego odnośnika
      do zalogowania się:
      <a href="<%=
userService.createLoginURL(request.getRequestURI())
      %>">
<%  }  %>
```

createLoginURL tworzy link przenoszący użytkownika na stronę logowania do usługi google. Po poprawnym zalogowaniu następuję przekierowania na docelowy adres podany jako parametr funkcji. Adres jest zakodowany w linku do strony logowania, w parametrze *continue*:

https://www.google.com/accounts/ServiceLogin?service=ah&passive=true&continue=https://appengine.google.com/_ah/conflogin%3Fcontinue%3Dhttp://instantpizza1.appspot.com/jsp/adminPanel.jsp&ltmpl=gm&shdf=CiQLEgZhaG5hbWUaGEluc3RhbnQgUGl6emEgUHJvZHVjdGlvbgwSAmFolhSAc_BDMC1bbpMAbienaxHtTdjzQygBMhQGUJN024FT6-nso2sgW6hJ3jXhVA

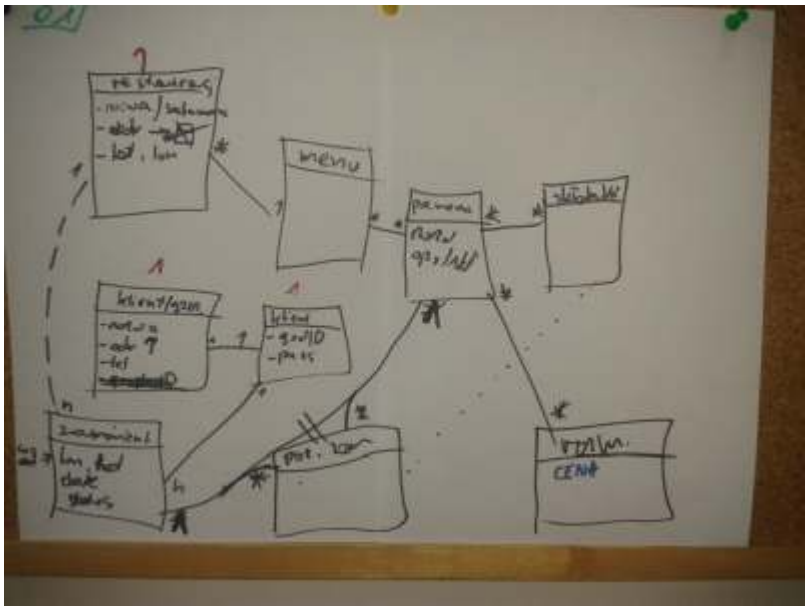
3. FUNKCJONALNOŚĆ APLIKACJI

3.1. Wprowadzenie

W tym rozdziale postaram się przybliżyć jak aplikacji realizuje opisane we wstępie zagadnienia, model biznesowy, a także podstawowy scenariusz użycia aplikacji, od momentu utworzenia nowej restauracji aż do złożenia zamówienia i dostawy. Wszystko to ujmę w kontekście serwera, a do symulacji urządzenia mobilnego posłużę się biblioteką cURL. Na początek jednak opiszę przebieg grupowej pracy nad projektem, podział zadań, a także sposób wymiany informacji na temat stopniowo rozrastającego się interfejsu komunikacji na linii klient - serwer.

3.2. Podział zadań w projekcie InstantPizza

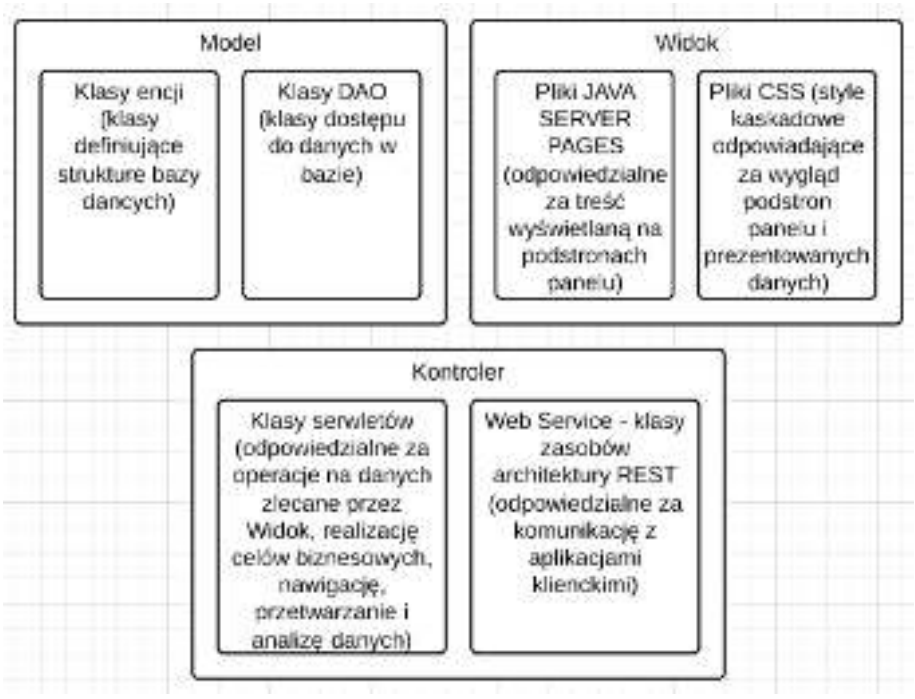
Pomysł na aplikację, jak to zwykle bywa, zrodziła potrzeba - "potrzeba matką wynalazku". Nad docelowymi funkcjonalnościami, założeniami i sformalizowaniem celów pracowaliśmy wspólnie z moim kolegą. Pierwsze spotkania w naszej (dwuosobowej) grupie projektowej miały charakter "burzy mózgów". Wymienialiśmy się pomysłami, proponowaliśmy biznesowe rozwiązania i weryfikowaliśmy ich optymalność, a także szacowaliśmy czas potrzebny na ich implementację. Rozważaniom nad modelem biznesowym aplikacji towarzyszył wstępny szkic struktury danych. W formie ciekawostki, zdjęcie pierwszego, odręcznie narysowanego, wielokrotnie zmodyfikowanego później diagramu danych:



Zaprojektowanie i optymalizacja modelu bazy danych na bazie sporządzonych poglądowo szkiców leżała po mojej stronie i musiałem poświęcić na to nieco czasu po każdym spotkaniu.
Po ustaleniu wstępnej koncepcji (w trakcie implementacji uległa ona sporej modyfikacji - z części funkcjonalności zrezygnowaliśmy, a kilka nowych zostało dodanych do projektu) podzieliliśmy się zadaniami. Zdecydowaliśmy że ja zajmę się serwerem, a mój kolega napisaniem mobilnej aplikacji na system Android. Taki podział okazał się bardzo dobrym rozwiązaniem, ponieważ nie musieliśmy się wdrażać nawzajem w swoje rozwiązania ani studiować swoich kodów źródłowych - każdy odpowiadał za implementację tylko swojej części, która jednocześnie dla drugiego stanowiła "czarną skrzynkę". Dzięki temu każdy mógł w niezależnie od partnera, w dogodnym dla siebie czasie pracować nad swoją częścią. Jedynie raz na jakiś czas musieliśmy spotkać się aby ustalić priorytety, kolejne cele, podsumować postępy prac i przekazać sobie specyfikację interfejsów lub ustalić ewentualne modyfikacje już istniejących. W zasadzie jedyne informacje techniczne jakie musiałem udostępnić programiście aplikacji mobilnej to adresy URL zasobów oraz strukturę obiektów JSON stanowiących nośniki informacji przekazywanych na linii klient - serwer.

3.3. Struktura aplikacji

Od strony architektury aplikacja opiera się o wzorzec MVC. Podział aplikacji InstantPizza (serwer) na poszczególne komponenty, w dużym uogólnieniu prezentuje poniższy diagram:



Podział aplikacji na warstwy modelu MVC

Na płaszczyźnie funkcjonalnej (biznesowej) serwer dzieli się na cztery główne części:

- CMS (system zarządzania treścią) - zarządzanie danymi restauracji, kartami dań, jednostkami (jako administrator)
- Centrala przyjmowania zamówień - po zalogowaniu się do systemu jako restaurator, otrzymujemy wgląd do ewidencji zamówień
- Zarządzanie użytkownikami i ograniczenia dostępu - administrator systemu ma pełną kontrolę nad użytkownikami, a konkretne grupy użytkowników mają określone uprawnienia i ograniczenia
- Moduł zamówień - obsługa zamówień, wymiana danych pomiędzy klientami i dostawcami, działa przez cały czas, niezależnie od pozostałych części aplikacji.

Wszystkie moduły opierają się na wspólnej bazie danych, a żadnego z nich nie da się pominąć zachowując zdolność systemu do realizacji założonych celów biznesowych.

3.4. Projekt bazy danych

Baza danych determinuje sposób działania aplikacji, a błędy podczas jej projektowania mogą prowadzić do uniemożliwienia realizacji celów biznesowych, lub sprawić że cele będą realizowane nieoptymalnie. Dlatego etap projektowania bazy danych, wymaga poświęcenia bardzo dużej ilości czasu i dokładnego przeanalizowania efektów pracy. Projektując poszczególne encje bazy danych starałem się aby były one zgodne trzecią postacią normalną (3NF). Oznacza to że każda encją musi spełniać wymagania trzeciej, jak również pierwszej i drugiej postaci normalnej:

- 1NF - należy zapewnić atomowość danych, czyli w praktyce jedno pole (kolumna) bazy danych może zawierać tylko jeden typ informacji (pojęcie to jest oczywiście umowne, najczęściej jako niezależny typ danych należy traktować minimalną porcję danych która może być przez aplikację wykorzystana w przyszłości).
- 2NF - każda klasa danych powinna zawierać informację o tylko jedym typie danych - wynika to również z ustalonych założeń. Przykładowo w InstantPizza zamówienia i klienci nie mogą być przechowywani w tej samej tabeli, pomimo iż obiekty te są ze sobą powiązane

- 3NF - żadna kolumna która nie należy do klucza nie może być zależna od innej kolumny. Np. w InstantPizza w pozycji zamówienia przechowywany jest rodzaj artykułu i ilość sztuk, natomiast cena nie jest przechowywana w osobnej kolumnie lecz wyliczana w następujący sposób:

```
@JsonProperty("totalPrice")
public float totalPrice(){
      float sum=this.getDeliveryPrice();
      for (OrderItem orderItem : this.getOrderItems()) {
      sum+=orderItem.getQuantity()*
      SizesDAO.getSizeById(orderItem.getMenuItemSizeKey()).getPrice(
);
      }
      return sum;
}
```

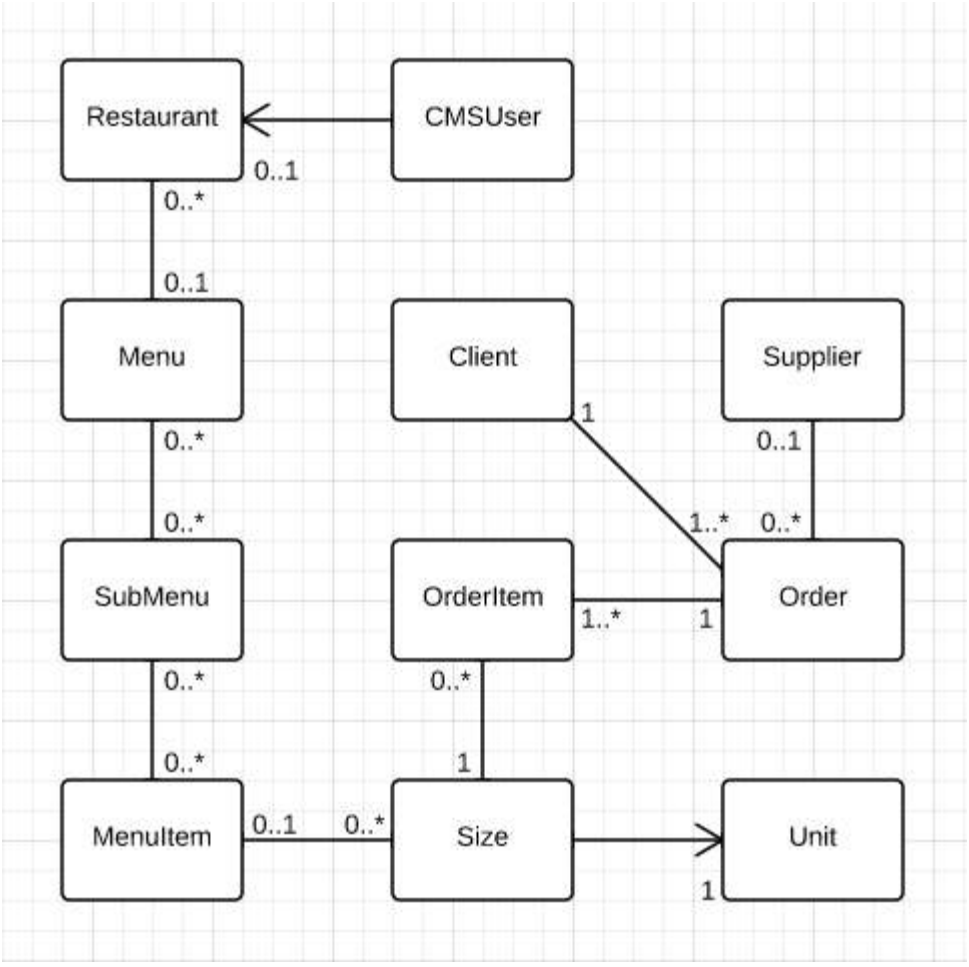


Diagram relacyjny bazy danych

Objaśnienia dla kilku wybranych powiązań i liczności:

- Order - Client - zamówienie może być utworzone wyłącznie przez klienta, nigdy więcej niż jednego. Klient jest zawsze przypisany do przynajmniej jednego zamówienia, ponieważ konto klienta tworzone jest automatycznie po złożeniu pierwszego zamówienia.
- Client - Order - klient jest zawsze przypisany do przynajmniej jednego zamówienia, ponieważ konto klienta tworzone jest automatycznie po złożeniu pierwszego zamówienia.
- Size - Unit - rozmiar zawsze wiąże się z jednostką, ponieważ sama wartość liczbowa bez określenia miary nie ma żadnego biznesowego znaczenia, nigdy z więcej niż jedną.
- OrderItem - Order - zamówienie nie może nie może istnieć bez przynajmniej jednej pozycji zamówienia, natomiast pozycja zamówienia zawsze tworzona jest w trakcie tworzenia zamówienia
- Menu - SubMenu - MenuItem - te trzy klasy służą do komponowania karty dań, dlatego powiązania pomiędzy tymi elementami mogą występować w dowolnych licznościach
- Restaurant - CMSUser - relacja jest kierunkowa ponieważ dotyczy ona uprawnień użytkowników, z przypisaną rolą "Restaurator", do edycji ustawień konkretnej restauracji. Restauracja nie musi przechowywać informacji o tym jacy użytkownicy mogą modyfikować jej dane.

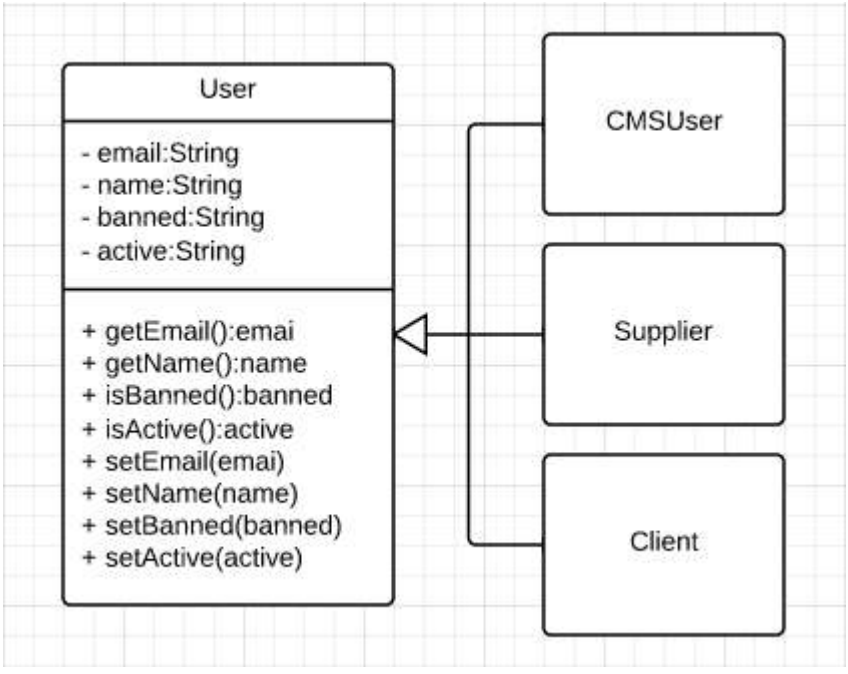


Diagram klas przedstawiający grupę klas dziedziczących po klasie *User*

Każdy rodzaj użytkowników aplikacji, z racji wykorzystania logowania za pośrednictwem google, identyfikuje się unikalnym adresem email, nazwa wymagana jest do komunikacji z użytkownikami na poziomie frontend (czyli m.in. w widokach aplikacji, a także w aplikacji mobilnej), natomiast pola *banned* i *active* służą blokowania i usuwania użytkowników. Wszystkie te wspólne cechy zostały zaimplementowane w klasie *User,* stanowiącej bazę dla obiektów *CMSUser, Supplier, Client* - zależność tą przedstawia powyższy diagram.

Sprawa ma się podobnie w przypadku klas *Menu* i *SubMenu*. Każdy z tych elementów dysponuje nazwą i opisem które mogą być później wyświetlone na ekranie aplikacji mobilnej.

Dodatkowo na jednym z ekranów panelu administracyjnego obiekty *Menu* i *SubMenu* wyświetlane są na jednej liście:

Nazwa menu	Opis	Typ	Akcje
TestoweMenu EDYTUJ USUŃ	TestoweMenuDesc	Menu	Edytuj podmenu
TestoweSubMenu EDYTUJ USUŃ	TestoweSubMenuDesc	SubMenu	Edytuj pozycje

Aby ułatwić sobie zaprogramowanie mechanizmu wyświetlania pozycji, wykorzystując polimorfizm, musimy w interfejsie który implementować bedą powyższe klasy dopisać także nagłówek metody *getId*. Zastosowanie takiego zabiegu jest konieczne, ponieważ jeżeli zmienna typu będącego interfejsem lub klasą nadrzędna, może wykonywać wyłącznie zadeklarowane w swoim ciele metody obiektu klasy podrzędnej, którego referencję przechowuje. Wówczas można fragment kodu odpowiadającego za wyświetlenie pojedynczego elementu pobranej kolekcji zapić w następujący sposób:

```
<%
List<IMenu> menus = MenusDAO.getIMenus();
for (IMenu m : menus) {
%>
```

```
<tr class="tableRow">
<td>
<%= m.getName() %> <br/>
<button onclick="editMenu('<%=
KeyFactory.keyToString(m.getId())%>',
'<%=m.getName() %>', '<%=m.getDescription() %>' /* ... */);
return false;">EDYTUJ</button>
<!-- ... -->
</td>
<td><%= m.getDescription() %></td>
<!-- ... -->
</tr>
<%  } %>
```

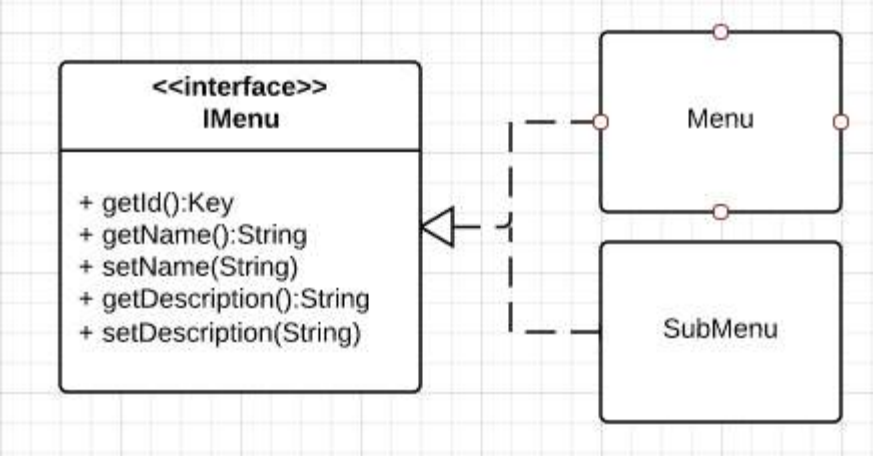


Diagram klas prezentujący relację dziedziczenia po interfejsie *IMenu*

Niestety DataStore nie umożliwia pobierania z bazy kolekcji obiektów implementujących określony interfejs. Wykonanie poniższego kodu zakończyłoby się niepowodzeniem:

```
Query q = pm.newQuery(IMenu.class);
q.execute();
```

Zamiast tego należy wykonać osobne zapytanie dla obydwu typów, a następnie połączyć je w jedną listę:

```
Query q = pm.newQuery(Menu.class);
Query q2 = pm2.newQuery(SubMenu.class);
list = (List<IMenu>) q.execute();
list2 = (List<IMenu>) q2.execute();
List<IMenu> all=new ArrayList<IMenu>();
for(IMenu m : list){all.add(m);}
for(IMenu m : list2){all.add(m);}
```

3.5. Opis działania aplikacji

W rozdziale tym opiszę jak działanie serwera z punktu widzenia jego użytkowników. Opis może stanowić także instrukcję obsługi dla ewentualnych przyszłych użytkowników. Przedstawiony przeze mnie scenariusz to tylko przykład przedstawiający jedną z wielu ścieżek jakimi użytkownik może podążyć, ale większość operacji może być wykonywana analogicznie - zwykle zmienić mogą się jedynie wprowadzane wartości. Sama obsługa panelu administracyjnego nie powinna sprawiać trudności nawet niedoświadczonemu użytkownikowi, ponieważ nawigacja oparta jest o sprawdzone i stosowane na wielu witrynach menu boczne, dostępne z każdej podstrony, natomiast dane wprowadzane są za pomocą standardowych formularzy HTML, z którymi każdy użytkownik internetu zapewne miał do czynienia.

Instalacja

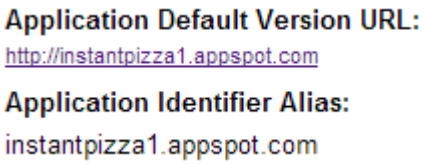
Załóżmy na początek że aplikację mamy na razie tylko w postaci kodu źródłowego - projektu Google App Engine w środowisku Eclipse. Pierwszy krok to umieszczenie aplikacji na serwerze. W tym celu w Eclipse wybieramy opcję *Deploy to App Engine...* . W kolejnym oknie wybieramy *App Engine project settings...* , uzupełniamy informacje o nazwie aplikacji (miejsce dla aplikacji o określonej nazwie zarezerwowane zostało wcześniej, z poziomu panelu na stronie *appengine.google.com*), a także o nazwie wersji którą chcemy umieścić na serwerze:

Zamykamy ekran ustawień i wybieramy przycisk *Deploy*.

Teraz w panelu administracyjnym *appengine.google.com* wystarczy przełączyć się na nową wersję. Co ważne, baza danych (*Datastore*) dla wszystkich wersji jest wspólna, więc mimo iż na serwerze uruchomiona jest nowa wersja aplikacji, dane wprowadzone w poprzednich wersjach nie ulegną zmianie.W tym przypadku na serwerze jest już kilka wprowadzonych testowo rekordów dotyczących restauracji.

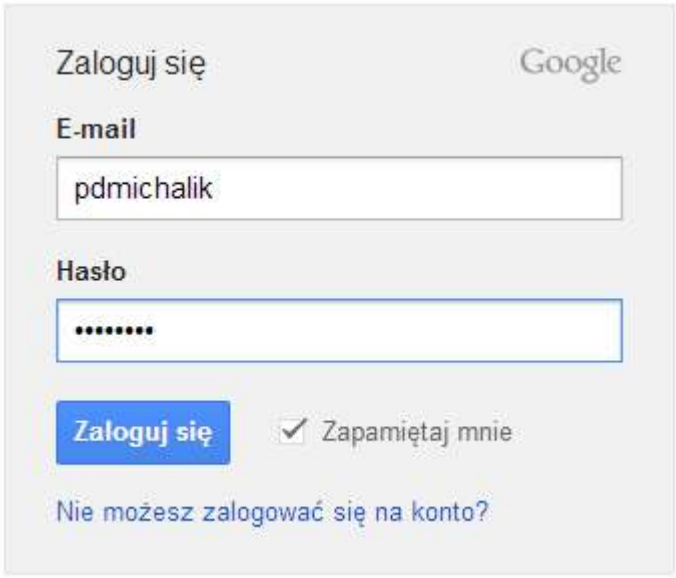
Wersja alfa-4 ustawiona jako domyślna

W menu *Application Settings* możemy sprawdzić pod jakim adresem dostępna jest nasza aplikacja:

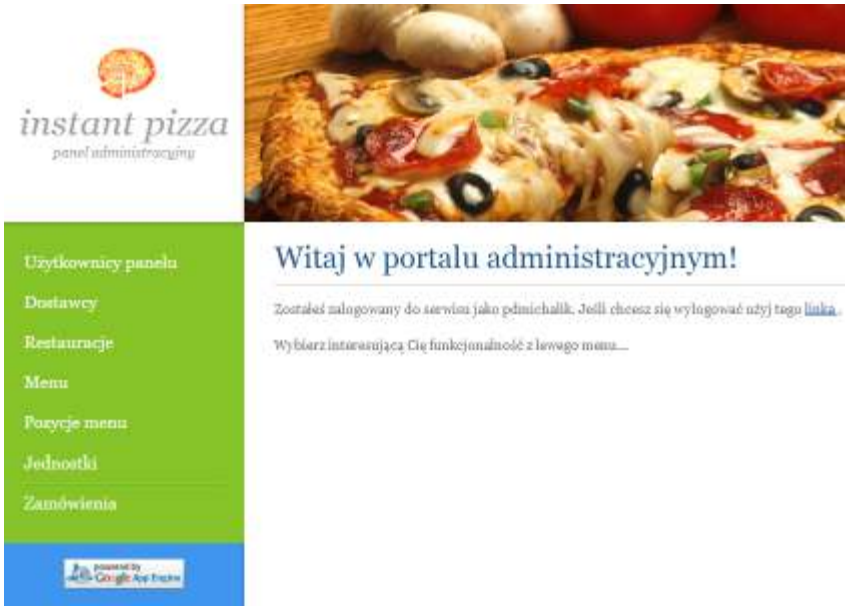


Application Default Version URL to adres wersji którą ustawiliśmy jako domyślną. Co ważne, możemy też w dowolnej chwili uzyskać dostęp do dowolnej wersji - działają one równolegle.
Wystarczy nazwy wersji użyć jako nazwy subdomeny w domenie aplikacji. Przykładowo dla wersji alfa-4 będzie to adres: *alfa-4.instantpizza1.appspot.com* .
Po otwarciu aplikacji w przeglądarce, o ile nie jesteśmy zalogowani do usługi Google, naszym oczom ukaże się następująca podstrona z informacją o konieczności zalogowania się:

Jeżeli mamy już swoje konto Google zarejestrowane w usłudze, wybieramy “potwierdź swoją tożsamość”:

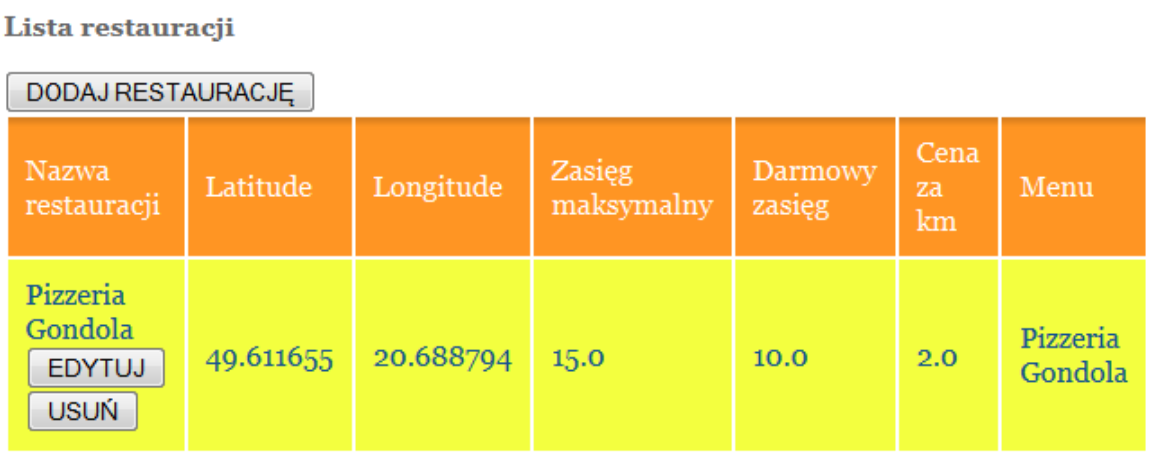


Wpisujemy swoje dane logowania i klikamy Zaloguj się (wygląd okienka może ulec zmianie - jest to podstrona należąca do Google). Zostajemy automatycznie przekierowani na stronę główną panelu InstantPizza:



CMS

Załóżmy że nasza firma otwarła nową restaurację i chcemy dodać ją do systemu. Wybieramy pozycję *Restauracje* w menu bocznym, aby wyświetlić listę istniejących w systemie restauracji:



Aby dodać nową restaurację do listy, wybieramy przycisk *DODAJ RESTAURACJĘ*.

Kod przycisku wygląda następująco:

<button onClick="addRestaurant(); return false;">DODAJ RESTAURACJĘ</BUTTON>

Dzięki możliwościom JavaScript możemy zdefiniować akcję dla zdarzenia kliknięcia myszką.

W tym wypadku kliknięcie w przycisk uruchamia funkcje JS (JavaScript) zdefiniowaną jako:

```
function addRestaurant() {
    $("#add").show();
}
```

Funkcja ta wykorzystuje bibliotekę jQuery do wyświetlenia okienka dodawania nowej restauracji - modyfikuję style CSS (kaskadowe arkusze stylów) elementu *div* przechowującego formularz:

Nazwa restauracji to wartość wyświetlana na urządzenie mobilnym, na liście restauracji.

Pola *Latitude* i *Longitude* to kolejno szerokość i długość geograficzna punktu w którym zlokalizowana jest dana restauracja. *Zasięg maksymalny* to największa odległość (w kilometrach) na jaką restauracja oferuje dowóz. *Darmowy zasięg* to maksymalna odległość darmowego dowozu do klienta. *Cena za km* to wysokość dodatkowej opłaty za każdy dodatkowy kilometr powyżej wartości darmowego zasięgu. *Menu,* to pole wyboru karty dań restauracji.

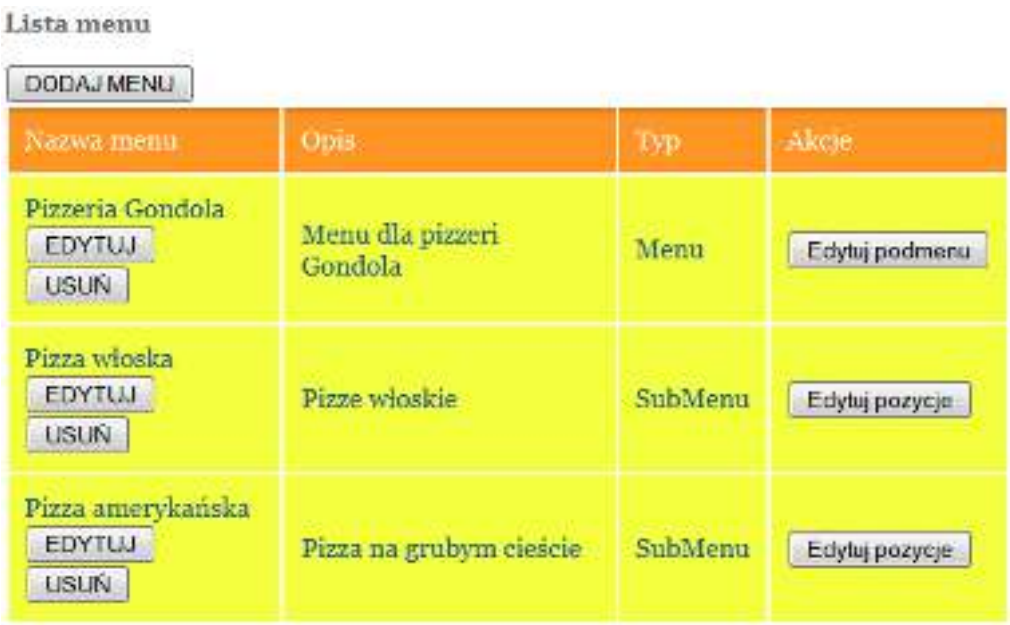
Jeżeli wprowadziliśmy poprawne dane, po kliknięciu w przycisk *DODAJ* otrzymamy komunikat:

Nowa restauracja jest już na liście:

Nazwa restauracji	Latitude	Longitude	Zasięg maksymalny	Darmowy zasięg	Cena za km	Menu
Da Grasso EDYTUJ USUŃ	49.612962	20.713707	20.0	10.0	2.0	brak
Pizzeria Gondola EDYTUJ USUŃ	49.611655	20.688794	15.0	10.0	2.0	Pizzeria Gondola

Jak widać restauracja nie ma jeszcze przypisanego menu, a jak wiadomo żadna restauracja nie może bez niego funkcjonować - tym bardziej restauracja realizująca zamówienia online.

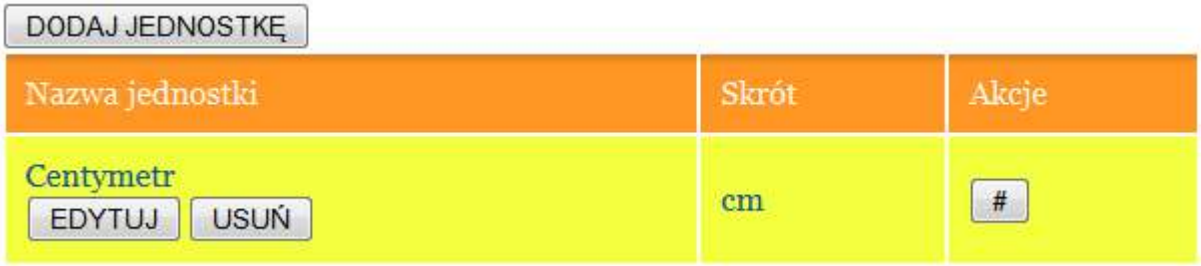
Aby utworzyć kompletne menu, najpierw przechodzimy do sekcji *Menu* dostępnej z menu:



Kolumna typ może przybrać jedną z dwóch wartości:

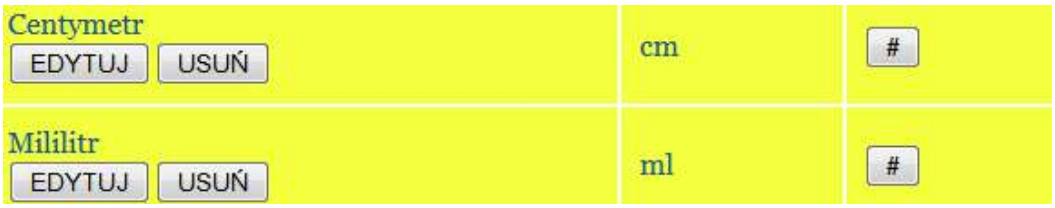
- SubMenu - dział menu np.: Pizza, Napoje, Desery
- Menu - karta dań składająca się z działów (SubMenu)

Załóżmy że SubMenu "Pizza włoska" oraz "Pizza amerykańska" wykorzystamy ponownie w naszej nowej karcie dań. Nowa restauracja oferuje jednak także napoje i ten dział menu musimy dodać. Musimy mieć na uwadze iż rozmiary napoi, wyrażają się w innych jednostkach niż rozmiary dań typu pizza. Najpierw musimy zdefiniować sobie nową jednostkę przechodząc do działu *Jednostki,* w którym na razie zdefiniowano jedynie jednostkę "centymetr", potrzebną do określenia rozmiaru pizzy:



Klikamy w przycisk *DODAJ JEDNOSTKĘ,* wypełniamy formularz i klikamy *DODAJ:*

Teraz lista jednostek wygląda tak:



Możemy teraz dodać nowe podmenu klikając *DODAJ MENU* w sekcji *Menu*:

Nazwa menu i *Opis* to pola których wartości wyświetlane będą w aplikacji mobilnej.

Rodzaj decyduje o tym czy dodawany przez nas element to Menu czy Podmenu (SubMenu).

W tym przypadku zaznaczona powinna być opcja *Podmenu.*

Teraz potrzebujemy kilku pozycji do rozdziału “Napoje” w naszej karcie dań.

Aby przypisać je do podmenu, musimy je wcześniej zdefiniować w sekcji *Pozycje Menu:*

Aby dodać nową pozycję menu, analogicznie jak w poprzednich przykładach klikamy w przycisk *DODAJ POZYCJĘ MENU*. Następnie wypełniamy następujący formularz:

Nazwa pozycji menu i *Opis* to teksty identyfikujące daną pozycję menu na urządzeniu mobilnym. Dodatkowo należy wybrać z dysku twardego grafikę która będzie wyświetlać się na urządzeniu mobilnym jako *Miniatura,* przedstawiająca danie/pozycję menu.

Dodamy jeszcze pozycję sok pomarańczowy. Dodane napoje są już na liście:

Nazwa pozycji menu	Opis	Miniatura	Akcje
Kola EDYTUJ USUŃ	Czarna woda		Rozmiary
Sok pomarańczowy EDYTUJ USUŃ	Ze świeżych truskawek		Rozmiary

Teraz należy dodać dostępne rozmiary poszczególnych artykułów. Klikamy przycisk *Rozmiary,* a na widoku na który zostaniemy przeniesieni wybieramy *DODAJ ROZMIAR:*

Rozmiary dla: Sok pomarańczowy

DODAJ ROZMIAR

Rozmiar:	Jednostka:	Cena:	Uwagi:	Akcje

Dodaj rozmiar

Rozmiar: 200

Jednostka: Mililitr

Cena (zł): 4

Uwagi:

DODAJ ANULUJ

Dodam jeszcze jeden rozmiar (500ml) i oto lista dostępnych rozmiarów soku pomarańczowego:

Rozmiar:	Jednostka:	Cena:	Uwagi:	Akcje
200.0 USUŃ	Mililitr	4.0		#
500.0 USUŃ	Mililitr	6.0		#

Analogicznie dodałem 2 rozmiary dla artykułu "Kola":

Rozmiary dla: Kola

DODAJ ROZMIAR

Rozmiar:	Jednostka:	Cena:
500.0 USUŃ	Mililitr	5.0
1000.0 USUŃ	Mililitr	9.0

Teraz należy wrócić do sekcji *Menu* i przy pozycji podmenu "Napoje" i kliknąć w przycisk *Edytuj pozycje,* aby dodać przed momentem utworzone pozycje menu (przycisk *DODAJ POZYCJE*):

Pozycje menu dla: Napoje

Dodaj pozycję menu

Nazwa pozycji: Sok pomarańczowy

Wybierz pozycję menu z listy
Kola
Pizza Frutti di Mare
Pizza Giuseppe
Pizza Margherita
Pizza Pomodorra
Pizza napoli
Pizza prosciutto
Sok pomarańczowy

DODAJ ANULUJ

Kompletna lista wygląda następująco:

Nazwa pozycji	Opis
Sok pomarańczowy USUŃ	Ze świeżych truskawek
Kola USUŃ	Czarna woda

Możemy już z gotowych elementów skomponować kompletną kartę dań. Wchodzimy ponownie do sekcji *Menu* i wybieramy *DODAJ MENU*:

Nazwa menu: Pizzeria Da Grasso

Opis: Karta dań Da Grasso

Rodzaj: ○ Podmenu ◉ Menu główne

Następnie, w wierszu z nowo dodanym menu, wybieramy *Edytuj podmenu* i postępujemy analogicznie jak w przypadku uzupełniania pozycji menu w podmenu, aż uzyskamy następującą listę podmenu dla menu "Pizzeria Da Grasso":

Lista podmenu dla menu: Pizzeria Da Grasso

DODAJ PODMENU

Nazwa podmenu	Opis	Akcje
Napoje USUŃ	Napoje gazowane i soki	#
Pizza amerykańska USUŃ	Pizza na grubym cieście	#
Pizza włoska USUŃ	Pizze włoskie	#

Mamy już gotowe menu, które teraz możemy przypisać naszej nowej restauracji. Wchodzimy ponownie na listę restauracji i edytujemy interesującą nas pozycję:

Klikamy zapisz - nasza nowa restauracja dysponuje już kartą dań:

Da Grasso EDYTUJ USUŃ	49.612962	20.713707	20.0	10.0	2.0	Pizzeria Da Grasso

Urządzenie mobilne może już pobierać informacje na temat naszej nowej restauracji.

Służą do tego następujące URL zasobów:

- /restaurant/{restaurantId} - zwraca restaurację o podanym Id
- /restaurants - zwraca wszystkie restauracje istniejące w systemie
- /menu/{menuId} - zwraca menu o podanym Id
- /menus - zwraca wszystkie menu istniejące w systemie
- /subMenu/{subMenuId} - zwraca subMenu o podanym Id
- /subMenus - zwraca wszystkie subMenu istniejące w systemie
- /menuItem/{menuItemId} - zwraca element menu o podanym Id
- /sizes/{menuItemId} - zwraca wszystkie rozmiary elementu menu o podanym Id

Przykładowy scenariusz wywołań wykorzystania poszczególnych zasobów przez urządzenie mobilne może być następujący:

1. Użytkownik chce otrzymać listę wszystkich restauracji i wybrać jedną w której zamówi jakieś danie na dowóz (urządzenie dysponujące GPS może automatycznie wybrać najbliższą restaurację, na tym etapie użytkownik może jeszcze nie chcieć udostępniać serwerowi swojej lokalizacji, lub używać aplikacji po raz pierwszy). Urządzenie pobiera listę restauracji za pomocą *instantpizza1.appspot.com/ws/restaurants* i otrzymuje w odpowiedzi:


```
[
    {
       "id": "ag9zfmluc3RhbnRwaXp6YTFyEgsSClJlc3RhdXJhbnQYqroEDA",
       "name": "Da Grasso",
       "longitude": 20.713707,
       "latitude": 49.612962,
       "maxRadius": 20,
       "freeRadius": 10,
       "kmPrice": 2,
       "menuId": "ag9zfmluc3RhbnRwaXp6YTFyDAsSBE1lbnUYq7oEDA"
    },
    {
       "id": "ag9zfmluc3RhbnRwaXp6YTFyEgsSClJlc3RhdXJhbnQY8qIEDA",
       "name": "Pizzeria Gondola",
       "longitude": 20.688794,
       "latitude": 49.611655,
       "maxRadius": 15,
       "freeRadius": 10,
       "kmPrice": 2,
       "menuId": "ag9zfmluc3RhbnRwaXp6YTFyDAsSBE1lbnUY04MEDA"
    }
]
```


2. Teraz użytkownik chce na urządzeniu mobilnym obejrzeć menu restauracji “Da Grasso” - urządzenie zna *menuId* dla tej restauracji więc może pobrać to menu korzystając z adresu *instantpizza1.appspot.com/ws/menu/*`ag9zfmluc3RhbnRwaXp6YTFyDAsSBE1lbnUYq7oEDA`

