AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca Magisterska

Sylwester Macura

kierunek studiów: **Informatyka Stosowana**

Wykorzystanie komponentów projektu Spring w Systemie Zarządzania Treścią

Opiekun: **dr inż. Barbara Kawecka-Magiera**

Kraków, Lipiec 2016

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomowa wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

(czytelny podpis)

Tematyka pracy magisterskiej i praktyki dyplomowej Sylwestra Macury, studenta V roku studiów kierunku Informatyka Stosowana, specjalność grafika komputerowa i przetwarzanie obrazów.

Temat pracy magisterskiej:Wykorzystanie komponentów projektu Spring w Systemie Zarządzania Treścią

Opiekun pracy:	dr inż. Barbara Kawecka-Magiera
Recenzenci pracy:	
Miejsce praktyki dyplomowej:	Wydział Fizyki i Informatyki Stosowanej Akademia Górniczo-hutnicza im. S. Staszica w Krakowie

Program pracy magisterskiej praktyki dyplomowej:

1. Zapoznanie się z tematem
2. Opracowanie bibliografii
3. Praktyka dyplomowa:
   - zapoznanie się z możliwościami projektu Spring
   - konfiguracja aplikacji i środowiska developerskiego
   - napisanie sprawozdania z praktyk
4. Implementacja systemu
5. Zebranie zdobytej wiedzy
6. Opracowanie pracy

…….. ……..

(podpis kierownika katedry) (podpis opiekuna)

Merytoryczna ocena pracy przez opiekuna

Końcowa ocena pracy przez opiekuna: . . . . . .

Data: . . . . . Podpis: . . . . . .

Skala ocen: (6.0 – celująca), 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 - niedostateczna

Merytoryczna ocena pracy przez recenzenta

Końcowa ocena pracy przez recenzenta: . . . . . .

Data: . . . . . Podpis: . . . . . .

Skala ocen: (6.0 – celująca), 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 – niedostateczna

Tę pracę chciałbym dedykować moim rodzicom

bez których nie zaszedłbym tak daleko.

Osobne podziękowania należą się

dr inż. Barbarze Kaweckiej- Magierze,

bez której udziału ta praca by nie powstała.

Spis treści

1 Wstęp

1.1 Wprowadzenie

W dzisiejszych czasach systemy działające w chmurze stają się coraz bardziej popularne. Oferują nie tylko większą wydajność, która jest potrzebna do przetwarzania coraz większych ilości danych, ale zapewniają łatwiejszą skalowalność. Wraz z nowym podejściem do tworzenia aplikacji pojawiają się nowe wyzwania, z którymi muszą się zmierzyć programiści i architekci. Błędy podczas tworzenia systemu mogą mieć poważne konsekwencje w działaniu aplikacji, gdy będziemy skalować naszą aplikację błędy także będą się powiększać.

Obecnie każdy może uruchomić swoje programy w chmurze, coraz więcej firm udostępnia swoją architekturę dla programistów. Na przykład Google posiada Google Compute Engine, Amazon stworzył Amazon Web Services, wszystkie te usługi są płatne, cena zależy od wykorzystanych zasobów: czasu procesora, pamięci RAM, użycia dysku. Przez to staje się niezmiernie ważne aby nasza aplikacja działała jak najwydajniej, ponieważ każde skalowanie naszego systemu będzie pociągało za sobą koszty.

Również twórcy oprogramowania dla programistów starają się wyjść naprzeciw tym oczekiwaniom dostarczając coraz to bardziej zaawansowane narzędzia i systemy, które będą działały w nowym środowisku. Przykładem jest tu projekt Spring, a konkretnie jeden z jego sub-projektów Spring Cloud. Dostarcza on wielu narzędzi nie tylko potrzebnych podczas samego pisania kodu aplikacji, ale również serwery pomocnicze które pozwolą uruchomić nasz system.

1.2 Cel pracy

Celem pracy było stworzenie systemu zarządzania treścią w oparciu o technologie dostępne w projekcie Spring. System ten jest skalowalny oraz łatwo rozszerzalny. System składa się z dwóch części, serwerowej napisanej przy pomocy Spring oraz klienta mobilnego stworzonego na platformę Android.

System jako całość posiada następujące funkcjonalności:

- zarządzanie użytkownikami (rejestracja, logowanie, weryfikacja przy pobieraniu dokumentów)
- tworzenie prostych dokumentów i formularzy
- prezentacja dokumentów i formularzy w aplikacji mobilnej
- generowanie dokumentów i formularzy do formatu PDF

Cześć serwerowa opiera się o architekturę małych łatwo skalowalnych serwisów (tak zwanych mikro-serwisów). W sumie istnieje sześć mikro-serwisów, trzy wspomagające:

- serwis proxy odpowiadający z przekierowywanie zapytań
- serwis konfiguracyjny odpowiadający z udostępnienie konfiguracji pozostałym serwisom
- serwis rejestrujący, w nim będą się rejestrować pozostałe serwisy

oraz trzy mikro-serwisy zawierające funkcjonalności:

- serwis z użytkownikami będzie odpowiadał za zarządzanie użytkownikami oraz dostarczanie o nich informacji pozostałym serwisom
- serwis z dokumentami będzie odpowiadał za zarządzanie dokumentami oraz udostępnianie ich aplikacji mobilnej
- serwis do generacji plików PDF będzie odpowiadał za generowanie plików PDF na podstawie dokumentów oraz za ich udostępnienie aplikacji mobilnej

Obecnie narzędziem które umożliwia zrealizowanie tych funkcjonalności jest Spring Cloud, jest to jeden z elementów projektu Spring. Jest to narzędzie otwartoźródłowe, dodatkowo świetnie współpracuje z pozostałymi elementami projektu Spring, które także zostaną wykorzystane w systemie. Spring Cloud dostarcza funkcjonalności serwisów wspomagających co bardzo ułatwia pisanie aplikacji.

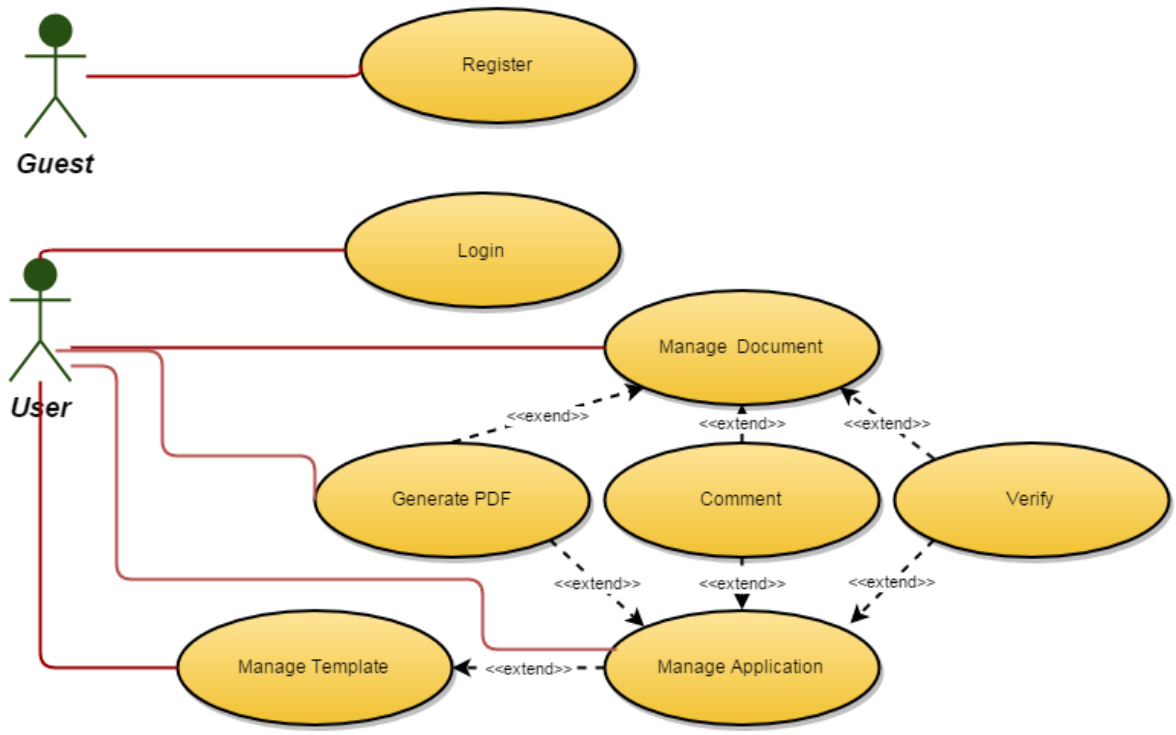
Aplikacja mobilna została stworzona na platformę Android. Jest to obecnie najpopularniejsza platforma mobilna na świecie. Umożliwia ona tworzenie aplikacji przy pomocy języka Java, jest to dużą zaletą ponieważ część serwerowa również używa tego języka. Dzięki temu obiekty transportowe mogą być współdzielone zarówno przez część kliencką jak i serwerową.

Do zbudowania całego systemu wykorzystano Gradle, ponieważ jest to zalecane narzędzie do budowy aplikacji Androidowych oraz jedno z narzędzi do budowy oraz uruchamiania Spring Cloud.

Aby zapewnić systemowi łatwą przenośność wykorzystano aplikację Docker. Pozwala to uruchomić cześć serwerową na każdym systemie z zainstalowanym Docker bez potrzeby konfiguracji środowiska. Kolejną zaletą jest wsparcie dla Docker przez większość usług hostujących. Innym istotnym powodem dla którego do uruchamiania systemu został użyty Docker jest także sposób dystrybucji naszego systemu. Aplikację zbudowaną przy użyciu Docker można dystrybuować poprzez centralny rejestr.

Szczegółowy opis zastosowanych w pracy technologi został zamieszczony w rozdziale drugim.

Zamieszczony poniżej ilustracja przedstawia diagram przypadków użycia.



Ilustracja 1: Diagram przypadków użycia

Niezalogowany użytkownik może się tylko zarejestrować. Zarejestrowany użytkownik może się zalogować. Po zalogowaniu można zarządzać rożnymi typami dokumentów dostępnych w systemie. Dodatkowo formularze oraz dokumenty można komentować, weryfikować oraz generować w postaci plików PDF. Formularz tworzymy wypełniając szablon.

Aby zaprezentować możliwości zrealizowanego w ramach pracy systemu, przygotowano filmik demonstrujący jego działanie, stanowi on załącznik do niniejszej pracy.

2 Omówienie technologii wykorzystanych podczas realizacji pracy

2.1 Java

Java jest obiektowym językiem programowania stworzonym w Sun Microsystems [1] (obecnie część Oracle). Programy napisane w Javie są kompilowane do kodu pośredniego (bytecode), a następnie uruchamiane na wirtualnej maszynie (ang. Java Virtual Machine, JVM) . Dzięki takiemu rozwiązaniu skompilowany program jest niezależny od platformy, wystarczy tylko że będziemy mogli zainstalować JVM. W JVM jest dodatkowo wbudowany mechanizm automatycznie zwalniający pamięć (ang. Garbage Collector) przez co nie musimy zajmować się zarządzaniem pamięcią. Składnia języka jest silnie wzorowana na C++. Oprócz tego Java posiada aktywną i rozbudowaną społeczność oraz wiele dostępnych narzędzi pomocniczych. Wszystko to sprawia że jest jednym z najpopularniejszych języków programowania. Java została wybrana jako język do stworzenia systemu ponieważ zarówno część serwerowa i cześć mobilna może zostać napisana w tym języku.

2.2 Gradle

Gradle jest otwartoźródłowym narzędziem budującym, pozwala na definiowanie skryptów budujących w języku Groovy [2]. Dzięki niemu możemy budować aplikacje na różne platformy napisane w różnych językach. Posiada bogatą kolekcje rozszerzeń, która pozwala rozbudowywać oraz upraszcza skrypty budujące. Gradle został wybrany jako narzędzie do budowy ponieważ jest to zalecany sposób budowy aplikacji Android. Dodatkowo posiada szereg pluginów które umożliwiają zbudowanie aplikacji serwerowych.

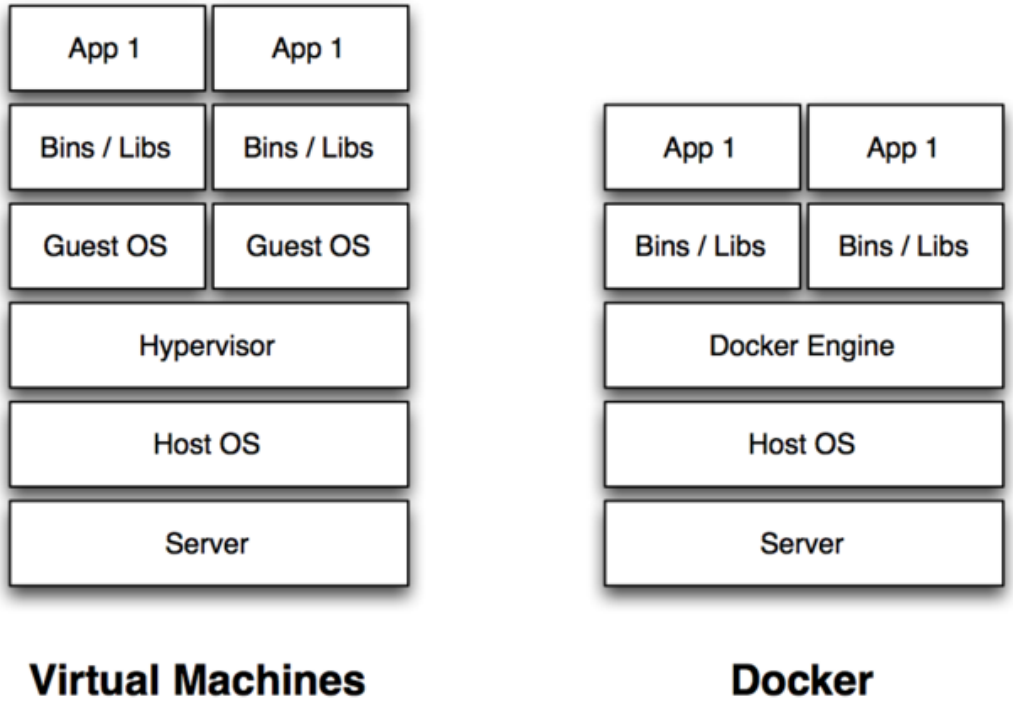
2.2.1 Użycie w pracy

W aplikacji gradle został wykorzystany do następujących rzeczy:

- Budowy części serwerowej
- Budowy aplikacji Android
- Generowanie plików Dockerfile
- Uruchamiania poszczególnych części aplikacji dla celów developerskich

2.3 Docker

Docker jest to otwarto źródłowe narzędzie do uruchamiania aplikacji wewnątrz kontenerów. Kontener jest to lekka i przenośna maszyna wirtualna, która może zostać uruchomiona na dowolnym serwerze z systemem Linux [3].



