# JD

Programowanie zespołowe

Jakub Brodziński
229781

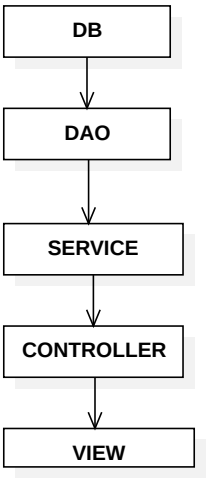
# 1 Wstęp

# 2 Analiza problemu

# 3 Projekt systemu

## 3.1 Architektura projektu

Cieżko jest mówić o naszej aplikacji jak o jednym bycie, ponieważ można ją podzielić na dwa podsystemu, gdzie w obu przypadkach archiktektura systemu jest wielowarstwowa. Jednym z podsystemów jest aplikacja webowa, która została zaprojektowana w oparciu o wzorzec **MVC** (ang. Model-View-Controller), który to narzucił wielowarstwową architektura. Oddzielając logikę biznesową od modelu (tj. danych) oraz interfejsu użytkownika aplikacji została zaprojektowana zgodnie z zasadami **GRASP** (ang. General responsibility assignment software patterns). Taka a nie inna architektura aplikacji webowej oprócz znacznego zwiększenia czytelności kodu pozwoliła na wielowarstwowe zabezpieczenia, które zostały nałożone na każdą z trzech głównych warstw naszej aplikacji. Samą część łącząca wartswe modelu oraz controller'a możemy podzielić na dwie podwarstwy: **DAO** (ang. Data Access Object) oraz **Service**.



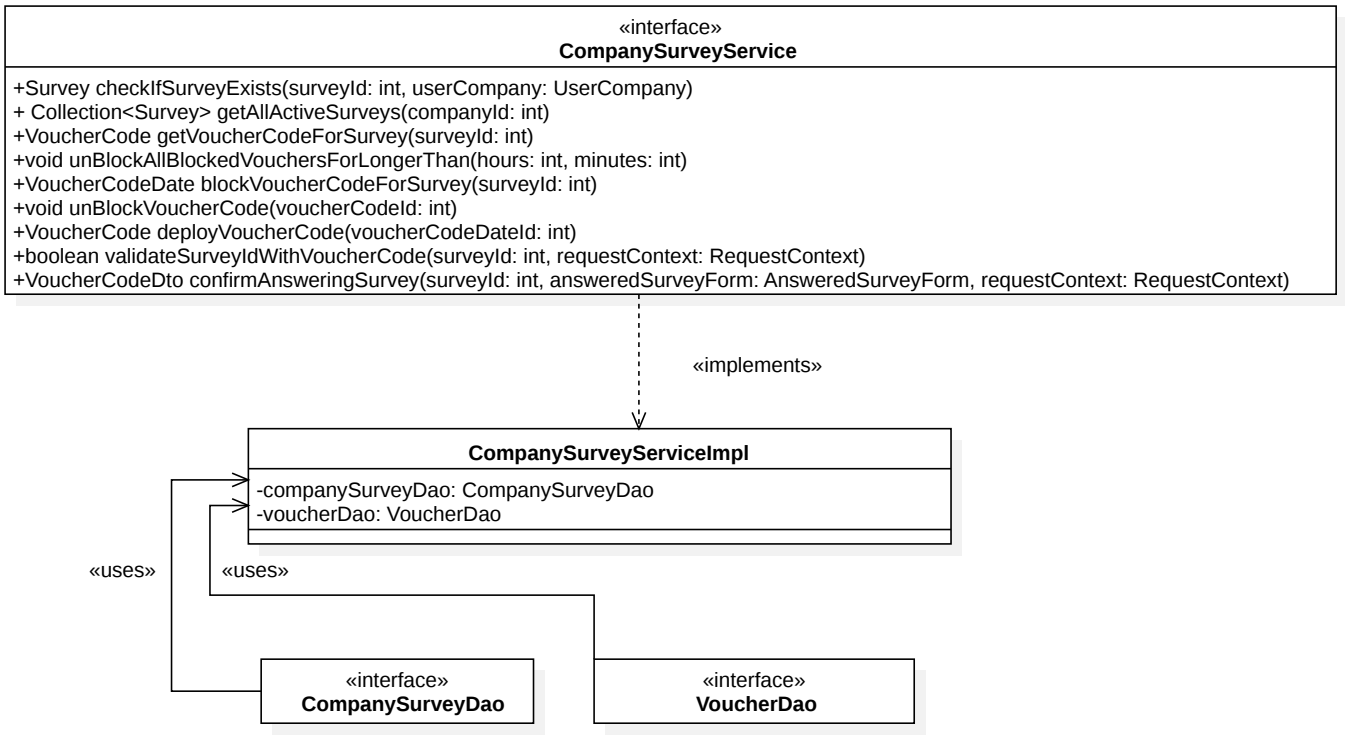

Rysunek 1: System podzielony na podwarstwy.

Warstwa **DAO** odpowiedzialna jest bezpośrednio na komunikacje z wartswą modelu, tj. bazą danych. Natomiast to w warstwie serwisowej, która korzysta z warstwy **DAO** została zaimplementowana cała logika biznesowa i to właśnie z warstwy serwisowej korzystamy w kontrolerach. Drugim podsystemem naszego projektu jest aplikacja webowa, która również jest systemem rozproszonym komunikującym się z serwerem (który jest częścią pierwszego podsystemu) wykorzystując **REST** (ang. Representational State Transfer) oferując użytkownikowi jedynie część funkcjonalności aplikacji webowej. W projekcie poza stosowaniem zasad **GRASP** zostały wykorzystane takie wzorce projektowe jak **Proxy**, **Template Method**, **Adapter**, **Factory** oraz **Decorator**, **Dependency Injection**, **Aspect Oriented Programing**, **Singleton**, które w sposób naturalny współgrały z użytymi przez nas technologiami.