```
{
  "name": "Pizzeria Da Grasso",
  "description": "Karta dań Da Grasso ",
  "subMenusKeys": [
    "ag9zfmluc3RhbnRwaXp6YTFyDwsSB1N1Yk1lbnUYkcIEDA",
    "ag9zfmluc3RhbnRwaXp6YTFyDwsSB1N1Yk1lbnUYoZMEDA",
    "ag9zfmluc3RhbnRwaXp6YTFyDwsSB1N1Yk1lbnUYuYsEDA"
  ],
  "id": "ag9zfmluc3RhbnRwaXp6YTFyDAsSBE1lbnUYq7oEDA"
}
```


3. Użytkownik wybiera kolejno interesujące go działy karty dań. Pobranie informacji o SubMenu z napojami odbywa się przy użyciu adresu *instantpizza1.appspot.com/ws/subMenu/ag9zfmluc3RhbnRwaXp6YTFyDwsSB1N1Yk1lbnUYkcIEDA* i w efekcie daje następujący obiekt JSON:


```
{
  "name": "Napoje",
  "description": "Napoje gazowane i soki",
  "menuItemsKeys": [
      "ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGMKyBAw",
      "ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGPOiBAw"
  ],
  "id": "ag9zfmluc3RhbnRwaXp6YTFyDwsSB1N1Yk1lbnUYkcIEDA"
}
```


4. Po wybraniu działu, urządzenie powinno wyświetlić użytkowniki dane poszczególych pozycji.

   Pobranie informacji o konkretnej pozycji odbywa się poprzez:
   instantpizz1.appspot.com/ws/menuItem/`ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGMKyBAw`

   Powyższy adres URL w odpowiedzi zwraca:


```
{
  "name": "Sok pomarańczowy",
  "description": "Ze świeżych truskawek",
```

```
  "id": "ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGMKyBAw",
  "thumbnailUrl":
"/img?blob-key=AMIfv97yyXGRWk5fxx62NEj_iN8gneJ0wae-uvOEfphuQaTDhF8-mn
P4wtpe9e07QLbH-Gt6ezHIumrej6wxP7WiM19zrf5zzokSffZXh7JTiP-P9V23E5zm8G0
DI1OVIvfJftZMLpeRtzd284Ul0zz3mKB1X70doTYF01fwbCB569ZjV7FeQH8"
}
```


Obrazek przedstawiający produkt może być pobrany z adresu:
instantpizza1.appspot.com/img?blob-key=AMIfv97yyXGRWk5fxx62NEj_iN8gneJ0wae-uvOEfphuQaTDhF8-mnP4wtpe9e07QLbH-Gt6ezHIumrej6wxP7WiM19zrf5zzokSffZXh7JTiP-P9V23E5zm8G0DI1OVIvfJftZMLpeRtzd284Ul0zz3mKB1X70doTYF01fwbCB569ZjV7FeQH8

5. Jeżeli klient zdecyduje się na dany produkt, musi wybrać jeszcze rozmiar porcji. Lista dostępnych rozmiarów pobierana jest przez aplikacjie z adresu:
instantpizza1.appspot.com/ws/sizes/ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGMKyBAw

   Efekt:


```
[
{
  "description": "",
  "value": 200,
  "price": 4,
  "points": 0,
  "active": true,
  "unitSignature": "ml",
  "unitName": "Mililitr",
  "id":
"ag9zfmluc3RhbnRwaXp6YTFyGgsSCE1lbnVJdGVtGMKyBAwLEgRTaXplGAEM",
  "menuItemId": "ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGMKyBAw"
},
{
  "description": "",
```

```
    "value": 500,
    "price": 6,
    "points": 0,
    "active": true,
    "unitSignature": "ml",
    "unitName": "Mililitr",
    "id":
"ag9zfmluc3RhbnRwaXp6YTFyGwsSCE1lbnVJdGVtGMKyBAwLEgRTaXplGOkHDA",
    "menuItemId": "ag9zfmluc3RhbnRwaXp6YTFyEAsSCE1lbnVJdGVtGMKyBAw"
  }
  ]
```


6. Dodanie produktu do koszyka.

Składanie zamówienia

Zamówienia obsługiwane są przez następujące zasoby:

- /order" - składanie zamówienia
- /order/{clientEmailString} - pobieranie listy zamówień użytkownika

Jeżeli użytkownik chce zamówić 2 sztuki 200ml soku pomarańczowego w Da Grasso, a znajduje się obecnie w miejscowości Łabowa, obiekt zamówienia wysłany przez urządzenie mobilne do serwera powinien mieć postać:


```
{
    "longitude": 20.855457,
    "latitude": 49.530445,
    "clientEmail": "adam@gmail.com",
    "phoneNumber": "555666444",
    "restaurantKeyStringValue":
      "ag9zfmluc3RhbnRwaXp6YTFyEgsSClJlc3RhdXJhbnQYqroEDA",
    "orderItems": [
        {
            "menuItemSizeKeyStringValue":
```

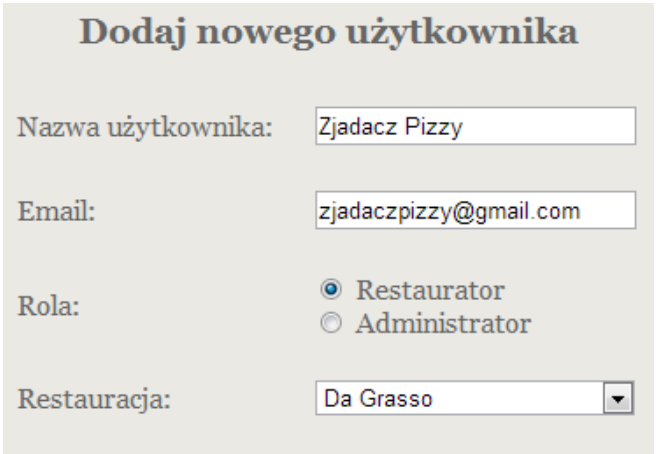
```
            "ag9zfmluc3RhbnRwaXp6YTFyGgsSCE1lbnVJdGVtGMKyBAwLEgRTaXplGAEM",
            "quantity": 2
        }
    ]
}
```

Obiekt ten należy wysłać na adres *instantpizza1.appspot.com/ws/order*

Na komputerze urządzenie mobilne można zasymulować korzystając z biblioteki cURL:

```
curl -i -H "Content-type:application/json" -X PUT --data
{\"longitude\":20.855457,\"latitude\":49.530445,\"clientEmail\":\"a
dam@gmail.com\",\"phoneNumber\":\"111666444\",\"restaurantKeyString
Value\":\"ag9zfmluc3RhbnRwaXp6YTFyEgsSClJlc3RhdXJhbnQYqroEDA\",\"or
derItems\":[{\"menuItemSizeKeyStringValue\":\"ag9zfmluc3RhbnRwaXp6Y
TFyGgsSCE1lbnVJdGVtGMKyBAwLEgRTaXplGAEM\",\"quantity\":2}]}
instantpizza1.appspot.com/ws/order
```

Zamówienie zostało złożone. Teraz pracownik restauracji może je odebrać i przekazać do realizacji. Najpierw musimy jednak założyć pracownikowi konto restauratora z przypisaną odpowiednią restauracją (w tym przypadku Da Grasso). Przechodzimy do sekcji *Użytkownicy* w panelu aplikacji, dodajemy nowego użytkownika:



Widnieje on już na liście użytkowników jako restaurator. Jeżeli się nie spisze, w każdej chwili możemy go zablokować zaznaczając pole “Zablokowany”:

Nazwa użytkownika	Email	Rola	Restauracja	Akcje
Albert EDYTUJ USUŃ	awajdans@gmail.com	ADMIN		Zablokowany
Zjadacz Pizzy EDYTUJ USUŃ	zjadaczpizzy@gmail.com	RESTAURATEUR	Da Grasso	Zablokowany

Zalogujmy się teraz do panelu używając konta “Zjadacz Pizzy”:

Logując się jako restaurator mamy dostęp wyłącznie do sekcji *Zamówienia,* z listą zamówień wyłącznie dla restauracji przypisanej restauratorowi przez administratora.

Na liście widzimy złożone wcześniej zamówienie. Klikając w wiersz tabeli z konkretnym zamówieniem możemy wyświetlić jego szczegóły:

adam@gmail.com	09-07-2013	14.0	111666444	PLACED

Pozycja menu	Rozmiar	Sztuk
Sok pomarańczowy	200.0 ml	2

Aplikacja wyróżnia 4 rodzaje statusów:

- *PLACED* - zamówienie złożone
- *SENDED* - zamówione produkty w doręczeniu
- *SERVED* - produkty doręczone
- *APPROVED* - pozytywnie ocenione przez użytkownika (funkcjonalność tę być może uda się zaimplementować w przyszłości)

Zakodowane są jako:

public enum Status {PLACED, SENDED, SERVED, APPROVED};

Kwota zamówienia to cena za 2 sztuki soku pomarańczowego (2 * 4zł = 8zł), oraz dopłata za dowóz.

Do Łabowej z Da Grasso jest około 13.73271km, a darmowy dowóz Da Grasso oferuje maksymalnie do 10km. Aplikacja odległość zaokrągla w dół (podłoga z wartości), więc nadmiarowe mamy 3km i cenę za kilometr na poziomie 2zł (3 * 2 = 6). W sumie dowóz (6zł) i zamówione artykuły są warte 14zł. Wszystko się zgadza. Odległość w aplikacji to odległość w linii prostej i wyliczana jest w następujący sposób:

```
public static float calculateDistance(double lat1, double lng1,
double lat2, double lng2) {
      double earthRadius = 3958.75;
      double dLat = Math.toRadians(lat2-lat1);
      double dLng = Math.toRadians(lng2-lng1);
```

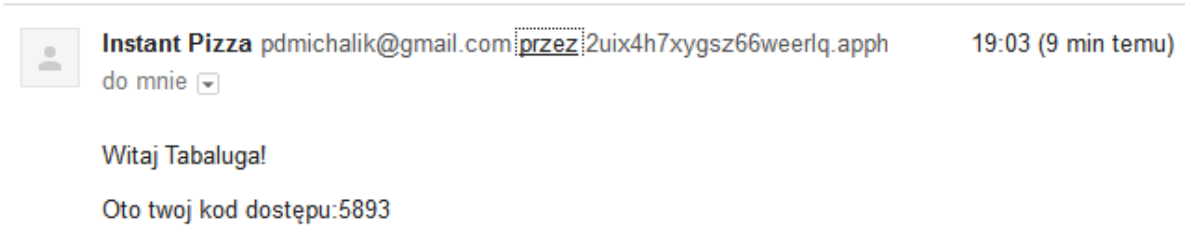
```
      double a = Math.sin(dLat/2) * Math.sin(dLat/2) +
                      Math.cos(Math.toRadians(lat1)) *
Math.cos(Math.toRadians(lat2)) *
                      Math.sin(dLng/2) * Math.sin(dLng/2);
      double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
      double dist = earthRadius * c;
      int meterConversion = 1609;
      return new Float(dist * meterConversion).floatValue();
}
```

W szczegółach zamówienia, oprócz listy pozycji zamówienia, jest także obrazek z kodem QR (dwuwymiarowy kod kreskowy), przechowujący adres id zamówienia. Urządzenia mobilne wykorzystując wbudowane kamery mogą z łatwością przy użyciu gotowych bibliotek zdekodować kod QR i odczytać id zamówienia. Może to stanowić duże ułatwienie, dla dostawców pizzy korzystających ze specjalnej aplikacji automatycznie na odczytującej kod i wysyłającej do serwera informacje iż zamówienie jest w doręczeniu.

Dostawa

Najpierw musimy założyć nowe konto dostawcy. Przechodzimy do sekcji *Dostawcy* w panelu aplikacji i dodajemy nowego dostawce w sposób analogiczny jak w przypadku nowego restauratora:

Automatycznie dostawcy przypisywane jest aktywne przez jeden dzień hasło. Przesyłane jest ono na adres email dostawcy:



Hasło to zawsze liczba pseudolosowa wygenerowana w następujący sposób:

```
java.util.Random r = new Random();
Integer.toString((r.nextInt()+1000) % 10000);
```

Mail z hasłem jest wysyłany za pośrednictwem następująco zdefiniowanej funkcji:

```
public static void sendPassword(Supplier s){
      Properties props = new Properties();
      Session session = Session.getDefaultInstance(props, null);
      String htmlBody = "<p>Witaj "+s.getName()+"!</p>"+
"<p>Oto twoj kod dostępu:" + s.getPassword() + "</p>";
      try {
            Message msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress("pdmichalik@gmail.com",
"Instant Pizza"));
            msg.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(s.getEmail(),
s.getName()));
            msg.setSubject("Nowy kod dostępu.");
            Multipart mp = new MimeMultipart();
            MimeBodyPart htmlPart = new MimeBodyPart();
            htmlPart.setContent(htmlBody, "text/html");
            mp.addBodyPart(htmlPart);
```

```
            msg.setContent(mp);
            Transport.send(msg);
        } catch (Exception e) { /* ... */ }
}
```

Do wykonania powyższego kodu wymagana jest biblioteka JavaMail API (javax.mail), której dokumentacja dostępna jest na stronie:
http://docs.oracle.com/javaee/5/api/javax/mail/package-summary.html

Dodatkowo codziennie, o godzinie 3:00, hasła wszystkich dostawców są resetowane i ponownie przesyłane na adresy email.
Odpowiedzialny za to jest wpis w *cron.xml*. Jest to plik w którym możemy zdefiniować harmonogram operacji, które mają być wykonywane w określonym czasie, lub co pewien interwał.

```
<cron>
    <url>/ws/resetPasswords</url>
    <description>Reset passwords every day at 3
o'clock</description>
    <schedule>every day 03:00</schedule>
</cron>
```

Węzeł <*url*> przechowuje adres URL który otwierany jest w ustalonym czasie - tym wypadku WebService *resetPasswords* odpowiedzialny za aktualizację haseł i wysłanie maili do dostawców:

<*description*> to po prostu opis ułatwiający identyfikację poszczególnych zadań.

<*schedule*> to węzeł w którym określamy kiedy akcja ma zostać wykonana. Kilka przykładów ustawień ze strony developers.google.com:

- *every 12 hours* - co 12 godzin
- *every 5 minutes from 10:00 to 14:00* - co 5 minut, pomiędzy godziną 10:00 a 14:00
- *every monday 09:00* - w każdy poniedziałek o 9:00
- *1st monday of sep,oct,nov 17:00* - w każdy pierwszy poniedziałek września, października i listopada o godzinie 17:00

W naszym wypadku wartość *every day 03:00* oznacza “codziennie o godzinie 3:00”.

Klasa odpowiedzialna zmianę haseł:

```
public class ResetPasswordsResource extends ServerResource{
      @Get("JSON")
      public String retrieve() {
            List<Supplier> list = SuppliersDAO.getSuppliers();
            Random r = new Random();
            for(Supplier s : list){
                  s.setPassword(Integer.toString((r.nextInt()+1000) %
10000));
                  SuppliersDAO.save(s);
                  MailManager.sendPassword(s);
            }
            return "Wykonano.";
      }
}
```

Dodatkowo aby uniemożliwić wykonanie procedury osobie postronnej, znającej URL zasobu, do pliku *web.xml* należy dodać wpis który informuje serwer że dostęp do linku z zewnątrz systemu mogą uzyskać tylko zalogowani użytkownicy:

```
<security-constraint>
        <web-resource-collection>
            <url-pattern>/ws/resetPassword</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
</security-constraint>
```

Wracając do obsługi doręczenia zamówienia - dedykowana dla dostawcy aplikacji po odczytaniu z kodu QR identyfikatora zamówienia, wysyła do serwera żądanie powiązania dostawcy z zamówieniem na adres URL: *instantpizza1.appspot.com/ws/link*

Zasób przyjmuje obiekt w JAVA zdefiniowany jako:

```
private static class SupplierOrderConnection{
      private String supplierEmail;
      private String orderId;
      private String pass;
      //...
}
```

Obiekt do przesłania w JSON zdefiniowany powinien mieć formę jak w poniższej komendzie cURL (dla linii komend Windows):