Ilustracja 2: Docker

http://core0.staticworld.net/images/article/2016/07/dockerswarm-fig01-100671308-large.idge.png

Plik Dockerfile to przepis jak stworzyć obraz Docker, zawiera on wszystkie elementy potrzebne do uruchomianie aplikacji oraz zależności. Następnie z tego obrazu możemy stworzyć kontener, czyli działającą maszynę wirtualną z naszą aplikacją. Dzięki wsparciu różnych platform PaaS (ang. Platform as a Service) dla Docker plik Dockerfile jest idealnym rozwiązaniem do wgrywania różnych usług chmurowych zachowując przy tym niezależność od platformy na którą jest wgrywany.

```
FROM java:8
MAINTAINER Sylwester Macura <sylwestermacura@gmail.com>
EXPOSE 8080
COPY libs/user-micro-service-0.0.1-SNAPSHOT.jar user-micro-service-0.0.1-
SNAPSHOT.jar
ENTRYPOINT ["java", "-Dspring.profiles.active=docker", "-jar", "user-micro-
service-0.0.1-SNAPSHOT.jar"]
```

Kod 1: Przykładowy Dockerfile

Każdy obraz Docker dziedziczy po innym, w tym przypadku dziedziczymy po obrazie java:8

zawiera on już zainstalowaną Javę w wersji 8. Kolejna linijka wskazuje na autora obrazu. Polecenie expose udostępnia porty kontenera, które będą widoczne z zewnątrz. Kolejna instrukcja wskazuje jak zbudować obraz, w przykładzie kopiujemy skompilowaną aplikację do obrazu. Entrypoint definiuje polecenie jakie ma się wykonać przy starcie kontenera.

2.3.1 Docker-compose

```
discovery-server:
  image: magisterka-cms/image-discovery-server
  ports:
    - "8770:8080"
  external_links:
    - image-config-server:config-server
edge-server:
  image: magisterka-cms/image-edge-server
  ports:
    - "8769:8080"
  links:
    - discovery-server
  external_links:
    - image-config-server:config-server
```

Kod 2: docker-compose.yml Przykład konfiguracji

Docker-compose jest narzędziem pomocniczym dla Docker, które pozwala na uruchomienie oraz połączenie wielu kontenerów. Narzędzie opiera się o plik docker-compose.yml, w którym konfigurujemy jakie obrazy Dockeramają być ściągnięte oraz jak mają być połączone, dodatkowo tworzy wewnętrzną sieć dla naszych kontenerów. Możemy upublicznić niektóre porty z konkretnych kontenerów aby uzyskać dostęp do aplikacji.

Kod 2 zawiera przykład pliku konfiguracyjnego docker-compose. W przypadku wykonania polecenia:

docker-compose up
Zostaną stworzone i uruchomione dwa kontenery. Pierwszy zostanie stworzony z obrazu o nazwie magisterka-cms/image-discovery-server, port kontenera 8080 zostanie zmapowany na port 8770 hosta. Dodatkowo kontener będzie posiadał wpis DNS (ang. Domain Name System) „config-server” który będzie prowadził do zewnętrznego kontenera o nazwie image-config-server. Następnie zostanie stworzony kolejny kontener z obrazu magisterka-cms/image-edge-server którego port 8080 zostanie zmapowany na port 8769 hosta. Ten kontener będzie posiadał dwa wpisy DSN jeden prowadzący do kontenera discovery-server i drugi prowadzący do zewnętrznego kontenera image-config-server.

Teraz możemy bardzo łatwo skalować wszerz nasze kontenery, wywołując polecenie:

docker-compose scale discovery-server=3
W efekcie zostaną stworzone dwie dodatkowe instancje obrazu magisterka-cms/image-discovery-server.

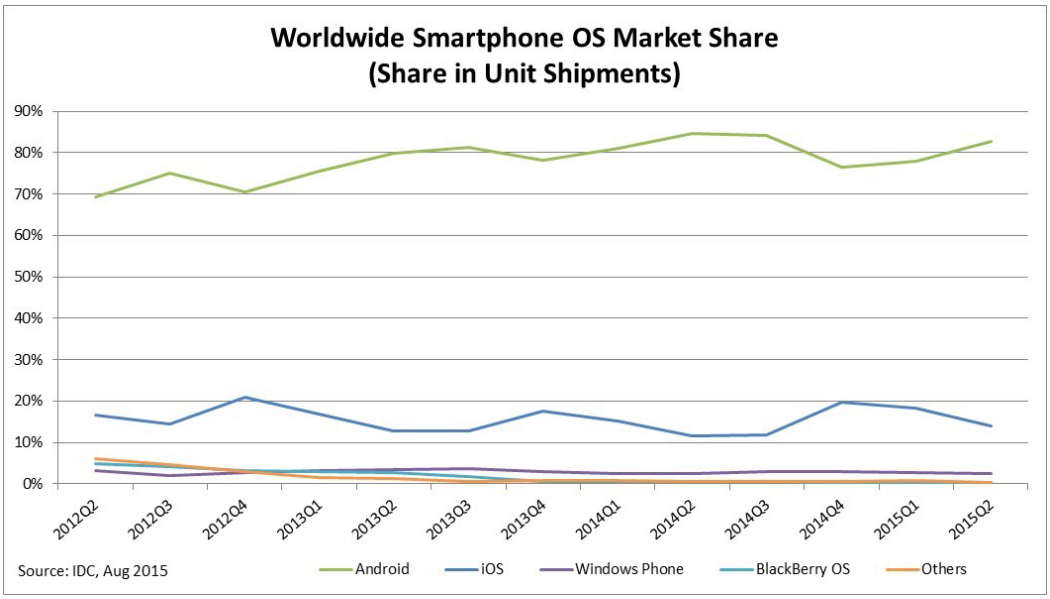
2.3.2 Docker Hub

Docker Hub jest centralnym repozytorium dla obrazów Docker. Z tego miejsca są ściągane obrazy gdy nie można ich znaleźć na lokalnej maszynie. Do zalet należą darmowa rejestracja oraz bogata kolekcja już gotowych obrazów.

2.3.3 Użycie w pracy

Docker został użyty aby wgrywać aplikację na serwer zewnętrzny Google Compute Engine. Dzięki użyciu narzędzia pomocniczego docker-compose wgranie całej aplikacji sprowadza się do wykonania jednego polecenia.

2.3.4 Android



Ilustracja 3: Popularność systemów mobilnych

http://www.idc.com/prodserv/smartphone-ms-img/chart-ww-smartphone-os-market-share.png

Android jest to system operacyjny z jądrem Linux dla urządzeń mobilnych takich jak telefony komórkowe, smartfony, tablety i netbooki [4]. Obecnie jest najpopularniejszym systemem operacyjny na urządzenia mobilne.

Oprócz dużej społeczności posiada bardzo rozwinięte narzędzia developerskie. Aplikacje natywne są tworzone w języku Java, a warstwa widokowa za pomocą plików XML, oprócz tego posiada zaawansowane narzędzia do dostosowywania widoków, kolorów w zależności od urządzenia.

2.3.5 Android Annotations

Android annotations jest otwartoźródłowym narzędziem wspomagającym tworzenie aplikacji na platformę Android [5]. Dzięki użyciu adnotacji możemy bardzo uprościć nasz kod oraz zwiększyć jego przejrzystość.

```
@EActivity(R.layout.activity_create_template)
@OptionsMenu(R.menu.create_template_menu)
public class ApplicationTemplateDetailsActivity extends AppCompatActivity {
    public static final String TEMPLATE_INTENT = "DOCUMENT_INTENT";
    ApplicationTemplateDTO template;
    @RestService
    ApplicationTemplateClient templateClient;
    @Bean
    ApplicationTemplateDetailsAdapter adapter;
    @Bean
    ActionsAdapter actionsAdapter;

}
```

Kod 3: Przykład użycia android Annotations

Na zaprezentowanym przykładzie widzimy activity androidowe do którego zostaje przypisany plik widoku activity_create_template.xml (adnotacja @Eactivity), oraz menu z pliku create_template_menu.xml (adnotacja @OptionsMenu). Po stworzeniu activity zostają wstrzyknięte instancje klas ApplicationTemplateDetailsAdapter oraz ActionsAdapter, oraz na podstawie interfejsu ApplicationTemplateClient zostaje wygenerowany klient do usługi REST (ang. Representational State Transfer).

2.4 Git

Git jest rozproszonym systemem kontroli wersji pierwotnie stworzonym na potrzeby rozwoju jądra Linuxa [6]. Obecnie jest jednym z najpopularniejszych systemów kontroli wersji (ang. Version Control System z skrócie VCS) używanych w tworzeniu oprogramowania. Jego rozproszona natura umożliwia łatwą współpracę w czasie tworzenia aplikacji. Dodatkowo istnieje wiele usług hostujących repozytoria Git np. Bitbucket czy GitHub.

2.4.1 GitHub

GitHub jest platformą hostującą repozytoria Git. Mamy do wyboru bezpłatne repozytoria publiczne oraz płatne repozytoria prywatne. Oprócz tego mamy do dyspozycji issue tracker, który pozwala na monitorowanie naszych postępów w pisaniu kodu, zgłaszanie błędów, podpinanie commitów pod poszczególne zgłoszenia oraz wyznaczanie kamieni milowych naszej aplikacji. Dodatkowo mamy do dyspozycji bogatą kolekcję webhooks, która pozwala na integrację z zewnętrznymi narzędziami np. TravisCI.

2.5 TravisCI

TravisCI jest narzędziem do Ciągłej Integracji (ang. Continuous Integration, CI). Ciągła integracja polega na ciągłym budowaniu i testowaniu aplikacji aby jak najszybciej wyłapywać błędy powstałe podczas tworzenia systemu. Travis posiada pakiet darmowy, (który choć z ograniczonymi funkcjonalnościami) stanowi i tak bardzo przydatne narzędzie do CI. Nasze środowisko konfigurujemy za pomocą pliku .travis.yml, pozwala to na łatwe dodawanie usług do naszego środowiska testowego np. Javy, Dockera.

```
sudo: required
language: java
services:
  - docker
script:
  - ./gradlew build -Pbuild-travis="" -x test
after_success:
  - docker login -e="$DOCKER_EMAIL" -u="$DOCKER_USERNAME" -p="$DOCKER_PASSWORD";
    docker push wemstar/magisterka-cms-image-edge-server;
```

Kod 4: .travis.yml Konfiguracja TravisCI

Plik .travis.yml dzieli się na następujące sekcje:

- sudo – określa czy potrzebujemy uprawnień administratora aby zbudować aplikację, w tym przypadku potrzebujemy dlatego umieszczamy wartość required
- language – określa język naszego projektu dla nas jest to java, dołącza JDK (ang. Java Development Kit) do środowiska oraz mavena i gradle
- script – jest to polecenie którym zbudujemy projekt, domyślnie Travis po wykryciu build.gradle zbuduje naszą aplikację narzędziem gradle, ale potrzebujemy dodatkowe zmienne aby zbudować obrazy dockera.
- after_sucess – co mamy zrobić po pomyślnym zbudowaniu aplikacji, tutaj zaloguje się do Docker Hub oraz przeniesie obraz wemstar/magisterka-cms-image-edge-server

2.6 Google Compute Engine

Google Compute Engine jest narzędziem IAAS i jest częścią Google Cloud Platform. Infrastructure as a Service (IAAS, z ang. „infrastruktura jako usługa”) to jeden z modeli chmury obliczeniowej. Jest to usługa polegająca na dostarczeniu całej infrastruktury informatycznej, takiej jak wirtualizowany sprzęt, skalowany w zależności od potrzeb użytkownika [7]. W aplikacji została wykorzystana do hostowania Dockera na którym jest uruchomiona aplikacja.

2.7 MongoDB

MogoDB jest to otwarta nierelacyjna baza danych (NoSQL) napisana w C++. Jej głównymi cechami jest łatwa skalowalność, wydajność oraz brak zdefiniowanej struktury danych. Dane są składowane w dokumentach podobnych do JSON (ang. JavaScript Object Notation) dzięki temu w sprawdza się lepiej w aplikacjach niż tradycyjne bazy SQL [8].

Bazy typu NoSQL zyskują coraz większą popularność dzieje się tak z kilku powodów:

- większa wydajność
- większa pojemność
- brak sztywnych reguł tworzenia obiektów

2.8 H2 Database

H2 Database jest relacyjną bazą danych napisaną w Javie. Dużą jej zaletą jest możliwość uruchomienia wewnątrz serwera (tgz. tryb embeded) oraz niewielki rozmiar (plik jar ma 1.5 MB) [9]. Tryb embeded ułatwia proces tworzenia aplikacji, ponieważ unikami problemów z zewnętrzną bazą danych. Wadą takiego rozwiązania jest brak przechowywania danych pomiędzy uruchomieniami serwera. H2 można uruchomić także w tradycyjny sposób jako zewnętrzny proces. Baza jest napisana w Javie co powoduje, że nie jest tak wydajna jak inne dostępne na rynku silniki SQL. Pomimo tych wad jest to świetne rozwiązanie dla małych projektów.

2.9 IntelliJ IDEA

IntalliJ IDEA jest zintegrowanym środowiskiem programistycznym (ang. Integrated Development Environment, IDE) stworzonym przez firmę JetBrains. Jest to jedno z najpopularniejszych narzędzi tego typu. Zapewnia świetna integracją z wieloma frameworkami oraz narzędziami wspomagającymi developerów np. Spring, SpringBoot, Docker, Git i wiele innych. Dodatkowo możemy rozszerzyć jego funkcjonalność dzięki bogatej bazie pluginów. Oprócz tego jest to wydajne środowisko oraz posiada intuicyjny interfejs.

2.9.1 Android Studio

Android Studio jest to wersja IntelliJ IDEA przeznaczona specjalnie do tworzenia aplikacji androidowych. Jest to oficjalne środowisko zalecane przez Google, posiada dodatkową integrację z usługami Google.

3 Projekt Spring

Projekt z Spring powstał jako narzędzie wspomagające tworzenie projektów J2EE (ang. Java Platform, Enterprise Edition). Jego pierwsza wersja została stworzona przez Roda Johnsona jako część książki po tytułem „Expert One-on-One J2EE Design and Development” i została wydana w 2002 [10]. Projekt wprowadzał wiele ułatwień i nowych rozwiązań które stawały się później standardami. Z czasem projekt się rozrósł, zarówno na inne języki programowania jak i na inne zagadnienia związane z tworzeniem aplikacji. Sam projekt jak i jego składowe są udostępnione na licencji Apache License 2.0. Aktualnie stabilną dostępną wersją jest 4.3.0. Ogromną zaletą Spring jest to że kod nie jest stale związany z frameworkiem. Pozwala to zmienić narzędzie na inne bez większych trudności.

3.1 Spring Core

Spring Core jest główną częścią projektu Spring, pozostałe projekty tylko rozszerzają jego funkcjonalności. Aby użyć jednego pod-projektów musimy w zależnościach mieć Spring Core.

