POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Jakub Nietupski
Nr albumu 18064

Koncepcja scaffoldingu jako wsparcie w tworzeniu aplikacji internetowych

Praca magisterska napisana
pod kierunkiem:

dra inż. Mariusza Trzaski

Warszawa, luty 2020

Streszczenie

Niniejsza praca dotyczy koncepcji scaffoldingu w tworzeniu aplikacji internetowych w języku Java. Rozpoczyna się od przedstawienia motywacji i sprecyzowania celu projektu. Następnie przechodzi do omówienia pojęcia scaffoldingu i zakresu w jakim jest tu używane. Później przeprowadzony zostaje przegląd stanu sztuki pod kątem narzędzi scaffoldingowych w języku Java.

Na potrzeby pracy stworzono prototypy aplikacji z wykorzystaniem wybranych scaffolderów. Praca zawiera opis przyjętych wymagań dotyczących projektu oraz sprawozdanie z implementacji. Na końcu znajduje się porównanie stworzonych programów m. in. pod względem wydajności, wielkości i łatwości tworzenia.

Spis Treści

1 Wstęp

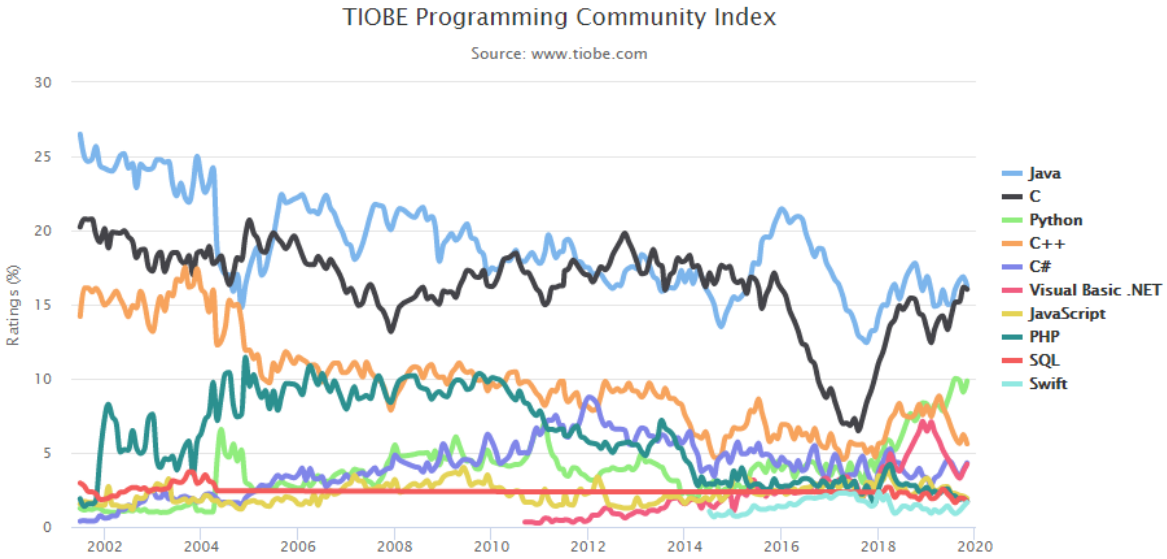
Scaffolding jest pojęciem z dziedziny automatycznej generacji oprogramowania. Zadaniem programów służących do scaffoldingu jest odciążenie programisty poprzez stworzenie za niego części aplikacji. Jest to możliwe, gdyż wiele programów zawiera generyczny, łatwy do zduplikowania kod ze względu na architekturę aplikacji lub mnogość podobnych encji.

Z biegiem czasu powstało wiele tego typu rozwiązań implementujących różne założenia. Niektóre programy za cel stawiają sobie jak największą genetyczność i elastyczność, inne możliwość wygenerowania jak najszerszego zestawu funkcji. Twórca aplikacji, jeśli chce skorzystać ze scaffoldingu w swoim projekcie, powinien wybrać narzędzie najlepiej dopasowane do swoich potrzeb i warunków w jakich powstaje projekt. Liczba oferowanych rozwiązań może być z początku przytłaczająca.

Niniejsza praca stara się wychodzić na przeciw temu problemowi w przypadku pracy nad aplikacjami webowymi. Zawiera teoretyczne omówienie stanu sztuki oraz analizę pewnych praktycznych aspektów związanych w wykorzystaniem scaffoldingu w tworzeniu programu internetowego.