## 3.2 Przypadki użycia i scenriusze

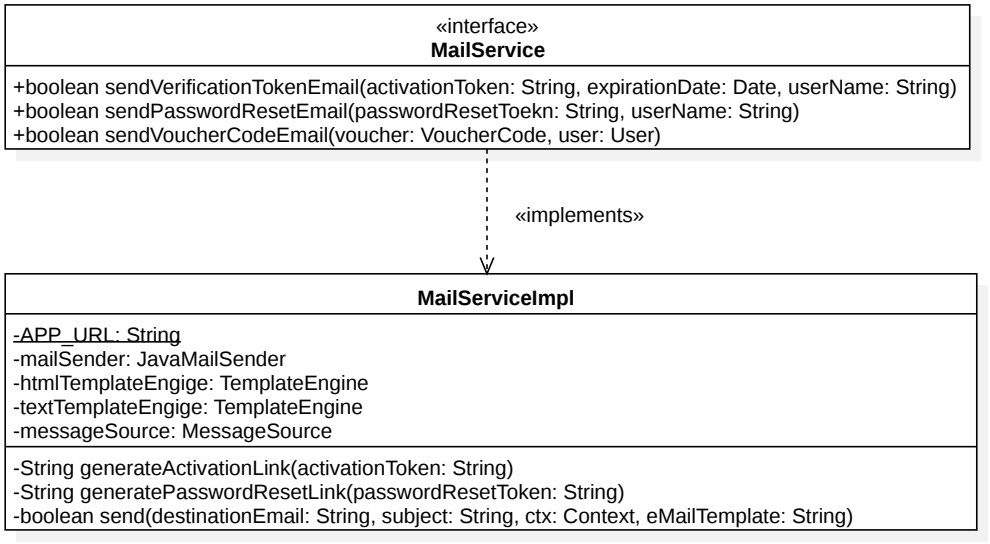
## 3.3 Diagramy klas

Diagramy przedstawione poniżej przedstawiają klasy serwisowe, w których to właśnie została zaimplementowana cała logika biznesowa naszego systemu i to one reprezentują funkcjonalność i możliwości naszej aplikacji webowej. W celu zwiększenia czytelności diagramów z ich większości zostały usunięte trywialne metody (nie zawierające logiki biznesowej), których zadaniem było dodanie, edycja, usunięcie przekazywanego obiektu w bazie danych.



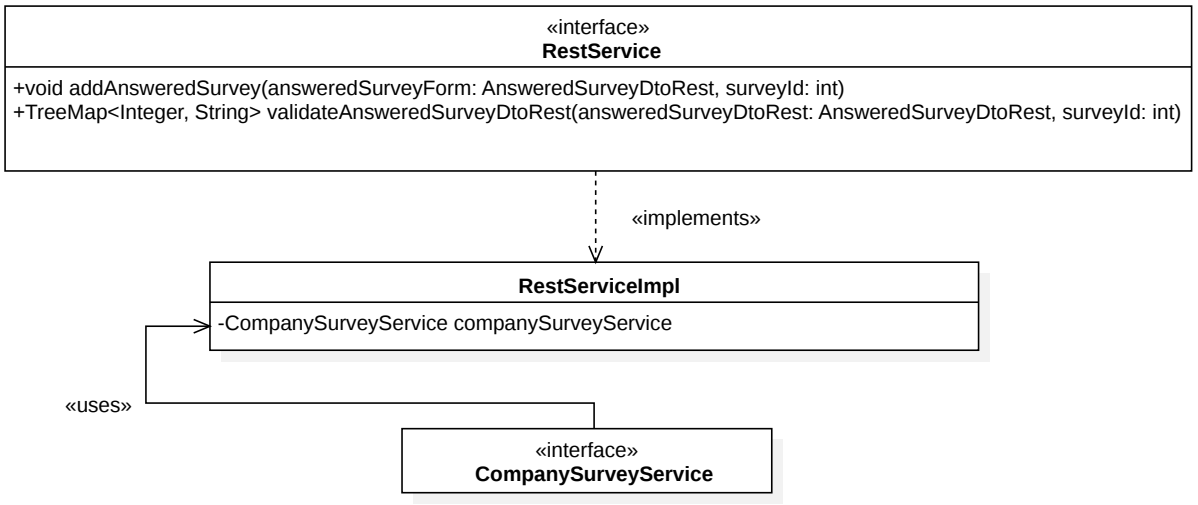

Rysunek 2: Diagram klas dla interfejsu serwisowego *CompanySurveyService*

Powyższy diagram przedstawia diagram klas dla serwisu, który przede wszystskim obsługuje funkcjonalność związana pobieraniem aktywnym, przeglądaniem oraz wypełnianiem ankiet jak również walidacją tych czynności. Ponadto odpowiedzialny jest za podjęcie decyzji czy dany kupon powininen zostać wydany czy też nie.