3.1.1 Contener Benów

Ważną częścią Spring jest jego kontener Beanów. To tu są tworzone klasy które zdefiniujemy w kodzie oraz te dostarczane przez narzędzie Spring. Oprócz samego tworzenie klas to tutaj są wstrzykiwane zależność (ang. Dependency Injection). To również w tym miejscu nasza klasa jest opakowywana np. w aspekty czy klasy tworzące logi. Beany możemy stworzyć na trzy sposoby, pierwszy przez konfigurację w pliku XML, drugi przy użyciu adnotacji na naszej klasie, trzeci to stworzenie metody w Javie która wyprodukuje instancję naszej klasy.

```
@Component("verifyElementService")
public class VerifyElementService {
@Autowired
DocumentRepository documentRepository;
@Autowired
ApplicationRepository applicationRepository;

}
```

Kod 5: Przykład adnotacji @Component

W przypadku gdy mamy skonfigurowane automatyczne skanowanie klas Java wtedy zostaną stworzone obiekty wszystkich klas z adnotacją @Component. Wadą takiego rozwiązania jest to, że nie możemy wywołać własnego konstruktora. W Kod 5 zostanie stworzony Bean z klasy VerifyElementService o nazwie verifyElementService a następnie zostaną wstrzyknięte Beany documentRepository oraz applicationRepository.

```
<bean id="verifyElementService"
class="pl.edu.agh.fis.services.VerifyElementService">
    <property name="documentRepository" ref="documentRepository"/>
    <property name="applicationRepository" ref="applicationRepository"/>
  </bean>
```

Kod 6: Tworzenie instancji w XML

Kod 6 prezentuje sposób tworzenia instancji za pomocą XML, tworzony jest Bean z klasy VerifyElementService o nazwie verifyElementService a następnie są wstrzykiwane dwa obiekty documentRepository oraz applicationRepository.

```
@Configuration
public class DefaultConfiguration {
    @Bean(destroyMethod="close")
    public VerifyElementService verifyElementService() {
      VerifyElementService verifyElementService = new VerifyElementService();
      verifyElementService.documentRepository = documentRepository();
      verifyElementService.applicationRepository = applicationRepository();
      return  verifyElementService
    }

}
```

Kod 7: Tworzenie instacji w Java

W Kod 7 tworzymy Bean VerifyElementService następnie wstrzykujemy dwa Beany documentRepository oraz applicationrepository. Obie metody applicationRepository() oraz documentRepository() są funkcjami tworzącymi Beany.

3.1.2 Dependency Injection

Wstrzykiwanie zależności (ang. Dependency Injection) jest jednym z najczęściej stosowanych technik zarówno w samym projekcie Spring jak i w aplikacja stworzonych przy jego pomocy. Obiekty możemy wstrzykiwać za pomocą konstruktorów lub seterów. Od strony użytkownika możemy to osiągnąć odpowiednimi adnotacjami umieszczonymi na konstruktorze lub seterze. Jest też możliwość umieszczenie konfiguracji w xml. Wstrzykiwanie zależności umożliwia pisanie luźno powiązanych ze sobą obiektów (ang. Loose coupling), pozwala to na łatwa wymianę poszczególnych komponentów aplikacji. Dzięki temu nasza aplikacja staje się prostsza i łatwiej ją utrzymywać.

W Spring jest wiele możliwości wstrzyknięcia obiektów między innymi te wymienione w Kod 5, Kod 6, Kod 7. Jednak najpopularniejsza z nich jest adnotacja @Autowired (Kod 5). Wyszukuje ona odpowiednie Beany po typie i je wstrzykuje do obiektu. Jeśli nie znajdzie odpowiedniego obiektu lub znajdzie więcej niż jeden zostanie rzucony wyjątek. Adnotacja ma pole required, w przypadku gdy ma wartość false nie zostanie rzucony wyjątek jeśli obiekt nie zostanie znaleziony. Możemy użyć adnotacji @Qualifier która jako parametr przyjmuje nazwę Beanu, wtedy podczas wyszukiwania Beanu zostanie również uwzględniona jego nazwa.

```
@Component
public class VerifyElementController {
  @Autowired(required = false)
  @Qualifier("verifyElementService")
  VerifyElementService service;
}
```
Kod 8: Przykład adnotacji Qualifier

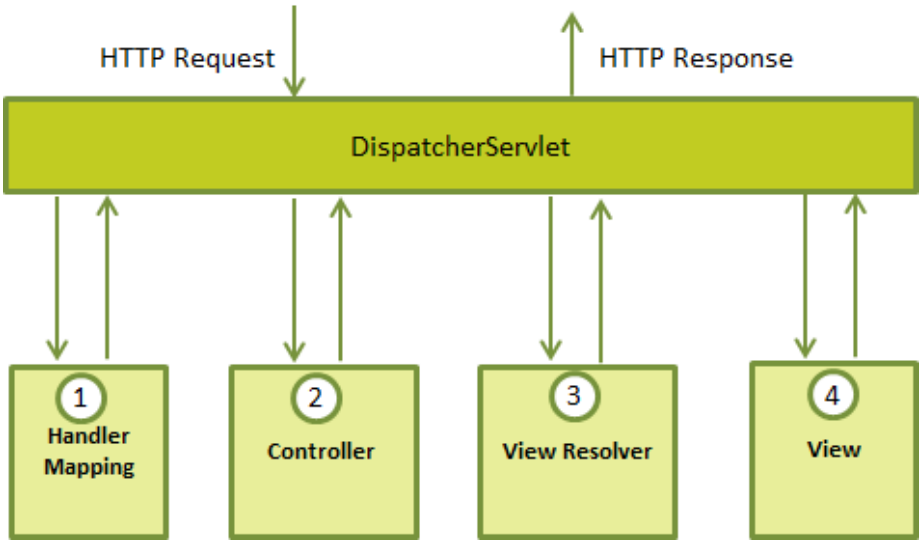
W Kod 8 tworzymy Bean z klasy VerifyElementController, następnie jest wstrzykiwany Bean typu VerifyElementService o nazwie verifyElementService, jeśli nie zostanie znaleziony wtedy pole service ma wartość null.

3.1.3 Spring MVC

Spring MVC jest jednym z komponentów Spring Core. Pozwala on na pisanie aplikacji webowych za pomocą wzorca MVC (ang. Model View Controller). MVC jest jednym z najpopularniejszych wzorców projektowych wykorzystywanych przy tworzeniu aplikacji webowych. Aplikacje składają się z 3 części:

- Model – odpowiada za pobieranie oraz przechowywanie danych.
- View – widok odpowiada za wyświetlanie danych z modelu oraz przekazywanie komunikatów do kontrolera.
- Controller – odpowiada za przetwarzanie danych w odpowiedzi na komunikaty przesłane z widoku.

Dzięki takiemu podejściu do pisania aplikacji, możemy łatwo wymieniać poszczególne elementy (modele, widoki i kontrollery). Daje nam to aplikację którą łatwo się rozwija.



Ilustracja 4: Architektura spring MVC

http://www.tutorialspoint.com/spring/images/spring_dispatcherservlet.png

Główną częścią Spring MVC jest DispatcherServlet, jest to servlet który rejestrujemy w pliku web.xml. W wywołanie metody HTTP przebiega następująco:

1. HandlerMapping mapuje zapytanie i wyszukuje odpowiedni kontroler, jeśli go znajdzie zapytanie jest tam przykazywane.
2. Controller jest miejscem gdzie wykonywana jest logika naszej aplikacji
3. ViewResolver wyszukuje widok w którym zostaną umieszczone dane z modelu.
4. View odpowiada za wyświetlenie wyniku zapytania.

```
<web-app id="WebApp_ID" version="2.4"
           xmlns="http://java.sun.com/xml/ns/j2ee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
           http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<servlet>
           <servlet-name>mvc-dispatcher</servlet-name>
           <servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
           <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
           <servlet-name>mvc-dispatcher</servlet-name>
           <url-pattern>*.htm</url-pattern>
   </servlet-mapping>
</web-app>
```

Kod 9: Przykład pliku web.xml

W tym przykładzie tworzymy servlet DispatcherServlet o nazwie mvc-dispatcher. Każdy adres URL kończący się na .htm zostanie do niego przekazany i jeśli znajdzie odpowiedni kontroler przekaże wywołanie do niego.

```
@Controller
public class HelloWorldController {

  @RequestMapping("/helloWorld")
  public String helloWorld(Model model) {
    model.addAttribute("message", "Hello World!");
    return "helloWorld.jsp";
  }
}
```

Kod 10: Przykład Kontrolera

Kontrolery możemy tworzyć poprzez rozszerzenie klasy AbstractController i nadpisanie odpowiedniej metody, jest to stary sposób i już nie zalecany. Polecanym sposobem jest użycie adnotacji.

W Kod 10 mamy przykład kontrolera, adnotacja @Controller wskazuje na to. Adnotacja @RequestMapping informuje dla jakiej ścieżki ma być wywołana metoda. Oprócz samej ścieżki możemy też ustawić dla jakiej metody HTTP będzie wykonywana funkcja. Sama metoda umieszcza w modelu parametr message o wartości „Hello World!" następnie wywołanie jest przykazywane do widoku helloWorld.jsp.

Adnotacja @RestController jest jedną z adnotaci wywodzących się z @Controller, ułatwia pisanie usług REST (ang. Representational State Transfer).

3.1.4 Spring AOP

Programowanie aspektowe (ang. Aspect-Oriented Programming, AOP) to sposób programowania wspomagający jak największą separację części programów niezwiązanych funkcjonalnie [11]. Główną przyczyną powstania tego paradygmatu były problemy z wykonywaniem zadań pobocznych (autoryzacji, monitoringu aplikacji) w ramach funkcjonalności. Powodowało to nie tylko zaciemnienie oryginalnego kodu ale również multiplikowanie tego samego kodu w różnych miejscach. Programowanie aspektowe stara się rozwiązać te problem poprzez przekierowanie zadań pobocznych do aspektów które opakowują naszą funkcjonalność. Z programowaniem aspektowym wiążą się trzy ważne pojęcia:

- Aspekt – zbiór zadań w ramach jednej funkcjonalności
- Joint Point – miejsce w którym zostanie nałożony aspekt
- Advice – funkcjonalność która ma zostać wykonana w ramach aspektu.

Jedną z implementacji dostępnych na platformę Java jest Spring AOP. Od strony implementacyjnej Spring AOP implementuje wzorzec Proxy na klasach wskazanych w Joint Point. Mamy do dyspozycji następujące Adnotacje:

- @Before – advice zostanie wykonane przed metodą
- @After – advice zostanie wykonane po metodzie
- @AfterReturning – advice zostanie wykonane po metodzie dodatkowo zostanie przechwycona zwracana wartość
- @AfterThrowing – advice zostanie wykonane w przypadku rzucenia wyjątku
- @Around – advice zostanie wykonane przed metodą, następnie zostanie wywołana metoda a później zostanie wykonana kolejna cześć advice

```
@Aspect
public class LoggingAspect {
	@Before(pointcut ="execution(* *(..))")
	public void logBefore(JoinPoint joinPoint) {
		System.out.println("" + Arrays.toString(joinPoint.getArgs()));
	}
	@After(pointcut ="execution(* *(..))")
	public void logAfter(JoinPoint joinPoint) {
		System.out.println("" + Arrays.toString(joinPoint.getArgs()));
	}
	@AfterReturning(pointcut = "execution(* *(..))", returning = "result")
	public void logAfterReturning(JoinPoint joinPoint, Object result) {
		System.out.println("" + Arrays.toString(joinPoint.getArgs()));
		System.out.println("" + result);
	}
	@AfterThrowing(pointcut = "execution(* *(..))",throwing= "error")
	public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {
		System.out.println("" + error.getMessage());
	}
	@Around(pointcut ="execution(* *(..))")
	public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {
		System.out.println("" + Arrays.toString(joinPoint.getArgs()));
		joinPoint.proceed();
		System.out.println("" + Arrays.toString(joinPoint.getArgs()));
	}
}
```

Kod 11: Przykład Aspektu

W Kod 11 mamy przykład Aspektu z pięcioma advice. Advice logBefore zostanie wykonany przed metodą i wydrukuje wszystkie parametry przesłane do funkcji, logAfter zrobi to samo ale po wyjściu z funkcji, logAfterReturning dodatkowo wypisze wynik metody, logAfterThrowing w przypadku rzucenia wyjątku wypisze jego wiadomość. Natomiast advice logAround wypisze argumenty przesłane do metody zarówno przed jej wykonaniem jak i po. Ważną częścią logAround jest wywołanie metody proceed(), to właśnie w tym miejscu jest wykonywana metoda na którą został nałożony Aspekt.

Każda adnotacja przyjmuje parametr pointcut to on określa na jakich metodh ma zostać wykonane advice. Parametr składa się z następujących części:

- kiedy ma zostać wykonany – w przykładzie execution oznacza, że podczas wywołania metody
- zwracany typ – w przykładzie * oznacza, że metoda może zwracać dowolną wartość
- sygnatura metody – w przykładzie * oznacza, że sygnatura może być dowolna
- przyjmowane prametry – w przykładzie (..) oznacza, że metoda może przyjmować dowolne parametry
-

3.2 Spring Security

Spring Security jest frameworkiem który pozwala na autoryzacje i uwierzytelnianie programów napisanych w Javie. Jego największa zaletą jest łatwość w rozszerzaniu tak aby mógł sprostać wyzwaniom klienta [12]. Do jego możliwości należą:

- wspomaganie autoryzacji i autentykacji
- zabezpieczenie przed atakami typu: session fixation, clickjacking, cross site request forgery i wiele innych
- integracja ze Spring MVC

W Spring Security, po poprawnym uwierzytelnianiu użytkownik otrzymuje jedną lub więcej ról. Każda rola składa się z pozwoleń. Na podstawie ról oraz pozwoleń sprawdzane jest czy użytkownik ma dostęp do zasobu. Role reprezentują wysokopoziomowe dostępy w systemie natomiast pozwolenia reprezentują niskopoziomowe dostępy. Pozwolenia są agregowane w role . Proces autoryzacji może dotyczyć całych klas lub poszczególnych metod. Preferowanym sposobem konfiguracji jest użycie odpowiednich adnotacji.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
	@Autowired
	private CustomAuthenticationProvider customAuthenticationProvider;
	@Override
	protected void configure(AuthenticationManagerBuilder builder) throws Exception {
	builder.authenticationProvider(customAuthenticationProvider);
	}
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.authorizeRequests().anyRequest().authenticated()
		.and().requestCache().requestCache(new NullRequestCache())
		.and().httpBasic();
	}
	@Override
	public void configure(WebSecurity web) throws Exception {
		web.ignoring().antMatchers("/hystrix.stream")
		.and().ignoring().antMatchers("/login");
	}
}
```

Kod 12: Przykład konfiguracji Spring Security

Adnotacja @PreAuthorize uruchamia autoryzację przed wywołaniem metody, sprawdzane jest czy użytkownik ma role ‘ROLE_USER’. W przypadku braku tej roli zostanie rzucony wyjątek. Oprócz konfiguracji za pomocą adnotacji możemy użyć kodu Javy aby ustalić dostęp do zasobów. Taka konfiguracja daje więcej możliwości bo poza samymi dostępami możemy także wyłączyć framework dla niektórych ścieżek, skonfigurować użytkowników, dodać własny system uwierzytelniania i wiele innych.

W tym przykładzie (Kod 12) mamy do czynienia z konfiguracją (o czym informuje nas adnotacja @Configuration), adnotacja @EnableWebSecurity uruchamia uwierzytelnianie dla całej aplikacji. Klasa rozszerza WebSecurityConfigurerAdapter aby możliwa była konfiguracja Spring Security za pomocą rozszerzenia odpowiednich metod. W metodzie configure(AuthenticationManagerBuilder builder) ustawiamy własny sposób uwierzytelnienia który zaimplementowaliśmy w CustomAuthenticationProvider.

W metodzie configure(HttpSecurity http) ustawiamy autoryzację dla wszystkich zapytań, sposobem uwierzytelnienia będzie Basic Http. Metoda configure (WebSecurity web) wyłącza Spring Security dla dwóch adresów "/login" oraz "/hystrix.stream"

3.3 Spring Session

Spring Session jest narzędziem pomocniczy do zarządzania sesją. Dostarcza API które jest niezależne od platformy na której zostanie uruchomione co daję na większą swobodę przy tworzeniu aplikacji.

3.4 Spring Data

Spring Data jest frameworkiem pomocniczym który ułatwia pracę z różnymi źródłami danych. Głównym jego zadaniem jest dostarczenie spójnego i niezależnego od źródła danych narzędzia. Za pomocą Spring Data możemy używać baz danych SQL (np. H2 Database, HSQLDB, MySQL) wtedy Hibernate jest używane do połączenia z bazą danych oraz generowania encji. Dodatkowo możemy użyć baz NoSQL (np. MongoDB, Apache Cassandra czy Redis).