2 Motywacja i cel pracy

Celem niniejszej pracy jest omówienie koncepcji scaffoldingu oraz funkcji jaką pełni w tworzeniu nowoczesnych aplikacji internetowych. Dziedziną, w ramach której przeprowadzono analizę są aplikacje webowe w języku Java. Wybór ten był podyktowany umiejętnościami zawodowymi autora oraz nieustającą od wielu lat dominacją tego języka na rynku (co przedstawia Rysunek 1).



Rysunek 1 Zestawienie popularności języków programowania [1]

Programista w swoim zawodzie powinien zawsze szukać sposobów na zwiększenie swojej produktywności, ponieważ wysokie koszty projektów IT skłaniają klientów i inwestorów do skrupulatnego oraz częstego mierzenia efektywności i przydatności zamawianych projektów (często pojawiającym się wśród ekonomistów zagadnieniem jest pytanie czy informatyzacja w ogóle jest opłacalnym przedsięwzięciem, np: [2]). W związku z tym podjęcie się tego tematu było motywowane chęcią znalezienia sposobów na zmniejszenie powtarzalności zadań jakie musi wykonywać twórca webowych programów biznesowych i w efekcie zwiększenie wydajności.

W odpowiedzi na wyżej wymienione potrzeby na rynku narzędzi dla programistów pojawiają się scaffoldery. Termin scaffolding (ang. rusztowanie) w odniesieniu do wytwarzania oprogramowania może być rozumiany na dwa sposoby. W węższym sensie jest to narzędzie służące do generowania kodu aplikacyjnego na podstawie pewnego modelu danych (np. w postaci DDL). Szerzej jest on rozumiany w ogóle jako techniki generowania kodu [3]. Przedmiotem niniejszej pracy są scaffoldery rozumiane jako narzędzia służące do generowania kodu aplikacji webowej w języku Java na podstawie danego modelu danych. Spośród dostępnych rozwiązań, część z tych które spełniają powyższe wymagania została omówiona w następnym rozdziale.

Na potrzeby porównania wybrano trzy z opisanych narzędzi i wykonano aplikacje testowe na podstawie zestawu wymagań wymienionych w rozdziale „Opis wymagań”. Po zakończeniu implementacji, opisanej w rozdziale „Opis implementacji aplikacji testowych” możliwe było przygotowanie testów i przeprowadzenie porównania. Efekty tych prób oraz wnioski wynikające z korzystania w wybranych technologii zostały opisane w rozdziale „Analiza porównawcza”.

3 Istniejące rozwiązania

Poniższy rozdział ma na celu przegląd dostępnych na rynku rozwiązań dotyczących scaffoldingu aplikacji webowych w języku Java i dostarczenie informacji potrzebnych do wyboru technologii użytych przy implementacji prototypów aplikacji testowych.

3.1 JHipster

JHipster pozwala na generowanie zarówno back- jak i front-endu aplikacji [4]. Backend jest tworzony we frameworku Spring Boot, natomiast w przypadku frontendu dostępne są technologie:

- React
- Angular
- VueJS

oraz wsparcie dla silnika szablonów Thymeleaf. Twórcy narzędzia oferują również możliwość tworzenia projektu w architekturze mikroserwisów oferując integrację z narzędziami do routingu HTTP: Netflix Zuul i Traefik oraz „service discovery": Netflix Eureka i HashiCorp Consul.

Na wejściu do generatora JHipster przyjmuje model danych zdefiniowany w oddzielnym pliku w dedykowanym języku JHipster Domain Language (JDL). Twórcy oprogramowania dostarczyli również narzędzia pozwalające na zaimportowanie diagramu UML w formacie XMI oraz online'owy edytor JDL wizualizujący diagram modelu danych.

Początkowo JHipster oferował interfejs graficzny oraz tekstowy. Obecnie zrezygnowano z prowadzenia wsparcia dla aplikacji okienkowej.

JHipster pozwala na wykorzystanie różnych modułów springowych: Spring Boot, Spring Security, Spring Cache, Spring MVC. Użytkownik wybiera jeden z dwóch dostępnych build systemów: Maven lub Gradle.