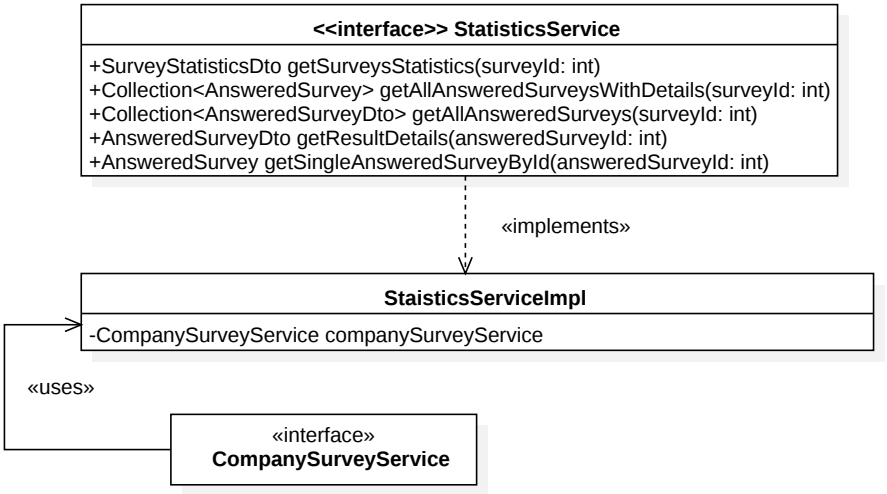

Rysunek 3: Diagram klas dla interfejsu serwisowego *MailService*

W **MailService** każda metoda odpowiada innemu typu wiadomości, która zostanie wysłąna.



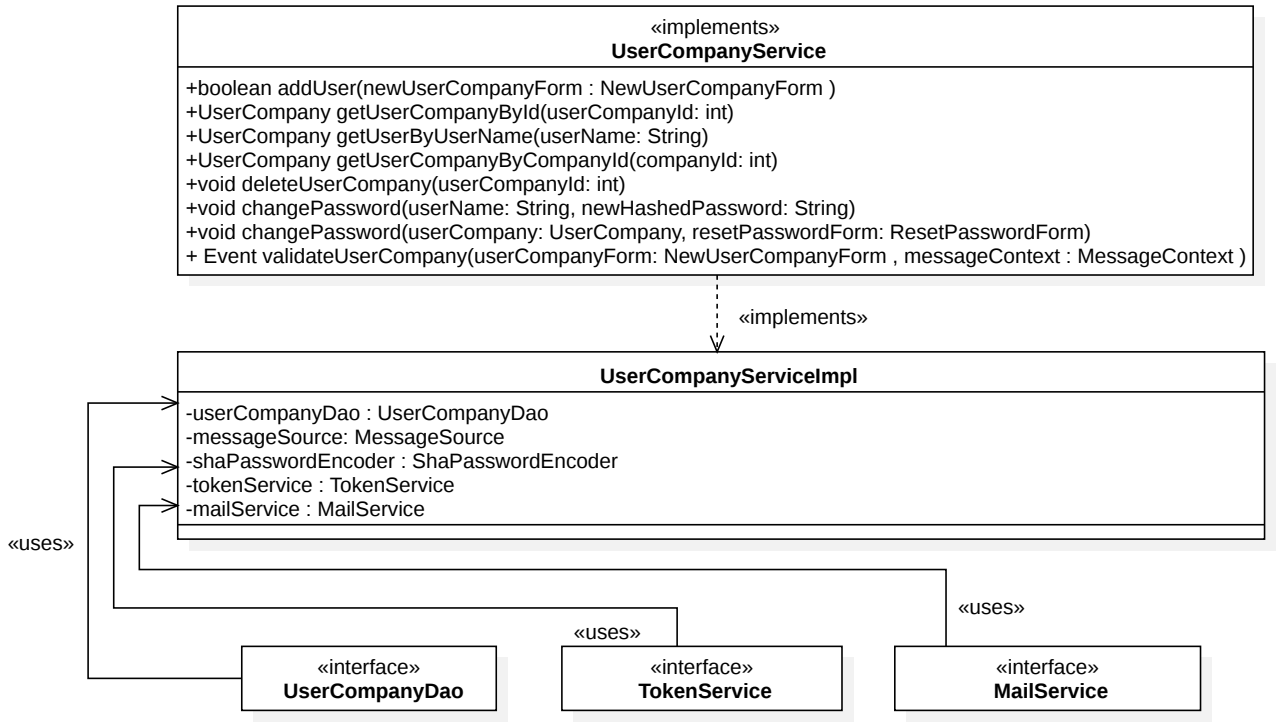

Rysunek 4: Diagram klas dla interfejsu serwisowego *RestService*

Powyższy diagram klas przedstawia diagram klas serwisowych obsługujących **REST Api**, które jest wykorzystywane przez aplikacje mobilna.




Rysunek 5: Diagram klas dla interfejsu serwisowego *StatisticsService*

W powyżej przedstawionych klasach serwisowych obliczane są statystyki, które każdy użytkownik ze stworzonymi ankietami może przeglądać.