```
@Entity
@Table(name = "USER_ENTITY")
public class UserEntity {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(name = "USER_ID")
  Long id;
  @Column(name = "USER_EMAIL",nullable = false,unique = true)
  String email;
  @Column(name = "USER_LOGIN",nullable = false,unique = true)
  String login;
  @Column(name = "USER_PASSWORD")
  String password;
}
```

Kod 14: Przykład Encji SQL

Aby używać Spring Data musimy zdefiniować klasy które będą odzwierciedlać informacje przechowywane w bazie danych.

W Kod 14 mamy przykład Encji SQL, odzwierciedla ona tablicę USER_ENTITY która zawiera kolumny:

- USER_ID – jest to wartość typu liczbowego, jest kluczem głównym w przypadku braku wartości jest ona generowana automatycznie.
- USER_EMAIL – przechowuje ciąg znaków, kolumna musi zawierać wartość oraz być unikalna
- USER_LOGIN – przechowuje ciąg znaków, kolumna musi zawierać wartość oraz być unikalna
- USER_PASSWORD – przechowuje ciąg znaków.

```
public class DocumentEntity {
  @Id
  public String id;
  public String title;
  public Date date;
  public List<ChapterEntity> chapters;
}
```

Kod 15: Przykład Encji MongoDB

Encja główna dla bazy danych MongoDB musi zawierać pole typu String o nazwie id, jego wartość jest generowana automatycznie w sposób losowy. Encja posiada następujące pola:

- id – pole przechowujące ciąg znaków jest identyfikatorem obiektu
- title – pole przechowujące ciąg znaków
- date – pole zawierające jedną z klas wbudowanych Javy
- chapters – lista obiektów własnej klasy, ChapterEntity nie jest encją główna i nie posiada id

Cała praca z baza danych odbywa się przy pomocy obiektów opakowujących. Wystarczy rozszerzyć interfejs CrudRepository lub jego pochodne. Dzięki temu zyskujemy następujące metody CRUD (ang. Create Read Update Delete):

- save – uaktualnia obiekt lub go tworzy jeśli obiekt o takim identyfikatorze nie istnieje. Możemy zapisać pojedynczy rekord jak i cała kolekcję.
- findOne – wyszukuje obiekt na podstawie id
- exist – sprawdza czy obiekt o podanym id istnieje w bazie danych
- findAll – zwraca wszystkie obiekty z bazy danych, możemy podać listę identyfikatorów obiektów.
- Count – zwraca liczbę dostępnych rekordów
- delete – usuwa rekordy z bazy danych. Możemy podać id obiektu, sam obiekt lub listę obiektów.
- deleteAll – usuwa wszystkie obiekty danego typu z bazy danych

Dodatkową ważną cechą jest możliwość pisania własnych metod, zostaną one zaimplementowane przez Spring Data na podstawie sygnatury metody.

```
public interface UserRepository extends CrudRepository<UserEntity,Long> {
  UserEntity findByLogin(String login);
}
```
Kod 16: Przykład repozytorium SQL

W tym przykładzie (Kod 16) używamy repozytorium do pracy z bazą danych SQL. Rozszerzając CrudRepository musimy podać dwa typy, pierwszy to nasza encja a drugi to klucz główny naszej encji. Dodatkowo w interfejsie znajduje się metoda która pobierze jednego użytkownika na podstawie jego loginu.

```
public interface DocumentRepository extends CrudRepository<DocumentEntity,String> {
}
```
Kod 17: Przykład repozytorium MongoDB

Przykład powyżej zawiera repozytorium wspomagające wymianę danych z bazą MongoDB.

Obie encje SQL (Kod 14) jak i MongoDB (Kod 15) są bardzo do siebie podobne tak samo jak repozytoria (Kod 16 i Kod 17). Dzięki temu zmiana bazy danych nie przysparza żadnych problemów. Przykładowo możemy sprawić aby encja UserEntity (kod 14) była zapisywana w MongoDB wystarczyłoby zmienić sama encje i zależności w projekcie. Nasz kod biznesowy pozostał by bez zmian.

3.4.1 Spring Data Rest

Spring Data Rest jest narzędziem pomocniczym dla Spring Data. Umożliwa w bardzo prostą implementację interfejsu REST na podstawie repozytorium.

```
@RepositoryRestResource(path = "user")
public interface UserRepository extends CrudRepository<UserEntity,Long> {
  UserEntity findByLogin(@Param("login") String login);
}
```
Kod 18: Przykład Spring Data Rest

Aby interfejs REST został wygenerowane musimy użyć adnotacji @RepositoryRestResource która przyjmuje jeden parametr path jest to ścieżka do zasobów.

Struktura zapytań wygląda następująco:

- /user GET – zwraca wszystkich dostępnych użytkowników
- /user POST – tworzy nowego użytkownika
- /user/1 GET – zwraca użytkownika o id 1

- /user/1 PUT – aktualizuje użytkownika o id 1
- /user/1 DELETE – usuwa użytkownika o id 1
- /user/search/findByLogin?login=user – zwraca użytkownika o loginie user

Domyślnie wartości są zwracane w formacie JSON (ang. JavaScript Object Notation) HAL (ang. Hypertext Application Language). HAL został zbudowany na dwóch koncepcjach: zasobach i linkach. Zasoby składają się z linków URI, zagnieżdżonych zasobów, standardowych pól. Linki zawierają linki URI do dodatkowych metod lub zasobów [13].

```
{
  "_embedded" : {
    "user" : [ {
      "id" : 1,
      "email" : "user@user.com",
      "login" : "user",
      "password" : "0ad7e108dbc1a0d6e8bb062c31950e90fb392b4a0e058cb5a",
      "_links" : {
        "self" : {
          "href" : "http://192.168.0.13:62038/user/1"
        },
        "userEntity" : {
          "href" : "http://192.168.0.13:62038/user/1"
        },
        "userGroups" : {
          "href" : "http://192.168.0.13:62038/user/1/userGroups"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://192.168.0.13:62038/user"
    },
    "profile" : {
      "href" : "http://192.168.0.13:62038/profile/user"
    },
    "search" : {
      "href" : "http://192.168.0.13:62038/user/search"
    }
  }
}
```

Kod 19: Przykład JSON HAL

Przykład pokazuje encje UserEntity w formacie JSON HAL, jest to wynik zapytania /user GET. Została zwrócona lista użytkowników wraz z dodatkowymi linkami. Ważnym linkiem jest search prowadzi on do własnych metod wyszukiwania np. findByLogin. Encja UserEtity (Kod 14) została podzielona na pola które zostały zwrócone bezpośrednio np. login, password oraz te zwrócone jako linki do których trzeba się odwołać.

3.5 Spring Boot

Spring Boot jest narzędziem pozwalającym w łatwy i szybki sposób tworzyć aplikację na podstawie frameworka Spring [14].

Możliwości [15]:

- aromatyczna konfiguracja – Spring Boot potrafi automatycznie dostarczyć konfigurację dla komponentów projektu Spring
- startowe zależności – w pliku do budowy Spring Boot podajemy jakie pod-projekty Spring chcemy
- monitorowanie aplikacji – wraz ze Sprign Boot możemy użyć narzędzia actuator które pozwala na monitorowanie naszej aplikacji.

Spring Boot silnie korzysta z zasady „konwencja ponad konfiguracja” (ang. convention over configuration). Dzięki temu już na początku dostajemy działająca aplikację z całym zestawem domyślnych ustawień które możemy w łatwy sposób zmienić.

Aplikacje możemy budować za pomocą dwóch narzędzi maven oraz gradle.

```
buildscript {
        ext { springBootVersion = '1.4.0.RELEASE' }
        repositories { mavenCentral() }
        dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
        }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'spring-boot'

jar {
        baseName = 'demo'
        version = '0.0.1-SNAPSHOT'
}

dependencies {
        compile('org.springframework.boot:spring-boot-starter-aop')
        compile('org.springframework.boot:spring-boot-starter-data-jpa')
        compile('org.springframework.boot:spring-boot-starter-security')
        compile('org.springframework.session:spring-session')
        compile('org.springframework.boot:spring-boot-starter-web')
        runtime('com.h2database:h2')
        testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

Kod 20: build.gradle dla Spring Boot

W tym przykładzie używamy Spring Boot w wersji 1.4.0.RELEASE, dzięki temu dalej w

konfiguracji nie musimy ustawiać konkretnych wersji innych komponentów projektu Spring. Spring Boot sam dobierze preferowane wersje narzędzi dla naszej aplikacji. Ważną sekcją pliku są zależności, to na ich podstawie zostanie zdefiniowana domyślna konfiguracja. Na przykład obecność spring-boot-starter-data-jpa oraz com.h2database.h2 spowoduje, że zostanie uruchomiona baza danych w trybie osadzonym dodatkowo aplikacja połączy się z nią za pośrednictwem Hibernate. Co więcej wszystko zostanie opakowane w Spring Data i nasze repozytoria skorzystają z tej bazy danych.

```
@SpringBootApplication
public class ConfigServerApplication {
   public static void main(String[] args) {
      SpringApplication.run(ConfigServerApplication.class, args);
   }
}
```

Kod 21: Gówna klasa Spring Boot

Najważniejszą rzeczą w przykładzie(Kod 21) jest adnotacja SpringBootApplication to ona sprawia, że aplikacja uruchamia się przy pomocy Spring Boot, oprócz tego konfiguruje component-scan. Jest to ustawienie które sprawia że wszystkie klasy w pakietach podrzędnych z adnotacją @Component lub pochodną zostaną stworzone jako Beany.

Do konfiguracji służą dwa pliki application i bootstrap oba mogą być w formacie .properties lub .yaml. Bootstrap jest używany do definiowania ustawień które muszą być zrobione przed uruchomieniem aplikacji (np. nazwa aplikacji), plik application ustawia zachowania które będą potrzebne podczas uruchamiania aplikacji.

```
spring.application.name=discovery-server
```

Kod 22: Przykład bootstrap.properties

W tym przykładzie nazwą naszej aplikacji będzie „discovery-server”.

```
server:
  port: 8888
```

Kod 23: Przykład application.yaml

Tutaj w pliku application.yaml zmieniamy domyślny port serwera na 8888.

Spring Boot całą aplikację wraz z serwerem aplikacyjnym (domyślnie Tomcat) kompresuje do jednego pliku jar. Do uruchomienia aplikacji potrzebujemy tylko i wyłącznie Java SE (ang. Java Standard Edition) w odpowiedniej wersji. Aby uruchomić taką aplikację musimy wykonać polecenie:

java -jar nasz_plik_jar.jar

Dodatkowo możemy definiować własne profile. Dla kazdego profilu możemy zdefiniować własny

zestaw konfiguracji. Możemy w tym celu utworzyć plik w formacie typPliku-nazwaProfilu.properties (np. application-docker.properties) lub definiujemy odpowiednią sekcję w pliku yaml.

```
server:
  port: 8770

---

spring:
  profiles: docker
server:
  port: 8080
```

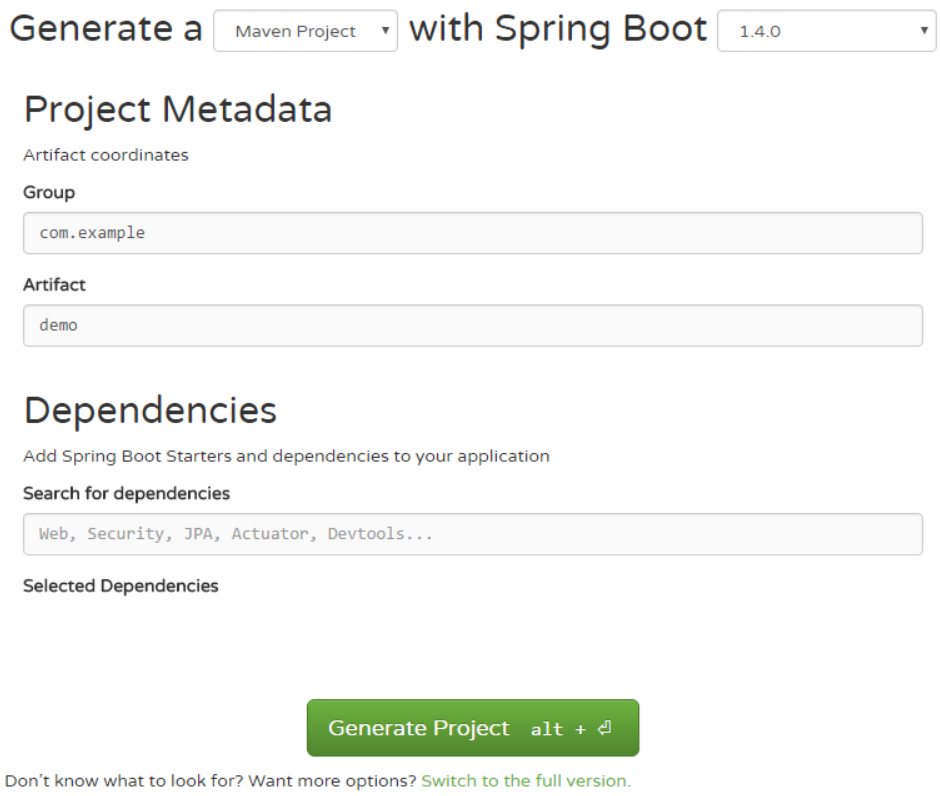
Kod 24: Przykład pliku application.yaml

Tutaj w zależności od profilu serwer będzie uruchomiony na porcie 8770 (brak profilu) lub na porcie 8080 (profil docker). Aby ustwić profil uruchamiamy aplikacje poleceniem:

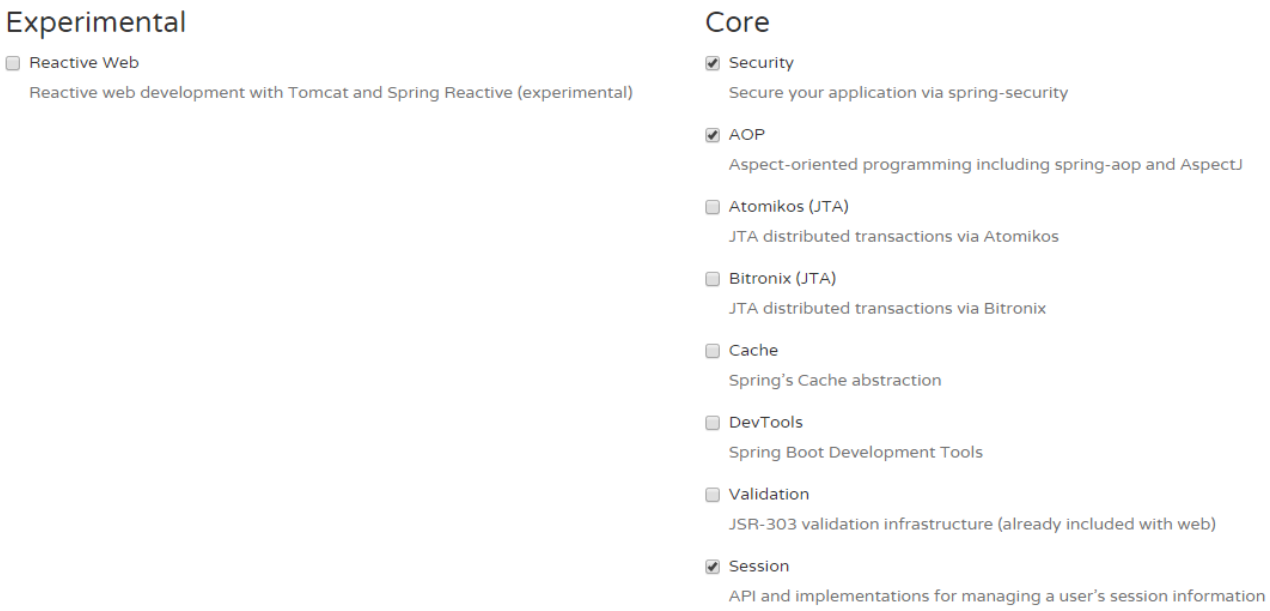
java -Dspring.profiles.active=nazwa_profilu -jar nasz_plik_jar.jar

3.5.1 Spring Initializr

Spring Initialize jest to narzędzie dostępne pod adresem https://start.spring.io/ . Służy ono do generowania projektów Spring Boot.