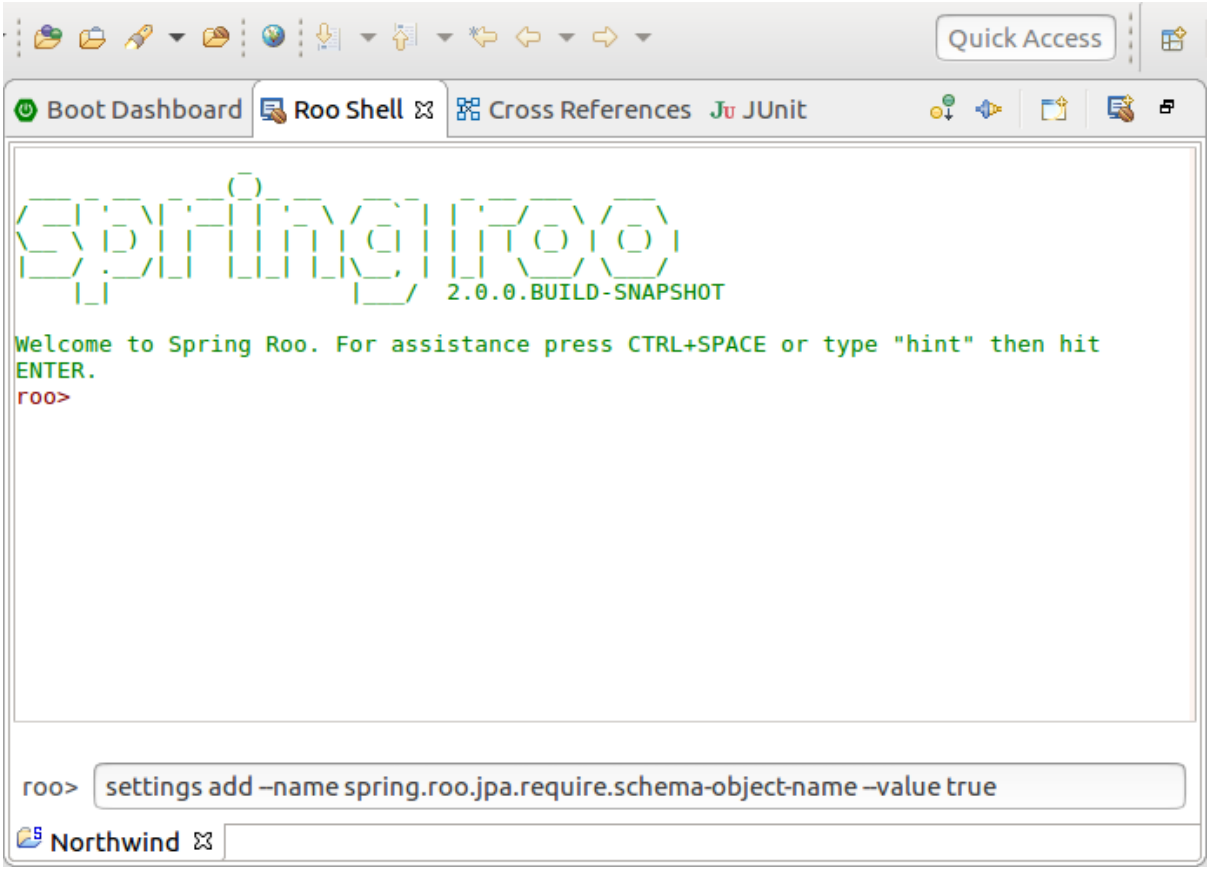
Od strony frontendu generowany jest panel administracyjny pozwalający m.in. na przeglądanie ustawień springowych, logów, statystyk dotyczących aplikacji oraz kontrolerów wygenerowanych na podstawie specyfikacji Open API (również generowanej).

JHipster jest dostępny na zasadach open-source'owej licencji Apache 2.0 [5].

3.2 Spring Roo

Roo należy do grupy projektów springowych, w związku z tym wspomaga generowanie aplikacji webowych w tej właśnie technologii [6]. Jest dostępny jako aplikacja terminalowa lub jako wtyczka do Eclipse. Wspieraną technologią generowania widoku jest Thymeleaf.

Obiekty domenowe są definiowane w Roo'owym shellu (patrz Rysunek 2). W ten sam sposób tworzone są RESTowe lub SOAPowe endpointy. Roo oferuje również automatyczną konfigurację Spring Security.



Rysunek 2 Shell Spring Roo [7]

Od strony zapisu danych Roo pozwala na integrację z Hibernate lub Eclipselink, a wspierane systemy baz danych to:

- DB2 400, DB2 Express C
- Apache Derby
- Firebird
- H2
- Hypersonic
- MySQL
- MsSQL
- Oracle
- PostgreSQL
- Sybas

Program ten jest rozpowszechniany na zasadach open-source'owej licencji Apache 2.0 [8].