Rysunek 6: Diagram klas dla interfejsu serwisowego $UserCompanyService$

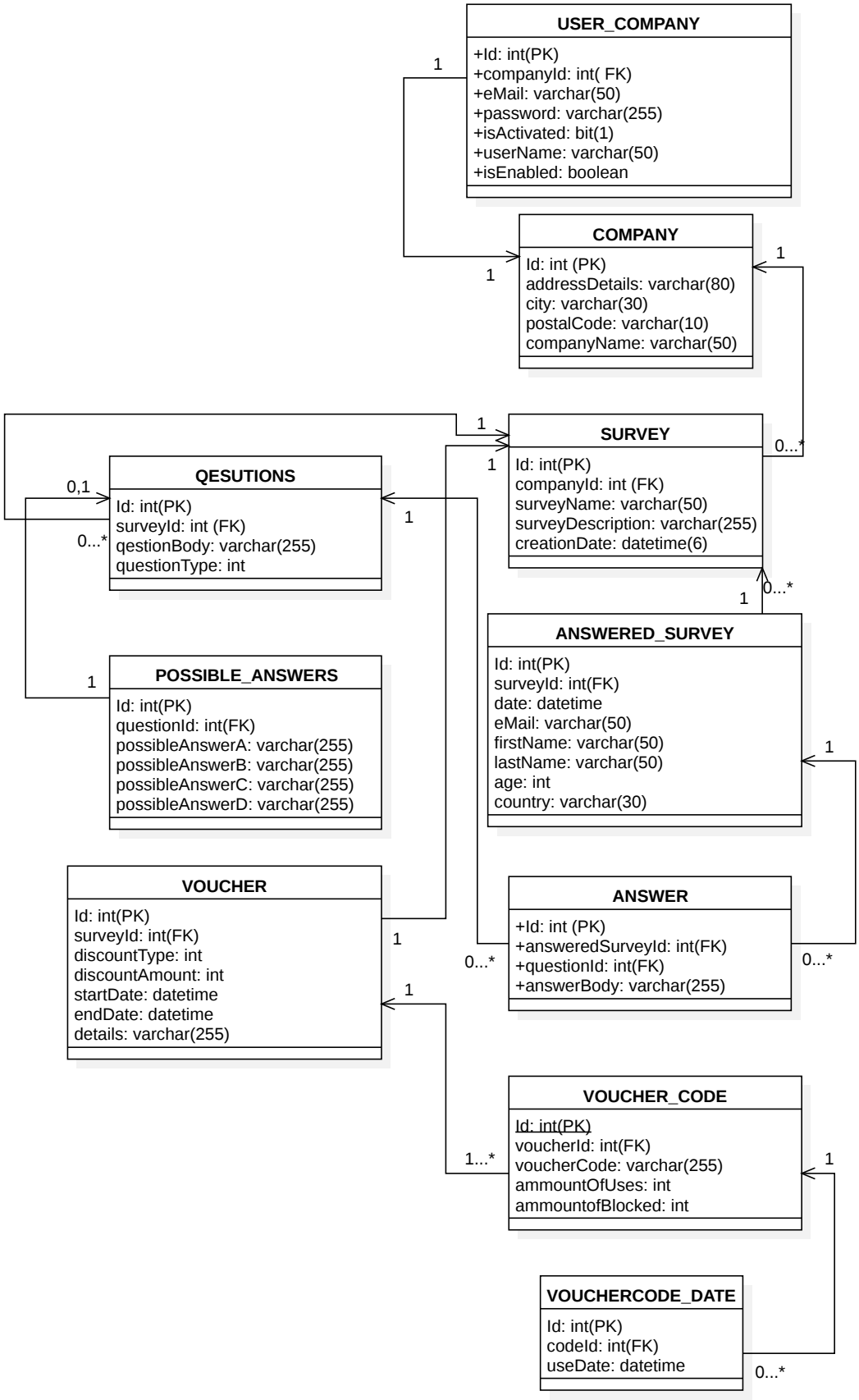
Klasy serwisowe w diagramie powyżej odpowiadają za część funckjonalności związanej z kontem firmy, tj. edycja danych, rejestracja, logowanie, zmiana hasła itd.

## 3.4 Diagramy aktywności

## 3.5 Diagramy sekwencji

## 3.6 Diagramy stanów

## 3.7 Projekt bazy danych




Rysunek 7: Diagram klas przedstawiający baze danych systemu

W celu odpowiedniego zrozumienia struktury naszej aplikacji koniecznie jest zaznajomienie się kilkoma kluczowymi założeniami naszego systemu, którego miały bezpośredni wpływ na sposób w jaki zaprojektowaliśmy baze danych. Najważniejszym założeniem było przypisanie tylko i wyłącznie jednego typu kupony do jednej ankiety. Żadna ankieta może oferować tylko wyłacznie jeden typ kupony jako nagrodę za wypełnienie ankiety. Każdy voucher (który reprezentuje typ nagrody) może już miec przypisany więcej niż jeden kupon, każdy z nich może byc wielokrotnego użytku ( wtedy należy zaznaczyć jak wiele osób z niego może skorzystać) lub też jednorazowego użytku.
Równie ważnym założeniem było stworzenie tablicy **VoucherCode_Date**, który odpowiada za "blokowanie" kupony na czas wypełnienia ankiet.