Ilustracja 5: Ekran główny Spring Initializr



Ilustracja 6: Wybór narzędzi dla naszej Aplikacji

Po wejściu na stronę wybieramy jakiego narzędzia do budowy użyć, do wyboru mamy gradle oraz maven. Następnie wybieramy wersje Spring Boot i podajemy podstawowe informacje dotyczące naszego projektu (nazwę, pakiety itd.). Po kliknięciu w „Switch to full version" ukazuje się nam widok z dostępnymi narzędziami dla Spring Boot.

Możemy wybrać narzędzia, które zostaną dołączone do naszego projektu, następnie klikamy „Generate Project", w ten sposób szablon naszego projektu w formie skompresowanej zostaje sciągnięty.

3.6 Spring Cloud

Spring Coloud dostarcza narzędzia do tworzenia systemów rozproszonych. Koordynacja systemów rozproszonych wymaga powielania pewnych schematów, Spring Cloud dostarcza te schematy przyśpieszając i ułatwiając budowę systemów rozproszonych. Spring Cloud może działać zarówno na małym środowisku developerskim jak i w dużych data center [16]. Cały Spring Cloud opiera się na Spring Boot, więc aby korzystać z możliwości Spring Cloud wystarczy dodać odpowiednie zależności

3.6.1 Eureka

Ważnym elementem systemu rozproszonego jest rejestr serwerów. To tu serwery się rejestrują i dowiadują o innych elementach systemu rozproszonego. Implementacja Spring Cloud nazywa się Eureka i została stworzona przez firmę Netflix. Eureka może działać jako pojedynczy serwer lub jako klaster serwerów.

```
dependencies {
  compile('org.springframework.cloud:spring-cloud-starter-eureka-server')
}
```
Kod 25: Zależności dla Eureka

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoverServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(DiscoverServerApplication.class, args);
  }
}
```
Kod 26: Główna klasa dla serwera Eureka

Dodanie zależności spring-cloud-starter-eureka-server oraz użycie adnotacji @EnableEurekaServer spowoduje, że aplikacja Spring Boot zostanie uruchomiona jako serwer Eureka. Domyślnym portem dla serwera jest 8761 i pod takim portem na localhost klienci będą szukać serwera rejestrującego.

Aby serwer zarejestrował się do Eureka musimy zrobić dwie rzeczy:

- dodać zależność

```
dependencies {
  compile('org.springframework.cloud:spring-cloud-starter-eureka')
}
```
Kod 27: Zależności dla klientów Eureka

- użyć adnotacji @EnableEurekaClient

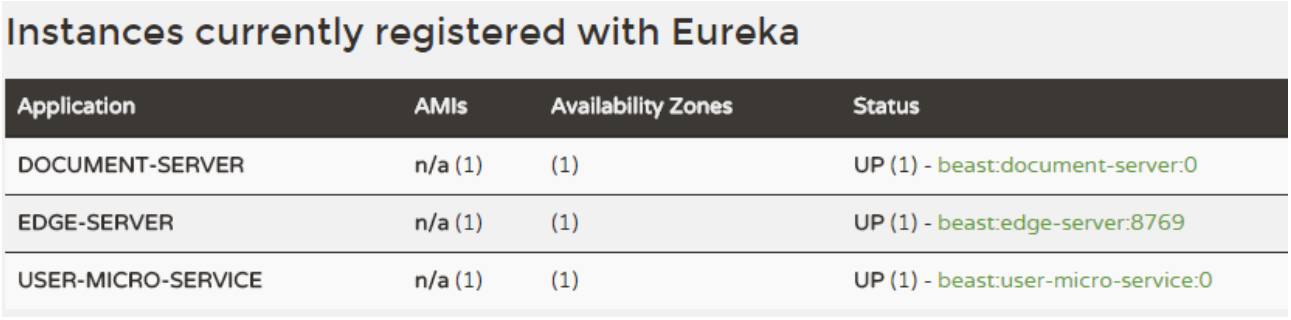
```
@SpringBootApplication
@EnableEurekaClient
public class UserMicroserviceApplication {
  public static void main(String[] args) {
    SpringApplication.run(UserMicroserviceApplication.class, args);
  }
}
```
Kod 28: Główna klasa dla klienta Eureka

Dodanie zależności spring-cloud-starter-eureka oraz adnotacji @EnableEurekClient spowoduje, że domyślnie klient będzie się starał zarejestrować pod adresem localhost:8761. Możemy zmienić adres dla klientów używając odpowiedniego wpisu w application.properties

```
eureka.client.serviceUrl.defaultZone = http://localhost:8770/eureka/
```

Kod 29: Zmiana adresu docelowego dla klientów Eureka

W powyższym przykładzie klienci będą szukali serwera eureka pod adresem localhost:8770. Po zarejestrowaniu instancji będzie ona co jakiś czas ją odnawiać aby serwer miał jak najbardziej aktualną listę dostępnych klientów. Kazdy klient tworzy cache w którym przetrzymuje adresy do pozostałych klientów, pozwala to ograniczyć ilość zapytań oraz sprawne działanie aplikacji nawet gdy Eureka przestała działać.



Ilustracja 7: Przykład widoku serwera Eureka

Na ilustracji widzimy że serwer eureka zarejestrował trzy serwery. Wszystkie rodzaje serwerów mają po jednej instancji i działają poprawnie (status UP).

3.6.2 Zuul

Zull jest serwerem proxy który przekierowuje zapytania do odpowiednich serwerów. Lista serwerów jest pobierana z Eureka. Domyslnie zapytania są przekierowywane na podstawie nazwy serwera, ogólna postać wygląda następująca:

http://adres-zull[:port]/nazwa-serwera/

Na przykład takie zapytanie:

http://localhost:8762/document-server/

Zostanie przekierowane do instancji document-serwer. Możemy ustawić własne przekierowania w jednym z plików konfiguracyjny serwera Zuul.

```
zuul:
  routes:
    documents:
      path: /documents/**
      serviceId: document-serwer
```

Kod 30: Przykład application.yaml dla serwera Zuul

W tym przykładzie zapytanie http://localhost:8762/documents/ zostanie przekierowane do serwera document-serwer. Oprócz samego przekierowywania zapytań Zuul jest dobry miejscem na wykonanie autoryzacji i autentykacji użytkowników, możemy to np. osiągnąć za pomocą Spring

Security. Na serwerze Zull można także filtrować zapytania np. dla urządzeń mobilnych zostanie zastosowana inna ścieżka.

```
dependencies {
  compile('org.springframework.cloud:spring-cloud-starter-zuul')
}
```
Kod 31: Zależność serwera Zuul

Aby używać Zuul musimy dodać odpowiednią zależność.

```
@SpringBootApplication
@EnableZuulProxy
public class EdgeServerApplication {
   public static void main(String[] args) {
      SpringApplication.run(EdgeServerApplication.class, args);
   }
}
```
Kod 32: Uruchomienie Zuul

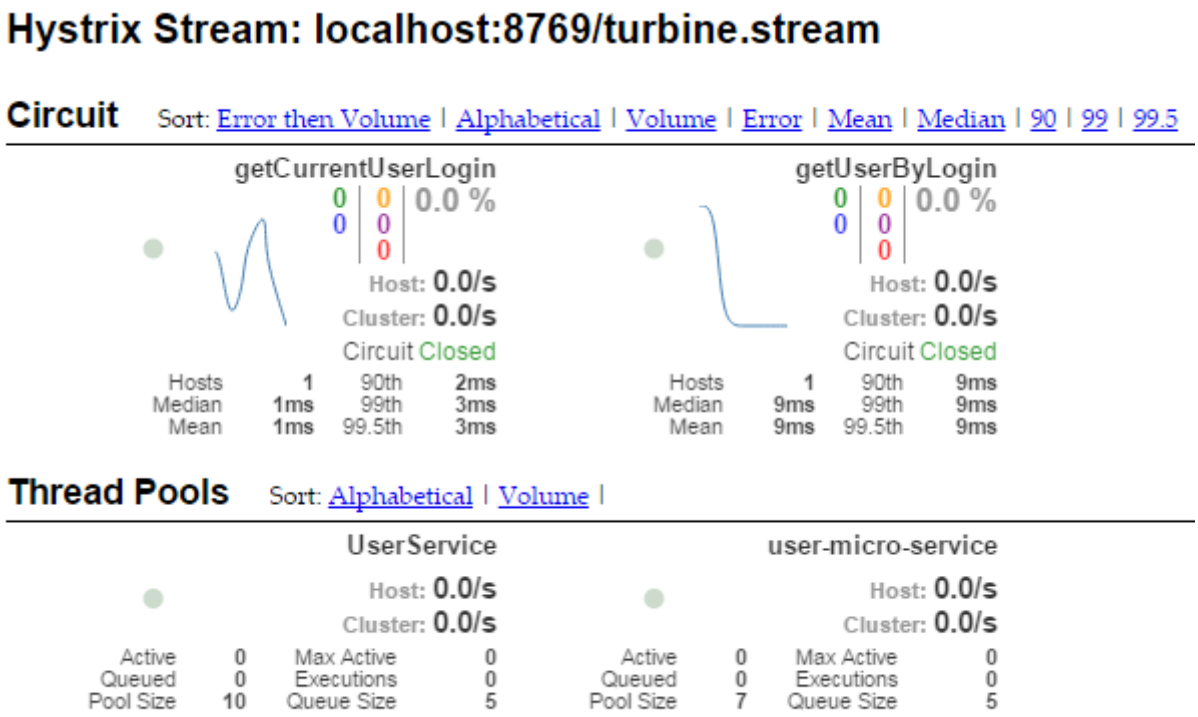
Następnie aby uruchomić Zuul musimy dodać adnotację @EnableZuulProxy na głównej klasie serwera.

3.6.3 Hystrix

W środowiskach rozproszonych częstym zjawiskiem jest odwoływanie się jednych elementów systemu do innych. Gdyby jeden z komponentów przestał działać, mogłoby to zagrozić stabilności całej aplikacji. Hystrix zapobiega roznoszeniu się błędów po całym systemie między innymi poprzez monitorowanie stanu poszczególnych komponentów oraz izolację „uszkodzonych” ogniw systemu.

Hystrix implementuje circuit breaker pattern. Każdy obwód ma dwa stany zamknięty i otwarty. Gdy serwer działa poprawnie obwód pozostaje zamknięty. Kiedy zapytanie do jednego z serwerów przekroczy pewną ilość błędów (domyślnie dla Hystrix 20 błędów w czasie 5 sekund) obwód zostaje otwarty i zapytanie nie dociera do serwisu, zamiast tego jest zwracana domyślna wartość [17].

Hystirx publikuje dane w postaci strumienia tekstowego przypominającego format JSON. Jest on dostępny pod adresem http://nazwa-serwera:port/hystrix.stream. W przypadku gdy posiadamy wiele serwerów najlepszym sposobem jest zebranie danych w jednym miejscu. Do agregacji tego typu informacji służy narzędzi Turbine. Narzędzie może działać na dwa sposoby albo na podstawie listy pobranej z Eureka odpytuje poszczególne serwery o ich strumienie Hystrix, albo same serwery wysyłają informacje o sobie do głównego serwera z Turbine.



Ilustracja 8: Hystrix Dashboard

Przydatnym narzędziem do monitorowania Hystrix jest tablica Hystrix Dashboard. Tablica pozwala w przejrzysty sposób oglądać status poszczególnych obwodów. Udostępnia podstawowe podstawowe dane dotyczące ilości poprawnych i niepoprawnych wywołań oraz stan obwodu.

```
dependencies {
compile('org.springframework.cloud:spring-cloud-starter-hystrix')
 compile('org.springframework.cloud:spring-cloud-starter-hystrix-dashboard')
 compile('org.springframework.cloud:spring-cloud-starter-turbine')
}
```

Kod 33: Potrzebne zależności dla Hystrix, Turbine, Hystrix Dashboard

Zależności obecne w przykładzie:

- spring-cloud-starter-hystrix – dodaje hystrix do projektu
- spring-cloud-starter-hystir-dashboard – dodaje Hystrix Dashboard do naszego projektu
- spring-cloud-starter-turbine – dodaje Turbine do naszego projektu

```
@SpringBootApplication
@EnableCircuitBreaker
@EnableHystrixDashboard
@EnableTurbine
public class EdgeServerApplication {
   public static void main(String[] args) {
      SpringApplication.run(EdgeServerApplication.class, args);
   }
}
```

Kod 34: Przykład użycia Hystrix

W przykładzie (Kod 34) widzimy 3 adnotacje:

- @EnableCircuitBreaker – uruchamia ona Hystrix
- @EnableHystrixDashboar – uruchamia ona Dashboard dla Hystrix
- @EnableTurbine – uruchamia ona Turbine

Po uruchomieniu serwera Hystrix Dashboard będzie dostępny pod adresem /hystrix, Hystrix możemy zobaczyć pod adresem /hystrix.stream a strumień Turbine pod /turbine.stream

3.6.4 Ribbon

Ważną cechą systemów rozproszonych jest ich skalowalność wszerz czyli multiplikowanie serwerów jednego typu. Gdy posiadamy wiele instancji serwerów przeznaczonych do tych samych zadań naszym celem jest jak najlepsze ich wykorzystanie. Tę funkcjonalność czyli równoważenie obciążeń (ang. load balancing) implementuje Ribbon. Ribbon działa na dwóch płaszczyznach. Pierwsza z nich to komunikacja pomiędzy serwerami. Po stronie klienta Ribbon pobiera z Eureka listę dostępnych serwerów i wybiera ten do którego ma zostać wysłane zapytanie. Drugi sposób to gdy odwołujemy się do serwisu przez proxy Zull, wtedy to po stronie serwera proxy Ribbon wybiera instancje do której ma zostać przekierowane zadanie.

Ribbon jest domyślnie używany przez klientów Feign więc nie potrzeba dodatkowej konfiguracji.

3.6.5 Feign

Feign jest narzędziem ułatwiającym pisanie zapytań do innych serwerów które znajduja się wewnątrz naszej aplikacji. Feign przy tworzeniu zapytania współpracuje z Ribbon.