3.3 Open Xava

Open Xava to platforma dostępna jako rozszerzenie do Eclipse i jest używana przez interfejs graficzny tego IDE [9].

Przyjmuje ona inne podejście niż reszta narzędzi, ponieważ zamiast generowania kodu projektu, oferuje generowanie aplikacji w runtime'ie. Dotyczy to zarówno front- jak i back-endu aplikacji. Programista musi zdefiniować model obiektów biznesowych. Robi to poprzez napisanie odpowiednich klas javowych (możliwa jest integracja z Lombokiem - narzędziem pozwalającym na zastąpienie często

powtarzających się fragmentów kodu, np. getterów i setterów przez znacznie krótsze adnotacje) i uzupełnienia je poprzez adnotacje określające właściwości widoku, relacje pomiędzy obiektami itp. Listing 1 prezentuje przykładowe wykorzystanie adnotacji `@Stereotype`, na podstawie której Open Xava wygeneruje odpowiednie pola formularza.

Listing 1 Przykład adnotacji Open Xava [10]

```
private class Demo {
    @Stereotype("MONEY")
    private BigDecimal price;

    @Stereotype("PHOTO")
    @Column(length=16777216)
    private byte [] photo;

    @Stereotype("IMAGES_GALLERY")
    @Column(length=32)
    private String morePhotos;

    @Stereotype("MEMO")
    private String remarks;
}
```

Nietypowe widoki można zaimplementować własnoręcznie, używając JavaScrptu i HTMLa lub JSP.

Projekt stworzony przy użyciu Open Xava integruje się z Hibernatem i dowolną bazą danych z nim współpracującą. Twórcy nie wspierają integracji z funkcjonalnościami Springowymi takimi jak wstrzykiwanie springowych beanów itp. Wersja premium tego oprogramowania udostępnia zarządzanie użytkownikami i kontrolą poziomu dostępów dla poszczególnych użytkowników w generowanym systemie.

W wersji podstawowej Open Xava jest wolnym oprogramowaniem dostępnym na zasadach licencji GNU LGPL 2.1 [11].

3.4 jOOQ

jOOQ powstało jako biblioteka javowa służąca do budowania zapytań SQL, ale posiada też funkcję generowania kodu [12]. Narzędzie nie udostępnia interaktywnego interfejsu, konfigurację umieszcza się w dedykowanym pliku a samą generację można uruchamiać automatycznie przy procesie kompilacji w Maven lub Gradle.

jOOQ nie jest związany z żadną inną biblioteką ani frameworkiem, kod generowany jest w czystej Javie, opcjonalnie z dodanymi adnotacjami do walidacji (JSR-303) oraz JPA (javax.persistence).

Podstawą do generacji jest schemat bazy danych. W wersji open source wspierane są następujące bazy:

- CUBRID
- Derby
- Firebird

- H2
- HSQLDB
- MariaDB
- MySQL
- PostgreSQL
- SQLite

Generowane są klasy odzwierciedlające strukturę bazy, a więc w szczególności tabele i rekordy tabel, oraz obiekty modelowe i DAO (data access object). Nie wspierana jest natomiast generacja serwisów i kontrolerów ani żadnego widoku.

Twórcy udostępniają darmową wersję tego oprogramowania na licencji Apache 2.0 [13].

3.5 Jspresso

Jspresso jest kolejnym frameworkiem oferującym generowanie aplikacji biznesowej w javie w oparciu o model danych [14]. Generowany projekt wykorzystuje Springa do generowania backendu aplikacji, jest więc z nim kompatybilny. Jego twórcy wśród wyróżniających projekt zalet wymieniają dependency injection oraz zaawansowane wsparcie dla obsługi bezpieczeństwa (autoryzacja i autentykacja) i internacjonalizacji.

Model danych jest definiowany w autorskim języku opartym o Groovy - Sugar for Jspresso (SJS). Definiuje się w nim obiekty i relacje. Listing 2 prezentuje przykład użycia SJS. Istnieje także pewna ograniczona możliwość zdefiniowania określonych akcji biznesowych z pomocą tego języka. Na tej podstawie generowane są obiekty modelowe, serwisy oraz poziom dostępu do danych. Możliwe jest też zastąpienie SJS przez Spring XML, który był poprzednim domyślnym sposobem konfiguracji projektu.