### 3.8 Opis protokołów

Ruch po naszej aplikacji ma miejsce przy użyciu protokołu **TLS**. Co ważne dostęp do każdej podusługi lub też podstrony naszego systemu wymaga połączenia zabezpieczonego, a użytkwonik chcący korzystać z naszej aplikacji przy użyciu protokołu **HTML** jest automatycznie przekierowywany na odpowiednią strone, która wykorzystuje już **TLS**. Podobno sytuacja ma miejsce z aplikacją mobilną, której połączenie z serwerem ma miejscu przy użyciu **TLS**.
Część aplikacji webowej dbająca o uwierzytelnianie oraz autoryzacje (ang. authentication and authorization) przychodzących połączeń są filtry, które możemy uznać za protokół. Są one częścią **Servlet'ów**, która ma za zadanie jak sama nazwa wskazuje filtrować przychodzące zapytania, jak również w zależności od samego zapytania w odpowiedni sposób reagować. Informacje zawarte w zapytaniach oraz odpowiedziach serwera, które zostają dynamicznie przechowycone przez filtry są wykorzystywane między innymi do tego aby zidentyfikować autora zapytania (autoryzacja) i jeżeli będzie taka konieczność nadać mu odpowiednie uprawnienia (uwierzytelnienie). Jednym z zaimplementowanych przez nas zabezpieczeniem jest autoryzacja na poziomie metod serwisowych, jak również nadawanie ról użytkownikom w zależności od typu uprawnienień, które posiadają. Każdej osboie korzystającej z naszej aplikacji zostaje nadany unikatowy indetyfikator sesji, którym zostaje powiązany on i uprawnienia, które posiada. Właśnie te informacje wykorzystuje **The Security Filter Chain**, który jest częścią jednego z używanych przez nas narzędzi. Kiedy użytkownik legitymuje się swoim identyfikatorem sesji **The Security Filter Chain** (od tego momentu nazywany filtrem w celu zwiększenia czytelności tekstu) sprwadza w bazie danych czy istnieje już powiązane z tym identyfikatorem połączenie, w przypadku gdy zostanie to potwierdzone filter sprawdza jakie uprawnienia posiada to połączenie, skutkuje to powiązaniem uwierzytelnieniu zapytania przychodzącego do serwera, co następnie umożliwia wykorzystanie autoryzacji na poziomie metod, która sprawdza czy nasze uprawnienia są wystarczające. Każdy użytkownik aplikacji webowej posiada odgórnie pewne uprawnienia, a w przypadku zalogowania się następuje dodatkowe uwierzytelnienie, gdzie jego identyfikator sesji wiązany z kontem do którego jest aktualnie zalogowany. To właśnie dzięki temu filter bezpieczeństwa potrafi jednoznacznie potwierdzić czy przychodzące zapytanie legitymujące się jedynie identyfikatorem sesji ma wystarczające uprawnienia do danej podusługi.
Ze względu na taki a nie inny sposób uwierzytelniania oraz autoryzacji konieczne również było użycie dodatkowego zabezpieczenia w postaci losowych tokenów chroniących przed atakami **CSRF** (ang. Cross-site request forgery), które w trakcie generowania zostają powiązane z identyfikatorem sesji, na którego życzenie zostały wygenerowane. Każdy użytkownik naszego serwisu podczas zapytań typu **POST** (sytuacja taka ma najczęściej miejsce podczas zatwierdzaniu różnego typu danych do serwera) musi "wylegitymować się" wygenerowanym wcześniej dla niego **CSRF tokenem** jak również identyfikatorem sesji, dopiero wtedy filter bezpieczeństwa zatwierdza takie zapytanie i przekazuje je do warstwy kontrolerów.
Częścią naszego systemu jest serwis mailowy, przez który jesteśmy w stanie komunikować się z klientami. Korzysta on z protkołu **STMP**.

## 4 Implementacja systemu

### 4.1 Użyte technologie

Całość systemu została zaimplementowana w jęzku obiektowym **Java 8** przy użyciu narzędzi dostępnych w **Java EE** (ang. enterprise edition). Do automatyzacji budowania projektu w przypadku aplikajci webowej został wykorzystany **Maven**, natomiast jeśli chodzi o aplikacje webową był to **Gradle**. Framework

w oparciu którego została napisana aplikacja mobilna jest **Retrofit**. Ułatwia on w znaczny sposób pracę z różnego typu HTML'owych API (ang. application programming interface), m.in REST API. Narzędzie w znaczny sposób ułatwiło nam mapowanie obiket na JSON'a oraz JSON'a na obiekty, jak również samo wysyłanie zapytań oraz odbieranie i interpretacja odpowiedzi serwera, pamietając o konieczności legitymacji się identyfikatorem sesji.
Jeśli chodzi o część systemu odpowiedzialną za aplikacje webową technologiami, które zostały wykorzystane są narzędzia z rodziny **Spring**. Program został zaprojektowany zgodnie z zasadami **GRASP**,a podczas implementacji z zachowaniem tych zasadł pomógł nam **Spring Core**, ułatwiał on wykorzystanie wzorca **DI** (ang. Dependency Injection), co skutkowało zachowaniem niskiego sprzężenia oraz wyskokiej spójności klas. Znaczna większość obiektów, które były wstrzykiwane poprzez **DI** były przedstawicielami wzorca projektowego **Singleton**, co skutowało zmniejszeniem zapotrzebania na zasoby komputera naszego serwera. Singletonami są wszystkie klasy serwisowe, kontrolery oraz klasy konfiguracyjne. Poza klasami konfiguracyjnymi wszystkie klasy wstrzykiwane przy użyciu **Spring Core** były implementacjami interfesjów, co pozwoliło **Springowi** w bardzo lekki i łatwy sposób przy użyciu refleksji budowanie **Proxy** tych obiektów w trakcie uruchomienia aplikacji i załadowania kontekstu,a nie jak miałoby by to miejsce w przypadki gdyby obiekty wstrzykiwane rozszerzały klase (lub też nie rozserzały żadnej kalsy abstrakcyjnej) modyfikując kod bitowy, który został wcześniej skompilowany. W różnych częsiach projektu mieliśmy doczynienia z programowaniem aspektowym, które również było możliwe, dzięki **Spring Core** oraz wcześniej stworzonym **Proxy** do wstrzykiwanych klas.
Framework, z którym najwięcej pracowaliśmy to **Spring MVC**. Jest to narzędzie, które umożliwia z jasny sposób budować aplikacje webowe w opraciu o wzorzec projektowy **MVC**. Dzięki niemu w łatwy sposób mogliśmy zbudować most łączący widok z modelem, czyli kontrolery. **Spring MVC** rownież ułatwił nam walidacje dancyh otrzymanych przez użytkowników poprzez pare na adnotacjach dostępnych w pakietach **javax.validation.api**. Wraz z **Spring MVC** wykorzystywany był **Jackson**, czyli biblioteka odpowiedzialana za mapowanie instancje klas na obiekty JSON, oraz w drugą stronę.
W zabezpieczaniu naszego systemu pomogły nam rownież **Spring Security** oraz **Spring Session**. Drugie z nich dawało nam większą kontrole nad sesjami i jej atrybutami (tak jak **CSRF tokeny** oraz blokowane kupony, które rownież wiązane były z sesją) jak również umożliwiało nam to przechowywanie identyfiaktorów sesji w bazie danych. Jeśli chodzi o **Spring Security** miał on wpływ na implementacja autoryzacji oraz uwierzytelniania w naszym serwisie. To narzędzie sprawdzało podczas otrzymania zapytania przez serwer, czy dany użytkownik legitymujący się identyfikatorem sesji ma odpowiednie uprawnienia do przeglądania danej podstrony jak również wywoływania metod zarówna w kontrolerze oraz w klasach serwisowych. Umożliwił w łatwy sposób również implementacje różnego typu handler'ów dla przypadków, gdy użytkownikowi nie udało się prawdiłowo zalogować/wylogować lub gdy właśnie probował przejść do strony, do której nie ma dostępu. Narzędzie to również automatycznie hashuje hasła podane przez użytkowników (po wcześniejszym zadeklarowaniu wybranego przez nas algorytmu haszującego), co jest znaczące jeśli chodzi o przechowywanie wrażliwych dla użytkoników danych.
Kolejnym narzędziem jest **Thymeleaf**, który został wykorzystanu do front-end'owej częsci projektu, czyli silnik umożliwiający tworzenie szablonów HTML, który jest w łatwy sposób jest integrowany ze **Spring MVC**. Dzięki możliwości korzystania m.in. z pętl, branch'ów oraz fragmentów kod HTML'owy stworzony przy użyciu Thymeleafa jest spójny oraz czytelny, gdyż miejscami przypomina kod jakiegoś standardowego języka programowania. Podczas prac nad modelem zostały wykorzystane również : **JavaScript**,**jQuery**,**CSS**.
Przydatnym narzędziem okazały się również **Spring WebFlow** oraz **Spring Mail**, które oferują wysokopoziomowe interfejsy dla wielostopniowych formularzy (ang. wizards) oraz wysyłce maili o zadanych wcześniej wyglądach (szablonach HTML). Pełna integracja z kodem Javowym w narzędziu odpowiedzialnym za formularze jest nie do opisania, ponieważ dzięki temu w formularzach jesteśmy wstanie zastosować branche, pracować na wprowadzonych przez użytkownika danych, jak również je przetwarzać na każdym etapie wykonywania formualrza.
Narzędziami przez nas użytymi były rownież frameworki ORM (ang. Object-Relational Mapping), a chodzi tu o **Hibernate**. Poprzez właśnie to narzędzie następowała komunikacja z naszą bazą danych **MySQL**, czyli wprowadzanie,edycja oraz usuwanie danych. **Hibernate** odpowiedzialny był za mapowanie krotek bazodanowych na instacje klas Modelu oraz instacji klas modelu na zapytania SQL'owe. Wykorzystany został rownież do wygenerowania gotowej bazy danych, na której później pracowaliśmy.