```
@FeignClient("user-micro-service")
public interface UserCore {
  @RequestMapping(method = RequestMethod.GET, value = "/user")
  Resource<UserDTO> getUsers();
}
```

Kod 35: Przykład klienta Feign

Powyższy kod jest przykładem na użycie Feign. Adnotacja @FeignClient przyjmuje jako parametr

nazwę serwera do jakiego ma się odwołać klient. Następnie w RequestMapping podajemy typ metody Http oraz dalszą część URI. Interfejs zostanie zaimplementowany przez framework. Przy wywołaniu metody getUsers() nastąpi pobranie listy użytkowników z serwera user-micro-service.

```
dependencies {
  compile('org.springframework.cloud:spring-cloud-starter-feign')
}
```
Kod 36: Zależności Feign

Aby uruchomić Feign najpierw musimy dodać odpowiednie zależności, w tym wypadku spring-cloud-starter-feign.

```
@EnableFeignClients
@SpringBootApplication
public class DocumentMicroserviceApplication {
  public static void main(String[] args) {
    SpringApplication.run(DocumentMicroserviceApplication.class,
args);
  }
}
```
Kod 37: Uruchomienie Feign

Następnie musimy na głównej klasie serwera użyć adnotacji @EnableFeignClients która uruchamia generowanie kodu klientów Feign.

3.6.6 Config Server

Jednym z problemów systemów rozproszonych jest utrzymywanie konfiguracji wielu serwerów. Narzędziem które to ułatwia jest Spring Config Serwer. Jest to centralny serwer przeznaczony do przechowywania konfiguracji. Konfigurację możemy zapisać w pliku yaml lub properties lokalnie na serwerze lub wykorzystać repozytorium Git. W plikach application.[yaml|properites] definiowane są domyślne ustawienia dla serwerów. Jeśli chcemy aby nasz serwer miał inne ustawienia musimy stworzyć plik nazwa-serwera.[properties|yaml]. Możliwa jest też konfiguracja dla poszczególnych profili, wtedy nasz plik nazywamy nazwa-serwera-profil[.properites|yaml]. Adres do serwera konfiguracyjnego podajemy w pliku bootstrap.[properties|yaml], musimy także podać nazwę naszego serwera. Ważną rzeczą na jaką trzeba zwrócić uwagę jest status serwera konfiguracyjnego. Serwer musi być w pełni uruchomiony dopiero później możemy uruchamiać pozostałe serwery.

```
dependencies {
  compile('org.springframework.cloud:spring-cloud-config-server')
}
```
Kod 38: Zależności Config Server

Aby uruchomić serwer konfiguracyjny potrzebujemy zależności spring-cloud-config-server.

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```
Kod 39: Główna klasa serwera konfiguracyjnego

Następnie musimy użyć adnotacji @EnableConfigServer aby uruchomić serwer konfiguracyjny, domyślnie zostanie on uruchomiony na porcie 8888 i na takim porcie domyślnie będą go szukali klienci.

```
spring.cloud.config.server.git.uri=https://github.com/wemstar/MagisterkaConfigCMS
```
Kod 40: Zródło konfiguracji

Ostatnią rzeczą jaką musimy zrobić aby serwer działał poprawnie jest podanie adresu systemu kontroli wersji, skąd będzie pobierana konfiguracja czyli do spring.cloud.server.git.uri. Dużą zaletą takiego podejścia jest to, że nie potrzebujemy restartować serwera konfiguracyjnego w przypadku zmiany konfiguracji.

Aby serwer mógł pobrać konfigurację musimy zrobić następujące rzeczy:

```
dependencies {
  compile('org.springframework.cloud:spring-cloud-starter-config')
}
```
Kod 41: Zależności dla klietów Config Server

Najpierw musimy dodać odpowiednią zależność w tym wypadku spring-cloud-starter-config. Nie musimy używać żadnej adnotacji Spring Boot wykryje zależność i spróbuje pobrać konfigurację z domyślnego adresu: localhost:8888

```
spring.cloud.config.uri=http://localhost:8889/
```
Kod 42: Zmiana domyślnej konfiguracji klienta

Aby zmienić adres serwera z którego będzie pobierana konfiguracja musimy w pliku bootstrap.[properties|yaml] przypisać do spring.cloud.config.uri odpowiedni adres.

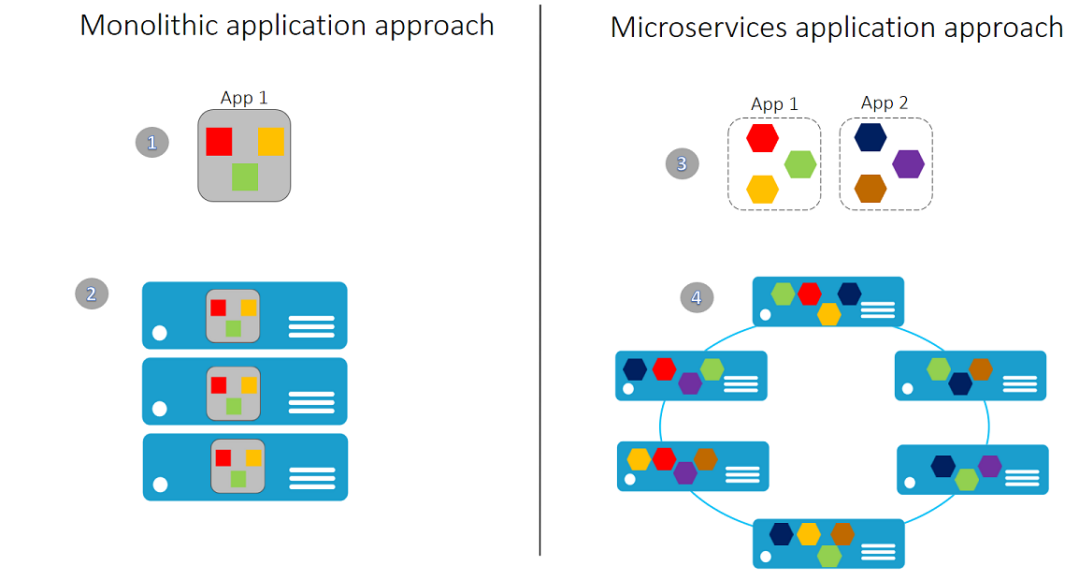
4 Mikro-serwisy

Mikro-serwisy są to małe niezależne serwisy które współpracują ze sobą [18]. Wraz z rozwojem aplikacji stają się one coraz bardziej skomplikowane. Powoduje to wiele problemów, trudności w dodawaniu nowych funkcjonalności, czasochłonne naprawianie błędów czy długi czas wdrożenia nowych programistów w naszą aplikację.

Mikro-serwisy starają się rozwiązać te problemy poprzez podział naszej aplikacji na mniejsze wyspecjalizowane części zgodnie z zasadą Single Responsibility Principle. Jednym z problemów takiego podejścia jest na jak małe części ma zostać podzielona nasza aplikacja, jest wiele odpowiedzi na to pytanie. Jedną z odpowiedzi jest to, że mikro-serwis powinien być wystarczająco mały aby mała grupa programistów mogła go samodzielnie rozwijać. Musimy pamiętać, że wraz ze wzrostem rozdrobnienia naszej aplikacji coraz bardziej uwidaczniają się zalety tej architektury ale również jej wady.

4.1 Zalety

4.1.1 Skalowalność



Ilustracja 9: Skalowalnośc mikroserwisów i monolitycznych aplikacji

https://acom.azurecomcdn.net/80C57D/cdn/mediahandler/docarticles/dpsmedia-prod/azure.microsoft.com/en-us/documentation/articles/service-fabric-overview-microservices/20160805053955/monolithic-vs-micro.png

Jedną z największych zalet mikroserwisów jest ich skalowalność. W przypadku aplikacji monolitycznych zawierających wszystkie nasze funkcjonalności, jeśli jedna z naszych funkcjonalności jest częściej używana to aby poprawić jej wydajność musimy uruchomić nową instancję naszej aplikacji. Spowoduje to zmarnowanie zasobów, ponieważ funkcjonalności, których w tej chwili nie potrzebujemy, również zostaną zeskalowane. Inaczej jest w przypadku mikro-

serwisów, tutaj możemy uruchomić dodatkową instancję tylko tego serwisu, którego funkcjonalność jest częściej używana. Powoduje to ogromne oszczędności zwłaszcza w usługach typu IASS w których płacimy za każdy wykorzystany zasób.

4.1.2 Odporność na zakłócenia

W architekturze mikroserwisowej dopuszczamy taką możliwość że część instancji naszych serwisów może nie działać poprawnie. W takim przypadku pozostałe części przejmują ich zadania, przez co system, jako całość, będzie działał poprawnie. W aplikacjach monolitycznych jest to nie do pomyślenia. Jeśli choć z jednym z komponentów będzie się działo coś niedobrego pozostałe też będą działać niewłaściwie. Weźmy hipotetyczną sytuację awarii fizycznego serwera. Aplikacja monolityczna zostanie natychmiast unieruchomiona i spowoduje to przerwę w dostawie naszych funkcjonalności. Natomiast w aplikacji mikro-serwisowej, zadania zostaną przejęte przez inną instancję tego serwisu położoną na innym serwerze, nasz system nadal jako całość będzie działał poprawnie.

4.1.3 Rozwijanie aplikacji

Rozwijanie aplikacji jest ważną częścią każdego projektu, użycie odpowiedniej architektury bardzo to ułatwia. Dzięki podziałowi aplikacji na mniejsze niezależne elementy, proces samego tworzenia też może zostać podzielony między mniejsze zespoły, które mogą pracować niezależnie. Ułatwia to także później poprawę błędów, ponieważ będą zajmowały się tym osoby, które są wyspecjalizowane w danej funkcjonalności. Jeśli rozwijamy oprócz projektu serwerowego również projekt fornt-endu, istotnym jest istnienie środowiska testowego, z którego będą mogli korzystać programiści odpowiadający za wygląd naszego systemu. W przypadku dużej aplikacji monolitycznej możemy mieć do czynienia z długimi okresami niedostępności serwisu, które są spowodowane wgrywaniem aplikacji. Mikro-serwisy eliminują ten problem, dzięki temu, że wgrywamy tylko tę cześć naszego projektu która się zmieniła. Tę samą strategię może zastosować nasz klient docelowy podczas wgrywania nowej wersji systemu, który w ten sposób uniknie przestojów swojej aplikacji.

4.2 Wady

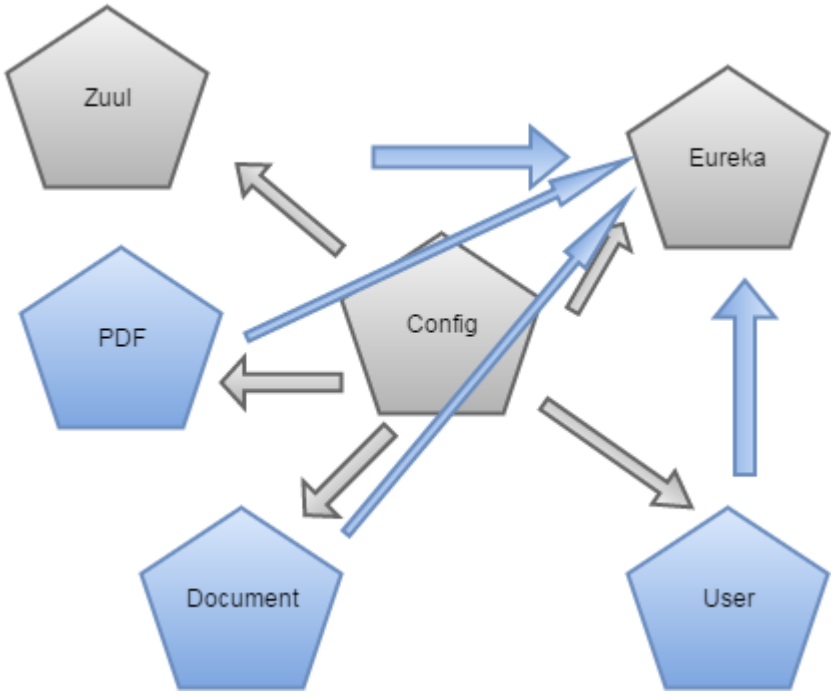
4.2.1 Testowanie

Dużym problemem jest testowanie aplikacji jako całości. Gdy nasz system się rozrośnie urządzenie na którym pracuje programista może mieć niewystarczająca ilość zasobów aby uruchomić wszystkie testy . Taka architektura wyklucza częste testy integracyjne naszego systemu. Może to spowodować brak możliwości wykrycia błędu.

4.2.2 Uruchamianie aplikacji

Dużym problemem dla rozproszonych systemów jest ich uruchamianie. Niektóre elementy muszą być uruchomione wcześniej inne później (jak na przykład z serwerem konfiguracyjnym). Komplikuje to proces wgrywania aplikacji.

5 Architektura aplikacji



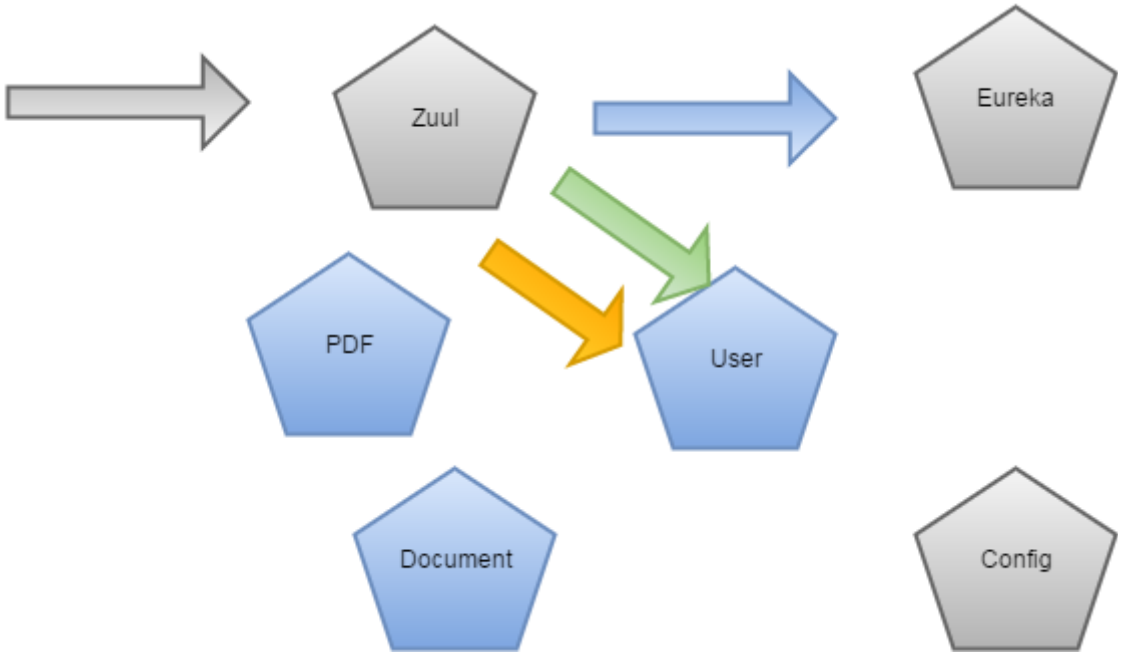
Ilustracja 10: Domyślna komunikacja mikro-serwisów

Cześć serwerowa aplikacji składa się z sześciu mikro-serwisów:

- Zuul – oprócz pełnienia funkcji serwera proxy, autoryzuje on użytkowników i udostępnia Hystrix Dashboard oraz Turbine.
- Eureka – spełnia funkcje rejestrujące inne serwery
- Config – centralne repozytorium konfiguracji
- User – odpowiada za przechowywanie użytkowników, grup oraz zarządzanie nimi
- Document – odpowiada za przechowywanie oraz zarządzanie różnymi typami dokumentów
- PDF – odpowiada za generowanie plików PDF z dokumentów

Ilustracja nr 9 przestawia minimalną liczbę instancji potrzebnych do poprawnego działania systemu. Mikro-serwisy zostały podzielone na dwie kategorie: support (szare) oraz core (niebieskie). Serwery support nie zawierają logiki biznesowej służą tylko i wyłącznie jako dostarczyciele funkcjonalności wynikających z zastosowanej architektury. Serwisy core zawierają funkcjonalności dostępne dla naszego systemu.

Diagram przestawia również domyślną (bez przetwarzania zapytań) komunikacje między mikro-serwisami. Strzałki szare pokazują pobranie konfiguracji podczas startu systemu, natomiast strzałki niebieskie rejestracje serwisu w centralnym rejestrze. Jest ona co jakiś czas ponawiana aby sprawdzić czy serwer dalej działa. Warto zauważyć, że nie wszystkie serwisy się rejestrują w Eureka, serwer konfiguracyjny nie jest rejestrowany. Sama Eureka w tym przypadku również nie rejestruje się u samej siebie przez co jest niewidoczna dla zewnętrznych klientów.



Ilustracja 11: Przetwarzanie zapytanie o listę użytkowników

Obsługa zapytania o listę użytkowników wygląda następująco:

1. Wysłanie zapytania o listę użytkowników do serwera proxy (szara strzałka)
2. Zuul sprawdza czy użytkownik jest zalogowany i ma dostęp do zasobów (zielona strzałka)
3. Zuul pobiera listę aktywnych serwerów (niebieska strzałka)
4. Zapytanie jest przekierowywane do serwera User i tam jest obsługiwane (pomarańczowa strzałka)

5.1 Uruchomienie aplikacji

Aby uruchomić aplikację potrzebujmy narzędzia Docker oraz Docker-compose. Instrukcja jak to zrobić, dla rożnych systemów operacyjnych, znajduje się na stronie https://www.docker.com/products/overview . Instrukcja uruchamiania składa się z dwóch kroków:

1. Uruchomienia serwera konfiguracyjnego:

   docker run –name image-config-server -i wemstar/magisterka-cms-image-config-server
2. Uruchomienie pozostałych części aplikacji, przed uruchomieniem komendy musimy poczekać, aż serwer konfiguracyjny zostanie uruchomiony, następnie polecenie:

   docker-compose up
   spowoduje uruchomienie kolejnych mikro-serwisów.

   Aby sprawdzić czy już wszystko jest uruchomione możemy wejść na serwer Eureka, który znajduje się ona na porcie 8870, adres IP możemy znaleźć wykonując polecenie:

   docker-machine ls

6 Funkcjonalności systemu

6.1 Zarządzanie Użytkownikami

Główna cześć tej funkcjonalności jest zbudowana wokół metod CRUD wygenerowanych na podstawie Spring Data.