```
curl -i -H "Content-type:application/json" -X PUT --data
{\"supplierEmail\":\"dostawcapizzy1@gmail.com\",\"orderId\":\"ag9zf
mluc3RhbnRwaXp6YTFyDQsSBU9yZGVyGLuLBAw\",\"pass\":\"5893\"}
instantpizza1.appspot.com/ws/link
```

Jeżeli podane hasło "jednodniowe" było poprawne, dostawca istniał w bazie i nie był zablokowany, do zamówienia i podanym nr id dopisywany jest adres email dostawcy, a status zmieniany jest na SENDED (czyli "w trakcie doręczenia"):

status	supplierEmail
SENDED	dostawcapizzy1@gmail.com

W przeciwnym wypadku zwracany jest kod błędu, informujący o braku praw do wykonania operacji. Za realizację tej funkcjonalności odpowiedzialny jest następujący fragment kodu *ServerResource*:

```
if(s.getPassword().compareTo(soc.getPass())!=0||s.isBanned()||
!s.isActive())
{
      setStatus(org.restlet.data.Status.CLIENT_ERROR_UNAUTHORIZED);
      return;
}
```

W analizowanym przypadku operacja powiodła się i dostawca został pomyślnie powiązany z zamówienie.

Aby otrzymać potrzebne dane, aplikacja dedykowana dla dostawcy może pobrać listę obsługiwanych zamówień za pomocą zasobu *link* z parametrem *supplierEmailString*:

http://instantpizza1.appspot.com/ws/link/dostawcapizzy1@gmail.com

Powyższy URL, w oparciu o metodę GET, zwróci kolekcję obiektów (zakodowanych w JSON) typu Order, z polami *clientEmail*, oraz *date* ustawionymi na *null*. Zabieg ten podyktowany jest względami bezpieczeństwa - o tym więcej nieco później przy opisie potwierdzenia doręczenia.

System umożliwia śledzenie obecnego położenia dostawcy. Aby było to możliwe, aplikacja przeznaczona dla dostawcy musi w regularnych odstępach czasu, lub po pokonaniu umownego dystansu, przesyłać do serwera aktualne położenie. Odbywa się to za pośrednictwem metody POST dla zasobu */supplierPosition* . Obiekt JSON powinien mieć następującą postać:

```
{"email":"dostawcapizzy1@gmail.com","latitude":"49.550719","longitu
de":"20.767567"}
```

```
C:\Users\Przemek>curl -i -H "Content-type:application/json" -X PUT --data {\"ema
il\":\"dostawcapizzy1@gmail.com\",\"latitude\":\"49.550719\",\"longitude\":\"20.
767567\"}   instantpizza1.appspot.com/ws/supplierPosition
HTTP/1.1 200 OK
Accept-Ranges: bytes
Date: Wed, 10 Jul 2013 17:38:29 GMT
Content-Type: text/html
Server: Google Frontend
Content-Length: 0
```

Aktualną pozycję danego dostawcy można pobrać w każdej chwili z serwera, dodając jednak parametr na końcu URL zasobu: */supplierPosition/{email}* .

{email} to adres email dostawcy którego położenie chcemy sprawdzić. Sprawdźmy zaktualizowane przed momentem położenie dostawcy *dostawcapizzy1@gmail.com*:

```
C:\Users\Przemek>curl -i -H "Content-type:application/json" -X GET instantpizza1
.appspot.com/ws/supplierPosition/dostawcapizzy1@gmail.com
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Accept-Ranges: bytes
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Date: Wed, 10 Jul 2013 17:53:00 GMT
Server: Google Frontend
Cache-Control: private
Transfer-Encoding: chunked

{"email":"dostawcapizzy1@gmail.com","latitude":49.550719,"longitude":20.767567}
```

Współrzędne pobrane są identyczne jak te przesłane na serwer. Zasób ten może być wykorzystywany przez aplikację dla klienta, do monitorowania odległości pozostałej do przejechania przez dostawcę.

Potwierdzenie dostarczenia

Ostatnią czynnością zamykającą realizację zamówienia jest potwierdzenie dostarczenia przesyłki do klienta. Schemat realizacji tej czynności na poziomie aplikacji klienta i dostawcy (na urządzeniach mobilnych) jest następujący:

1. Klient udostępnia w postaci kodu QR swój email oraz datę (czas) przyjęcia zamówienia przez serwer w postaci liczby całkowitej *long* (data w JAVA przechowywana jest w pamięci jako ilość milisekund jaka upłynęła od 01 stycznia 1970[1]).
2. Dostawca skanuje kod QR i za pośrednictwem aplikacji przesyła do serwera potwierdzenie dostarczenia

Przesłanie potwierdzenia odbywa się za pośrednictwem metody PUT zasobu */delivered.*
Należy przesłać obiekt klasy o następującej strukturze:

```
class Delivered{
     String clientEmail;
     String orderId;
     String date;
}
```

[1] http://docs.oracle.com/javase/6/docs/api/java/util/Date.html

Komenda cURL wraz z dołączonymi danymi reprezentującymi obiekt zakodowany w JSON:

```
curl -i -H "Content-type:application/json" -X PUT --data
{\"clientEmail\":\"adam@gmail.com\",\"date\":\"1373384319785\",\"or
derId\":\"ag9zfmluc3RhbnRwaXp6YTFyDQsSBU9yZGVyGLuLBAw\"}
instantpizza1.appspot.com/ws/delivered
```

Po przyjęciu żądania przez serwer dane poddawane są analizie zgodności - jeżeli przesłana data lub email klienta mają wartości różne od odczytanych z bazy danych serwer zwraca komunikat o błędzie, a zmiany nie zostają wprowadzone:

```
Order o = OrderDAO.getOrderById(KeyFactory.stringToKey(d.orderId));
if(d.clientEmail.compareTo(o.getClientEmail().getEmail())!=0 ||
   d.date.compareTo(Long.toString(o.getDate().getTime()))!=0 ){
      setStatus(org.restlet.data.Status.CLIENT_ERROR_UNAUTHORIZED);
      return;
}
o.setStatus(Status.SERVED);
```

Jeżeli autoryzacja przebiegła pomyślnie, w panelu możemy teraz odczytać nowy status zamówienia:

Email klienta	Data złożenia	Kwota zamówienia	Nr telefonu	Status
adam@gmail.com	09-07-2013	6.0	111666444	SERVED

Zamówione produkty zostały dostarczone do klienta - cel biznesowy został osiągnięty.
Dalsze funkcjonowanie restauracji opierać się będzie na podobnym schemacie - zgodnie z zamysłem właściciela restauracji. System jest bardzo elastyczny i dzięki rozbudowanym możliwościom konfiguracji, pozwala właścicielowi restauracji na dostosowanie go do indywidualnych potrzeb.

4. PODSUMOWANIE

Cel niniejszej pracy dyplomowej, czyli stworzenie serwera aplikacji mobilnej, obsługującego zamówienia online udało się w pełni zrealizować. Końcowa implementacja została przetestowana pod kątem komunikacji z aplikacją mobilną (wykonaną w ramach osobnej pracy dyplomowej) w realnych warunkach, w oparciu o kilka przykładowych scenariuszy biznesowych. Proces projektowania i tworzenia aplikacji współpracującej z aplikacją opartą o odrębną architekturę, umożliwił nam znaczne poszerzenie swojej wiedzy technicznej jak również umiejętności współpracy i realizacji wspólnych założeń.

Zdobyte doświadczenie z całą pewnością zaowocuje podczas prac nad kolejnym projektem. Sądzę, że stworzony w ramach tej pracy inżynierskiej system, mógłby znaleźć realne zastosowanie w wielu firmach gastronomicznych, jako moduł obsługi zamówień. Aby zapewnić kompleksową obsługę, należałoby system rozszerzyć o moduł sprzedaży detalicznej, a także zmodyfikować sposób przechowywania danych o produktach pod kątem dokumentów sprzedaży.

Składanie zamówień drogą elektroniczną, w dobie powszechnie dostępnego internetu i urządzeń mobilnych o dużej mocy obliczeniowej i szerokiej gamie zastosowań, może w ciągu najbliższych lat znacznie zwiększyć skalę. Sądzę że warto poświęcić nieco więcej czasu na zgłębianie tajników tej i pokrewnych tematyk, oscylujących wokół technologii mobilnych.

BIBLIOGRAFIA

1. http://www.developers.google.com/appengine/
2. http://restlet.org/learn/tutorial/2.1/
3. http://db.apache.org/jdo/
4. M. Jarzębski, Wprowadzenie do MVC, http://www.php.pl
5. http://docs.oracle.com/javaee/6/tutorial/doc/