### 4.2 Obliczanie statystyk

```
1  @Cacheable("surveyStat")
2  @Override
3  public SurveyStatisticsDto getSurveysStatistics(int surveyId) {
4    Collection<AnsweredSurvey> answeredSurveys = getAllAnsweredSurveysWithDetails(
         surveyId);
5    Survey survey = companySurveyService.getSurveyByIdWithQuestion(surveyId);
6    SurveyStatisticsDto surveyStatisticsDto = new SurveyStatisticsDto();
7    surveyStatisticsDto.setAmmount(answeredSurveys.size());
8    surveyStatisticsDto.setSurveyName(survey.getSurveyName());
9
10   //average age
11   double averageAge = answeredSurveys.stream().mapToInt(a -> a.getUser().getAge()
         ).average().orElse(0.0);
12   surveyStatisticsDto.setAge(averageAge);
13
14   //average country
15   List<String> countries = answeredSurveys.stream().map(a -> a.getUser().
         getCountry()).collect(Collectors.groupingBy(Function.identity(), Collectors.
         counting())).entrySet().stream().sorted(Comparator.comparingLong(Map.Entry::
         getValue)).limit(3).map(Map.Entry::getKey).collect(Collectors.toList());
16   while (countries.size() != 3)
17     countries.add("N/A");
18   surveyStatisticsDto.setCountry(countries.toArray(new String[3]));
19
20   if (answeredSurveys.size() == 0)
21     return surveyStatisticsDto;
22
23   //initialaizing iterators
24   Iterator<AnsweredSurvey> answeredSurveyIterator = answeredSurveys.iterator();
25   Iterator<Question> qIterator = survey.getQuestions().iterator();
26   Iterator<Answer>[] aIteratorArray = new Iterator[answeredSurveys.size()];
27   IntStream.range(0, aIteratorArray.length).forEach(i -> aIteratorArray[i] =
         answeredSurveyIterator.next().getAnswersList().iterator());
28
29   int answersSize = answeredSurveys.size();
30   List<QuestionStatisticsDto> questionStatisticsDtoList = surveyStatisticsDto.
         getQuestionWithAnswersList();
31   while (qIterator.hasNext()) {
32     Question q = qIterator.next();
33     QuestionType qType = q.getQuestionType();
34     QuestionStatisticsDto questionStatisticsDto = new QuestionStatisticsDto();
35     questionStatisticsDto.setQuestionBody(q.getQuestionBody());
36     questionStatisticsDto.setQuestionType(qType);
37
38     switch (qType) {
39       case OPEN:
40         Arrays.stream(aIteratorArray).forEach(Iterator::next);
41         questionStatisticsDto.setAnswers(null);
42         break;
43       case RANGED:
44         double average = 0;
45         for (Iterator<Answer> anAIteratorArray : aIteratorArray) {
46           String temp = anAIteratorArray.next().getAnswer();
47           average += Double.parseDouble(temp);
48         }
49         questionStatisticsDto.getAnswers()[0].setAnswersStat(Double.toString(
               average / answersSize));
50         break;
```

```
51          default:
52            double[] apperances = new double[4];
53            for (Iterator<Answer> anAIteratorArray : aIteratorArray) {
54              Answer a = anAIteratorArray.next();
55              String[] splited = a.getAnswer().split(",");
56              for (String s : splited) {
57                switch (s) {
58                  case "A":
59                    apperances[0]++;
60                    break;
61                  case "B":
62                    apperances[1]++;
63                    break;
64                  case "C":
65                    apperances[2]++;
66                    break;
67                  case "D":
68                    apperances[3]++;
69                      }
70              }
71            }
72            IntStream.range(0, apperances.length).forEach(a -> questionStatisticsDto.
                getAnswers()[a].setAnswersStat(Double.toString(100 * apperances[a] /
                answersSize)));
73            questionStatisticsDto.setPossibleAnswers(q.getPossibleAnswers());
74            break;
75        }
76
77        questionStatisticsDtoList.add(questionStatisticsDto);
78      }
79      return surveyStatisticsDto;
80  }
```