```
@RepositoryRestController
public class UserRepositoryImpl {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private ShaPasswordEncoder shaPasswordEncoder;

    @RequestMapping(method = RequestMethod.POST, value = "/user")
    @ResponseStatus(HttpStatus.CREATED)
    public void createUser(@RequestBody UserEntity user) {
        user.setPassword(shaPasswordEncoder
            .encodePassword(user.getPassword(),user.getLogin()));
        userRepository.save(user);
    }
    @RequestMapping(method = RequestMethod.PUT, value = "/user/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateUser(@PathVariable Long id,@RequestBody UserEntity user) {
        UserEntity repoUser = userRepository.findOne(id);
        user.setId(repoUser.getId());
        user.setPassword(shaPasswordEncoder
            .encodePassword(user.getPassword(),user.getLogin()));
        userRepository.save(user);
    }

}
```

Kod 43: Nadpisanie domyślnej implementacji Spring Data Rest

Funkcjac createUser i updateUser nadpisują domyślne funkcje wygenerowanie przez Spring Data Rest, aby to osiągnąć należy:

1. Stworzyć klasę z adnotacją @RepositoryRestController
2. W klasie stworzyć metodę która przyjmie odpowiednie parametry. Na przykład aby nadpisać metodę HTTP PUT musimy stworzyć funkcję która przyjmuje dwa parametry oraz nic nie zwraca. Pierwszy parametr to id obiektu który zmieniamy, drugi to nowy obiekt który nadpisze stary.
3. Ścieżka REST musi pokrywać się z metodą wygenerowaną za pomocą Spring Data Rest. Na przykład metoda PUT ma ścieżkę „/user/id” i taką wartość podajemy w adnotacji @RequestMapping

ShaPasswordEncoder jest to klasa dostarczana przez Spring Security której bean musimy zdefiniować sami, klasa ta udostępnia szereg funkcjonalności związanych z szyfrowanie haseł i jego późniejszą weryfikacją. Użytkownicy są przechowywani w relacyjnej bazie danych H2.

6.2 Zarządzanie Dokumentami

Dokumenty zostały podzielone na trzy typy:

- dowolne – dokumenty składają się z rozdziałów i paragrafów
- szablony – szablony definiujące jak mają wyglądać formularze
- formularze – możemy stworzyć formularz na podstawie wzoru i wypełnić go danymi

Dokumentami można komentować oraz dodawać stopnie weryfikacji. Dokumenty są przechowywane w bazie danych NoSQL MongoDB.

6.2.1 Dokumenty dowolne

Dowolne dokumenty składają się z:

- id – unikalny identyfikator dokumentu
- title -tytuł dokumentu
- chapters -rozdziały dokumentu
- date – data utworzenia
- activites – wszystkie akcje wykonane na dokumencie np. zmiana zawartości, komentarze
- verificationSteps – kroki weryfikacyjne które muszą zostać spełnione aby dokument został zatwierdzony
- allowedUserGrouups -lista grup które mają dostęp do dokumentu

Dokumenty dowolne nie są tak restrykcyjne jak formularze, aby mogły pełnić zadania które trudno przewidzieć lub są tak rzadkie że nie potrzeba tworzenia osobnych funkcjonalności dla nich.

6.2.2 Szablony formularzy

Aby możliwe było wypełnienie formularzy musimy najpierw stworzyć jego szablon.

Szablony formularzy składają się z:

- id - identyfikator szablonu
- title – tytuł szablonu
- fields – pola formularza
- activities - wszystkie akcje wykonane na szablonie np. zmiana zawartości, komentarze
- verificationSteps – kroki weryfikacji formularza
- templateVerificationSteps -kroki weryfikacji szablonu
- allowedUserGroups - lista grup które mają dostęp do szablonu

6.2.3 Formularze

Formularze składają się z:

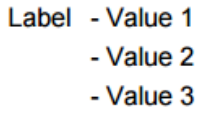
- id – identyfikator formularza
- templateId – id szablonu z jakiego został wygenerowany formularz
- title – tytuł formularza (generowane z szablonu)
- fields – pola formularza wraz z wartościami (nazwy i typy pól są pobierane z szablonu)
- activities - wszystkie akcje wykonane na formularzu np. zmiana zawartości, komentarze
- verificationSteps - kroki weryfikacji formularza (generowane z szablonu)
- allowedUserGroups - lista grup które mają dostęp do formularza

6.3 Generowanie Dokumentów

```
public void createPdf(String dest) throws IOException, DocumentException {
  Document document = new Document();
  PdfWriter.getInstance(document, new FileOutputStream(dest));
  document.open();
  PdfPTable table = new PdfPTable(2);
  table.setTotalWidth(200);
  table.setWidths(new int[]{ 1, 10 });
  table.setHorizontalAlignment(Element.ALIGN_LEFT);
  PdfPCell cell;
  cell = new PdfPCell();
  cell.setBorder(PdfPCell.NO_BORDER);
  cell.addElement(new Paragraph("Label"));
  table.addCell(cell);
  cell = new PdfPCell();
  cell.setBorder(PdfPCell.NO_BORDER);
  List list = new List(List.UNORDERED);
  list.add(new ListItem(new Chunk("Value 1")));
  list.add(new ListItem(new Chunk("Value 2")));
  list.add(new ListItem(new Chunk("Value 3")));
  cell.addElement(list);
  table.addCell(cell);
  document.add(table);
  document.close();
}
```

Kod 44: Przykład iText
http://developers.itextpdf.com/examples/itext5-building-blocks/list-examples

Do generowania dokumentów w formacie PDF służy biblioteka iText. Jest to darmowa biblioteka dla programistów, która umożliwia tworzenie i manipulowanie dokumentami w formacie PDF, z poziomu języka Java [19].



Ilustracja 12: Wynik przykładu

W przykładzie dokument PDF jest tworzony następująco:

1. Tworzymy dokument i plik do którego ma zostać zapisany
2. Otwieramy dokument
3. Tworzymy tabelkę i ustawiamy jej atrybuty
4. Tworzymy pierwsza komórkę do której wstawiamy ciąg znakowy „Label”
5. Tworzymy druga komórkę która zawiera listę elemetów „Value 1”, „Value2”, „Value 3”.
6. Dodajemy tabelkę do dokumentu
7. Zamykamy dokument, jest to ważny krok ponieważ w tym momencie plik jest zapisywany na dysk, dalsze zmiany w obiekcie dokument nie zostaną zatwierdzone.

7 Aplikacja Android

Aplikacja na platformę Adnroid została skompilowana przy pomocy Android SDK w wersji 23 (Android6.0) minimalna wersja to 19 (Android 4.4) co daje nam możliwość jej zainstalowania na większości nowoczesnych telefonów. Aplikacja została podzielona na następujące segmenty:

- activity – zawiera kod w Javie dla widoków
- adapter – zawiera adaptery dla widoków w postaci listy
- rest – zawiera klientów usług REST
- utils – zawiera klasy wspomagające np. dane użytkownika
- view – zawiera implementacje własnych widoków, najczęściej są to komórki tabel

Każda funkcjonalność zaimplementowana składa się z następujących elementów:

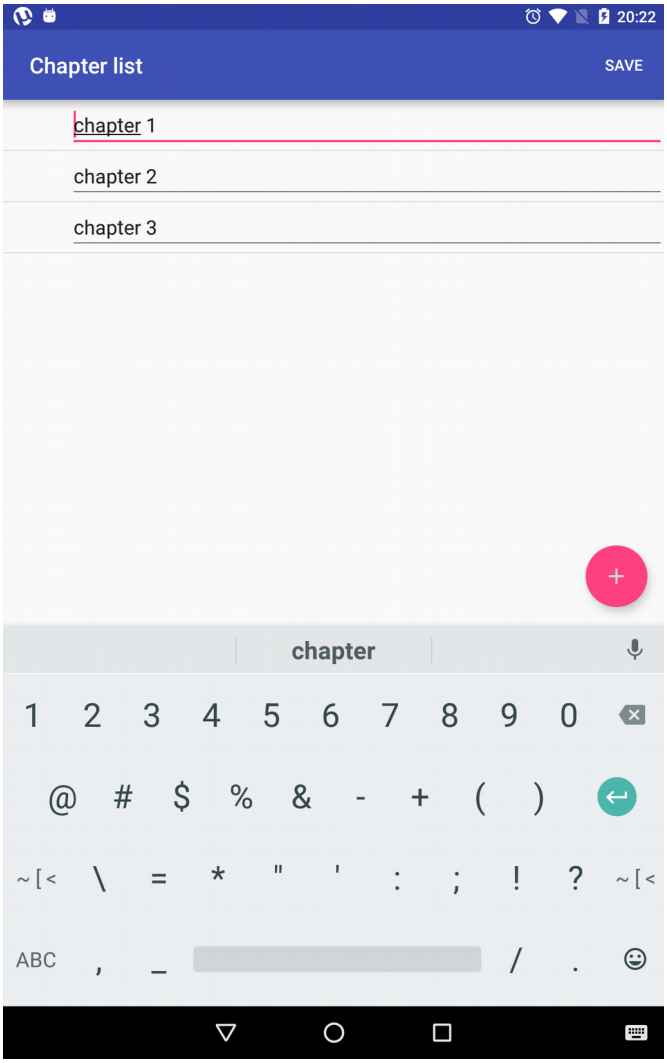
- widok listy – prezentuje wszystkie rekordy dostępne dla użytkownika (np. lista dokumentów), tutaj dodajemy i usuwamy elementy oraz wybieramy te które chcemy edytować
- adapter – odpowiada za pobranie listy rekordów za pomocą klienta REST oraz ich połączenie z odpowiednim widokiem rekordu.
- widok rekordu – widok prezentujący ogólne informacje o rekordzie
- widok szczegółowy – widok prezentujący szczegóły pojedynczego rekordu (np. szczegóły dokumentu), możemy edytować informacje o rekordzie po zapisaniu dane są wysyłane na serwer
- klient REST – przetwarza wszystkie dane potrzebne do działania funkcjonalności np. pobiera listę dokumentów, usuwa dokumenty, tworzy nowe

7.1 Realizacja przykładowej funkcjonalności

Poniższe przykłady obrazują realizacje jednej funkcjonalności w aplikacji Android

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:descendantFocusability="afterDescendants"
        android:focusable="false"></ListView>
    <android.support.design.widget.FloatingActionButton
        android:id="@+id/floatingButton"
        app:layout_anchor="@id/listView"
        app:layout_anchorGravity="bottom|right|end"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|right"
        android:layout_margin="16dp"
        android:src="@drawable/ic_add"
        android:tint="@color/button_material_light" />
</FrameLayout>
```