Na powyższym listingu warto zauważyć, że metoda jest cache'owana tj. zapmietywane jest przez serwer wartość, która zostanie zwrócona dla danego argumentu. Przy dużej liczbie rozwiązanych ankiet obliczanie statystyk, może być stosunkow czasochłonne, a sam cache zapewnia nam, że nie statystyki dla tych samych danych nie będa liczone kilkukrotnie.

Pierwszym etapem jest pobranie z bazy danych listy wypełnionych ankiet, a następnie policzenie średniego wieku przy użyciu **Stream API**, **Lamba Expressions** oraz klasy generycznej **Optional<T>**. Przy użyiu tych samych "narzędzi" dostępnych w Javie zostaje obliczona posortowana lista krajów, względem częstości wypełniania danej ankiety w tym kraju. Lista ta jest ograniczona do 3 najlepszych wyników, a w przypadku gdy zawiera ona mniej niż 3 kraje zostaje wypełniona wyrażeniem "N/A". W przypadku gdy ankieta nie została rozwiązana zwracany jest pusty obiad, w innej sytuacji w pętli iterujemy po pytaniach w kolejnych odpowiedziach, tzn. na początku dla każdej odpowiedzi na pytanie numer 1 liczymy statystyki, następnie robimy to dla każdej istniejącej odpowiedzi na pytanie numer 2 itd. Jeżeli pytanie jest pytaniem typu "ranged" (tj. w skali od 0 do 10) liczona jest średnia arytmetyczna wyniku. Jeżeli jest to pytanie typu zamkniętego lub zamkniętego wieloktronego wyboru przy pomocy **Stream API**, **Lamba Expressions** oraz klasy generycznej **Optional<T>** liczona jest częstość występowania danej odpowiedzi (która należy do zbioru $\{A, B, C, D\}$). Następnie przy pomocy set'erów i get'erów klasy **SurveyStatisticsDto** obliczone wcześniej dane zostają "wrzucone" do obiektu wyjściowego, który jest ostatecznie zwracany przez metode.

## 4.3 Restowanie hasła dla konta

```
1  @RequestMapping(value = "forgot_password", method = RequestMethod.GET)
2  public String forgottenPassword(Model model) {
3    model.addAttribute("form", new ForgotPasswordForm());
```

```
4     return "auth/forgot_password.html";
5   }
6
7   @RequestMapping(value = "forgot_password", method = RequestMethod.POST)
8   public String forgottenPassword(@ModelAttribute(name = "form") @Validated
        ForgotPasswordForm forgotPasswordForm, BindingResult bindingResult) {
9     if (bindingResult.hasErrors())
10      return "auth/forgot_password.html";
11
12    UserCompany userCompany = userCompanyService.getUserByUserName(
          forgotPasswordForm.getUserName());
13    if (userCompany == null) {
14      bindingResult.rejectValue("userName", "email.dont.exist", messageSource.
            getMessage("message.wrong.email", null, LocaleContextHolder.getLocale()));
15      return "auth/forgot_password.html";
16    } else if (!userCompany.isEnabled()) {
17      bindingResult.rejectValue("userName", "account.not.activated", messageSource.
            getMessage("messages.account.not.activated", null, LocaleContextHolder.
            getLocale()));
18      return "auth/forgot_password.html";
19    } else {
20      PasswordResetToken passwordResetToken = tokenService.
            generateNewPasswordResetToken(userCompany);
21      mailService.sendPasswordResetEmail(passwordResetToken.getToken(), userCompany
            .getUsername());
22    }
23
24    return "redirect:/?acc=5";
25  }
```

Metoda oznaczona adnotacją $@RequestMapping(value = "/reset_password", method = RequestMethod.GET)$ przygotowywuje pusty formularz który ma zawierać jedynie adres e-mail konta dla którego użytkownik chce zresetować hasło. W przypadku, gdy użytkownik poda adres e-mail, który faktycznie znajduje się w naszej bazie danych zostaje generowany jednorazowy token (co zostało przedstawione w listingu poniżej), który zostaje wysłany w mail'u na wcześniej wskazany adres e-mail.

```
1   @Override
2   public PasswordResetToken generateNewPasswordResetToken(UserCompany userCompany)
        {
3     tokenDao.deleteUsersResetTokens(userCompany.getUsername());
4     PasswordResetToken passwordResetToken = new PasswordResetToken();
5     passwordResetToken.setUserCompany(userCompany);
6     passwordResetToken.setToken(RandomString.make(TOKEN_LENGTH));
7     tokenDao.addPasswordResetToken(passwordResetToken);
8     return passwordResetToken;
9   }
10
11  @Override
12  public PasswordResetToken validatePasswordResetToken(String passwordResetToken)
        throws WrongTokenException {
13    try {
14      return tokenDao.getPasswordResetTokenByToken(passwordResetToken);
15    } catch (NoResultException ex) {
16      throw new WrongTokenException();
17    }
18  }
```