Kod 45: Przykładowy widok



Ilustracja 13: Widok przykładowej funkcjonalności

Widok tabelki składa się z dwóch elementów Listy oraz Przycisku, wszystko jest zamknięte w FrameLayout. Layouty są to elementy, które zarządzają położeniem elementów wewnątrz siebie, FrameLayout ustawia elementy za pomocą parametru layout_gravity. FloatingActionButton (Przycisk) dzięki temu parametrowi, który jest ustawiony na bottom|right znajdzie się w prawym dolnym rogu ekranu. ListView nie posiada tej zmiennej ponieważ zajmuje cały obszar rodzica, informują nas o tym parametry layout_width oraz layout_height oba ustawione na match_parent. ListView jest to element który wyświetla listę obiektów. FloatingActionButton jest to domyślnie okrągły przycisk który po kliknięciu realizuje główna funkcjonalność widoku. Według MaterialDesign [20] FloatingActionButton powinien zajmować to samo miejsce w całej aplikacji dodatkowo na ekranie nie powinno być więcej takich przycisków niż jeden.

```
@EActivity(R.layout.activity_floating_button_list)
@OptionsMenu(R.menu.save_only_menu)
public class ChapterListActivity extends AppCompatActivity {
    public static final String DOCUMENT_INTENT = "DOCUMENT_INTENT";
    DocumentDTO document;
    private Integer currentChapterPosition;
    @Bean
    ChapterListAdapter adapter;
    @ViewById(R.id.listView)
    ListView listView;
    @AfterViews
    void initTable() {
        document = (DocumentDTO)
getIntent().getSerializableExtra(DOCUMENT_INTENT);
        adapter.setDocument(document);
        listView.setAdapter(adapter);
    }
    @Click(R.id.floatingButton)
    void addChaptersClicked() {
        document.chapters.add(ChapterDTOBuilder.aChapterDTO().build());
        adapter.notifyDataSetChanged();
    }
    @ItemLongClick(R.id.listView)
    public void itemLongClicked(int position) {
        currentChapterPosition = position;
        Intent intent = new Intent(this,ParagraphListActivity_.class);

intent.putExtra(ParagraphListActivity.PARAGRAPH_INTENT,document.chapter
s.get(position));
        startActivityForResult(intent,1);
    }

    @OptionsItem(R.id.action_save)
    void saveDocument() {
        Intent intent = new Intent();
        intent.putExtra(DOCUMENT_INTENT,document);
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

Kod 46: Główne Activity funkcjonalności

Kod 46 zawiera główne Activity funkcjonalności, jest ono dołączone za pomocą adnotacji @Eactivity z widokiem activity_floating_button_list.xml (Kod 45). Menu jest tworzone z pliku save_only_menu.xml o czym informuje nas adnotacja @OptionsMenu. Do instancji zostają wstrzyknięte dwa obiekty:

- ChapterListAdapter (Kod 47) za pomocą adnotacji @Bean
- ListView za pomocą adnotacji @ViewById jest to ten sam obiekt co w Kod 45

Po wstrzyknięciu obiektów oraz inicjalizacji widoków zostaje wywołana metoda initTable(), osiągamy to dodając adnotacje @AfterViews. Metoda wyciąga dokument przesłany z poprzedniego

Activity a następnie umieszcza go w adapterze, sam adapter przypisuje do widoku listy. Metoda addChaptersClicked dzięki adnotacji @Clicked zostanie wywołana po naciśnięciu FloatingActionButton, spowoduje to dodanie kolejnego rozdziału do listy, a następnie powiadomienie adaptera o zmianie zawartości. W przypadku gdy naciśniemy któryś element dłużej zostanie wywołana funkcja itemLongClicked() zapewnia to adnotacja @ItemLongClick. Metoda pozwala na edycję rozdziału dzięki wywołaniu kolejnego Activity (ParagraphListActivity), które to umożliwia. Menu zawiera jeden element „Save”, po jego naciśnięciu zostanie wywołana metoda saveDocument(), prześle ona rozdział z powrotem do Activity, które wywołało ChapterListActivity.

```
@EBean
public class ChapterListAdapter extends BaseAdapter {
    DocumentDTO document;
    @RootContext
    Context context;
    @Override
    public int getCount() {
        return document.chapters.size();
    }
    @Override
    public ChapterDTO getItem(int position) {
        return document.chapters.get(position);
    }
    @Override
    public long getItemId(int position) {
        return position;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ChapterItemEditView chapterItemEditView;
        if (convertView == null) {
            chapterItemEditView = ChapterItemEditView_.build(context);
        } else {
            chapterItemEditView = (ChapterItemEditView) convertView;
        }
        chapterItemEditView.bindChapter(getItem(position));
        return chapterItemEditView;
    }
    public void setDocument(DocumentDTO document) {
        this.document = document;
    }
}
```

Kod 47: Przykład Adaptera

Adapter pozwala na przetłumaczenie listy obiektów na ich widoki, jest niezbędnym elementem widoków ListView oraz podobnych. ChapterListAdapter używa adnotacji @Ebean, dzięki temu dla każdego pola w aplikacji które jest danego typu które posiada adnotacje @Bean, zostanie stworzony obiekt, a następnie wstrzyknięty do tego pola.

Klasa ChapterListAdapter (Kod 47) rozszerza klasę abstrakcyjną BaseAdapter, implementacja musi zawierać następujące metody:

- getCount() - zwraca ilość elementów które pojawią się na liście

- getItem(int position) - zwraca obiekt znajdujący się na danej pozycji
- getItemId (int position) – wraca id obiektu z danej pozycji
- getView(int position, View convertView, ViewGroup parent) – zwraca widok konkretnego elementu. Funkcja ta oprócz pozycji przyjmuje również widok, który możemy dostosować do danej pozycji. Oszczędza to zarówno pamięć jak i czas procesora. Jeśli convertView nie ma wartości (obiekt nie istnieje) wtedy musimy utworzyć obiekt. Następnie do widoku zostaje przypisane odpowiedni element listy, na jego podstawie będzie można uzupełnić zawartość.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal" android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/chapter_title"
        android:layout_alignParentEnd="true"
        android:layout_alignParentStart="true"
        android:layout_marginLeft="60dp" />
</RelativeLayout>
```

Kod 48: Layout widoku elementu listy

Widoki elementów listy powinny być jak najbardziej proste. Widok z Kod 48 zawiera dwa elementy RelativeLayout oraz EditText. RelativeLayout ustawia elementy dzieci obok siebie, w tym przypadku horyzontalnie od lewej do prawej. Cały obszar zajmuje EditText, jest to widok pozwalający na wprowadzanie tekstu.

```
@EViewGroup(R.layout.edit_text_layout)
public class ChapterItemEditView extends RelativeLayout {
    private ChapterDTO chapterDTO;
    public ChapterItemEditView(Context context) {
        super(context);
    }
    @ViewById(R.id.chapter_title)
    EditText chapterTitile;
    public void bindChapter(ChapterDTO chapterDTO) {
        this.chapterDTO = chapterDTO;
        chapterTitile.setText(chapterDTO.name);
    }
    @FocusChange(R.id.chapter_title)
    void focusChangedOnTextView(EditText editText) {
        chapterDTO.name = editText.getText().toString();
    }
}
```

Kod 49: Kod widoku elementu listy

Kod 49 zawiera klasę, która współpracuje z widokiem z Kod 48, informuje nas o tym adnotacja

@EviewGroup. Klasa ChapterItemEditView rozszerza klasę RelativeLayotut musi ona posiadać konstruktor z jednym parametrem, który wymagany jest przez rodzica. Do obiektu zostanie wstrzyknięty widok EditText, następnie za pomocą metody bindChapter do widoku tekstowego zostanie przypisana nazwa rozdziału do wyświetlenia, metoda ta jest wywoływana z poziomu adaptera. W przypadku opuszczenie edycji EditText zostanie wywołana metoda focusChangeOnTextView() osiągamy to za pomocą adnotacji @FocusChange. Metoda ta ma za zadanie przypisać nowy tytuł dla rozdziału na podstawie zawartości widoku EditText.

7.2 Material Design

Material Design jest zbiorem wskazówek jak powinna wyglądać aplikacja aby były intuicyjna dla użytkowników. Został on opracowany przez Google, w celu stworzenie uniwersalnego „języka" opisu wyglądu tak aby był on czytelny i dobrze się prezentował [21]. Material składa się z trójwymiarowego środowiska, światła oraz obiektu. Każdy obiekt ma pozycje x,y,z oraz ma grubość 1. Wymiar z jest osiągany poprzez cienie. Material oprócz samego wyglądu aplikacji definiuje również animacje. Naczelną zasada jest to aby każda animacja coś znaczyła dla użytkownika oraz by wyglądała naturalnie np. obiekty nigdy nie powinny nagle poruszać czy zatrzymywać, preferowanym sposobem jest stopniowe zwiększanie prędkości.

Material Design kładzie duży nacisk na dobór odpowiednich kolorów, możemy wyróżnić trzy najważniejsze kolory w całej aplikacji:

- Primary color - jest to najczęściej używany kolor w aplikacji
- Secondary color – jest to kolor który ma podkreślać ważne informację
- Accent color – jest to kolor używany do zaznaczenie interaktywnych elementów np. FloatingActionButton

Przydatnym narzędziem do generowania odpowiednich kolorów jest Material Palette [22]. Narzędzie to pozwala łatwo dobrać odpowiedni dla naszej aplikacji zestaw barw, udostępnia też podgląd, dzięki czemu możemy zobaczyć jak wybrane kolory będą prezentowały się w naszej aplikacji.

8 Podsumowanie

8.1 Realizacja celów

Celem pracy było stworzenie systemu zarządzania treścią w oparciu o komponenty projektu Spring. W aplikacji zostały wykorzystane najpopularniejsze części projektu Spring czyli Spring MVC Spring AOP, Spring Data oraz Spring Security. Oprócz tego wykorzystano nowe elementy projektu takie jak Spring Cloud oraz Spring Boot. Wszystkie te komponenty doskonale się uzupełniają oraz integrują ze sobą. Spring Boot dodatkowo dba o odpowiednie wersje bibliotek, tak aby nie było między nimi konfliktów, co więcej dzięki wbudowanemu serwerowi Tomcat pozwala na bardzo łatwe uruchamianie aplikacji. Obecnie na rynku nie istnieje rozwiązanie bardziej kompleksowe od projektu Spring, niektóre jego komponenty można zastąpić innymi, ale nie osiągniemy równie dobrych rezultatów.

System jest zbudowany z sześciu mikro-serwisów, trzy z nich dostarczają funkcjonalności biznesowe natomiast pozostałe trzy wynikają z zastosowanej architektury, dostarczają one funkcjonalności potrzebne do działania systemu np. przekierowywanie zapytań. Dzięki zastosowanej architekturze cały system jest łatwo skalowalny. System umożliwia zarządzanie dokumentami, użytkownikami, dodawanie komentarzy do dokumentów, rejestr zmian poszczególnych dokumentów, wielostopniowa weryfikację dokumentów. Do tego jego wielką zaletą jest skalowalność. Rozwiązania obecnie dostępne na rynku posiadają podobne funkcjonalności ale nie są tak łatwo skalowalne. Poprzez zastosowanie architektury mikro-serwisów uzyskaliśmy system, który nie tylko jest skalowalny, ale również łatwo rozwijalny co było jednym z głównych celów pracy.

System posiada testy jednostkowe niektórych systemów. Są one stworzone przy pomocy narzędzia MockMVC, jest to jedna z bibliotek dostarczana przez Spring Boot. Podczas testu uruchamia ona serwer Tomcat, ale tworzy tylko kontroler który testujemy. Jest to bardzo wygodne narzędzie, które znacząco poprawia pokrycie testami aplikacji.

8.2 Możliwości rozszerzenia systemu

System jest zbudowany w architekturze mikro-serwisowej co ułatwia dodawanie nowych funkcjonalności jak i rozszerzanie starych. Nowe funkcjonalności możemy dodać jako nowy mikro-serwis, w pracy istnieje pusty serwis z domyślną konfiguracją który to ułatwia. Możemy również rozbudować już istniejące komponenty, każdy mikro-serwis możemy importować do naszego IDE jako osobny projekt i rozwijać go niezależnie. System można rozszerzyć o interfejs webowy dla użytkowników co poprawi dostępność aplikacji. Dodatkowo można stworzyć moduł który będzie umożliwiał komunikację między użytkownikami. Cały kod systemu jest dostępny w repozytorium pod adresem: https://github.com/wemstar/MagisterkaCMS można dzięki temu stworzyć własną wersje tego systemu.

8.3 Problemy podczas tworzenia systemu

Główne problemy, które wystąpiły podczas tworzenia systemu wynikały z pracy przy pomocy nowej technologi. W chwili obecnej nie ma zbyt wielu źródeł z których można czerpać praktyczną wiedzę. Chociaż dokumentacje do projektu Spring i pochodnych jest bardzo dobrze napisana, nie zawiera pewnych specyficznych przypadków na które, natrafiono podczas realizacji systemu.

Kolejnym problemem są koszty związane z usługami hostującymi aplikację. Duży koszt hostingu powodował, że aplikacja nie mogła być na stale uruchomiona na zdalnym środowisku. Przy rozbudowie aplikacji można pomyśleć o nowym hostingu dla niej, ponieważ przy dużych systemach platforma Google Compute Engine może generować zbyt wysokie koszty.

Ostatnim problemem na który natrafiono jest wykonanie testów integracyjnych. Staje się to widoczne zwłaszcza w mikro-serwisach, które pobierają dane z innych. Wykonanie takiego testu byłoby zbyt czasochłonne oraz wymagałoby dużych zasobów.

8.4 Wnioski

Spring jako projekt przeszedł długą drogę i w znacznej mierze zmienił sposób tworzenia aplikacji przy pomocy Java EE. Choć od pierwszego wydania minęło już 14 lat Spring jest stale unowocześniany i wzbogacany o nowe funkcjonalności. Przykładowo Spring coraz bardziej opiera się na konfiguracji za pomocą adnotacji, podczas gdy na początku swojego istnienia wszystkie ustawienia musiały być w pliku XML, znacząco to zwiększa komfort używania narzędzia. Jednym z nowszych projektów jest Spring Cloud, który w bardzo prosty sposób pozwala na tworzenie systemów opartych o architekturę mikro-serwisową.

W dzisiejszym świecie przetwarzanie danych ma coraz większe znaczenie. Stawia to przed programistami i architektami wiele problemów. Architektura mikro-serwisowa stara się sprostać tym wyzwaniom. Pozwala ona na tworzenie systemów, które są łatwo skalowalne dzięki temu są dostosowane do coraz większego przepływu danych.

9 Zawartość dołączonej płyty CD

- Kod części serwerowej
- Dokumentacja części serwerowej
- Kod części Androidowej
- Dokumentacja części Androidowej
- Filmik prezentujący możliwości aplikacji Androidowej

10 Bibliografia

1: https://pl.wikipedia.org/wiki/Java [Dostęp 23-08-2016]
2: https://en.wikipedia.org/wiki/Gradle [Dostęp 23-08-2016]
3: https://en.wikipedia.org/wiki/Docker_(software) [Dostęp 23-08-2016]
4: https://pl.wikipedia.org/wiki/Android_(system_operacyjny) [Dostęp 23-08-2016]
5: http://androidannotations.org/ [Dostęp 23-08-2016]
6: https://pl.wikipedia.org/wiki/Git_(oprogramowanie) [Dostęp 23-08-2016]
7: https://pl.wikipedia.org/wiki/Infrastructure_as_a_Service [Dostęp 23-08-2016]
8: https://pl.wikipedia.org/wiki/MongoDB [Dostęp 23-08-2016]
9: https://en.wikipedia.org/wiki/H2_(DBMS) [Dostęp 23-08-2016]
10: Craig Walls, Spring in Action, 2014,
11: https://pl.wikipedia.org/wiki/Programowanie_aspektowe [Dostęp 23-08-2016]
12: http://projects.spring.io/spring-security/ [Dostęp 23-08-2016]
13: https://en.wikipedia.org/wiki/Hypertext_Application_Language [Dostęp 23-08-2016]
14: http://projects.spring.io/spring-boot/ [Dostęp 23-08-2016]
15: Craig Walls, Spring Boot in Action, 2015,
16: http://projects.spring.io/spring-cloud/ [Dostęp 23-08-2016]
17: http://projects.spring.io/spring-cloud/spring-cloud.html#_circuit_breaker_hystrix_clients [Dostęp 23-08-2016]
18: Sam Newman, Building Microservices, 2015,
19: https://pl.wikipedia.org/wiki/IText [Dostęp 23-08-2016]
20: https://material.google.com/components/buttons-floating-action-button.html [Dostęp 23-08-2016]
21: https://material.google.com/# [Dostęp 23-08-2016]
22: https://www.materialpalette.com/ [Dostęp 23-08-2016]

11 Indeks ilustracji

12 Indeks zamieszczonych fragmentów kodów aplikacji