Tylko i wyłącznie jeden token odpowiadający za resetowanie hasła może być pisany do konta, tak więc przed dodaniem nowego "PasswordResetToken" ewentualny obiekt, który mógł istnieć wcześniej jest kasowany.

```
@RequestMapping(value = "/reset_password", method = RequestMethod.GET)
public String resetPassword(@RequestParam("t") String token, Model model) {
  try {
    tokenService.validatePasswordResetToken(token);
    ResetPasswordForm form = new ResetPasswordForm();
    form.setResetPasswordToken(token);
    model.addAttribute("resetPasswordForm", form);
    model.addAttribute("tokenStatus", TokenStatus.OK);
  } catch (WrongTokenException ex) {
    model.addAttribute("tokenStatus", TokenStatus.WRONG);
  }
  return "token/reset_password.html";
}

@RequestMapping(value = "/reset_password", method = RequestMethod.POST)
public String resetPassword(@ModelAttribute @Validated ResetPasswordForm
    resetPasswordForm, BindingResult bindingResult, Model model) {
  if (bindingResult.hasErrors()) {
    try {
      tokenService.validatePasswordResetToken(resetPasswordForm.
          getResetPasswordToken());
      model.addAttribute("tokenStatus", TokenStatus.OK);
    } catch (WrongTokenException ex) {
      model.addAttribute("tokenStatus", TokenStatus.WRONG);
    }
    return "token/reset_password.html";
  }
  userCompanyService.changePassword(resetPasswordForm);
  return "token/reset_password_success.html";
}
```

Użytkownik klikający w link w mailu wysłanym przez nasz system przekazujemy w postaci parametru zapytani *GET* wcześniej wygenerowany token. Po wcześniejszej walidacji tokenu użytkwonik jest przeniesiony do strony zawierający formularz, w którym podaje dwa razy to samo nowe hasło, którym od momentu potwierdzenia formularza będzie się posługiwał.

## 4.4 Rejestracja

Do implementacji mechanizmu rejestracji zostało wykorzystane narzędzie o nazwie **Spring WebFlow**, które jest wykorzystywane do wielostopniowego formularza (ang. wizard). W pliku XML-owym możemy odwoływac się do metod serwisowych, implementować na podstawie tego branch'e oraz decydowac na jakim etapie formularza jaka grupa walidacji będzie walidowana.

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow http://
          www.springframework.org/schema/webflow/spring-webflow-2.4.xsd">

    <var name="company" class="pwr.groupproject.vouchers.bean.form.
        NewUserCompanyForm"/>

    <view-state id="step1" view="/signup/signup1.html" model="company"
                validation-hints="'validationGroup1'" >
        <transition on="nextStep" to="step2">
            <evaluate expression="userCompanyServiceImpl.validateUserCompany(
                company,messageContext)" />
        </transition>
        <transition on="cancel" to="cancel" validate="false" bind="false" />
    </view-state>
```

```
15
16      <view-state id="step2" view="/signup/signup2.html" model="company"
17                  validation-hints="'validationGroup2'">
18          <transition on="nextStep" to="success">
19              <evaluate expression="userCompanyServiceImpl.addUser(company)" />
20          </transition>
21          <transition on="cancel" to="cancel" validate="false" />
22          <transition on="previousStep" to="step1" validate="false" />
23      </view-state>
24
25      <end-state id="success" view="externalRedirect:/?acc=1" />
26      <end-state id="cancel" view="externalRedirect:/" />
27  </flow>
```

Dla pierwszego etapu rejestracji do modelu (który jest ten sam dla całego trwania procesu) dodawany jest pusty formularz, czyli obiekt w którym będziemy przechowywać wszystkie dane wprowadzone przez użytkownika. Następnie po zatwierdzeniu pierwszego etapu przez użytkownika , sprwadzamy czy wprowadzone dane są zgodne z wymogami przedstawionymi przez serwer (np. czy podany przez użytkownika adres e-mail nie znajduje się już w naszej bazie danych) poprzez $< evaluateexpression =$ "$userCompanyServiceImpl.validateUserCompany(company, messageContext)$"/ >, w której wywołujemy klase serwisową naszego systemu. W przypadku, gdy dane nie spełnią wymogów wyświetlamy odpowiednie komunikaty błędów, w których jasno informujemy klienta co należy zrobić, aby owe wymogi spełnić. W przypadku gdy dane zostaną zatwierdzone przez serwer przechodzimy do drugiego etapu formularza $< view - stateid = śtep2$"... $< /view - state >$. Tam użytkownik uzupełnia formularz kolejną serią danych (np. hasłem oraz jego powtórzeniem) po zatwierdzeniu których serwer poraz kolejny waliduje dane wejściowe, tym razem już cały formularz. W przypadku przejścia wszystkich wymogów walidacji zostaje wykonana instrukcja $< transitionon = ńextStep$"$to = śuccess$»... $< /transition >$, co skutkuje wywołanem funkcji serwisowej $addUser(obj : Company)$. Po pomyślnym dodaniu użytkownika do bazy danych proces trafia do linijki 25 w powyższym listingu i zostaje przekierowany na strone startową, na której dzięki parametrowi “acc” wyświetli się odpowiedni komunikat informujący o konieczności aktywacji konta przed zalogowaniem się.