Listing 2 Przykład definiowania encji w SJS [15]

```
Entity('Employee', extend: 'Nameable',
        processor:'EmployeePropertyProcessors',
        uncloned:['ssn'], icon:'male-48x48.png') {
    string_64 'firstName', mandatory:true,
            processors:'FirstNameProcessor'
    date 'birthDate'
    string_10 'ssn', regex:'[\\d]{10}', regexSample: '0123456789',
            reference 'company', ref: 'Company',
    mandatory: true, reverse:'Company-employees'
    set 'teams', ref: 'Team'
}
```

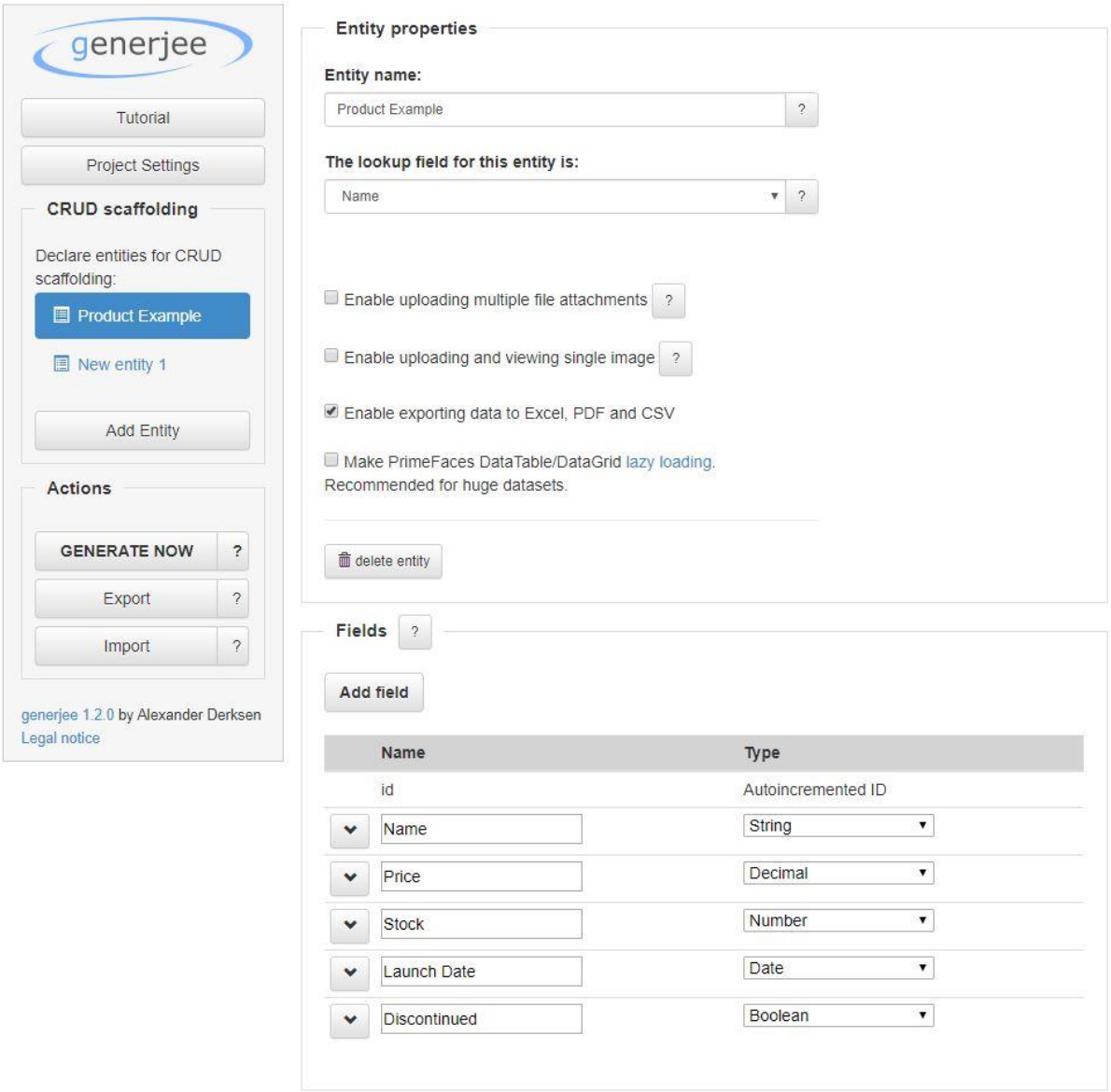
Domyślnie do mapowania obiektowo-relacyjnego wykorzystywany jest Hibernate. Twórcy narzędzia nie wymieniają listy współpracujących systemów bazodanowych ale można się spodziewać że warunkiem jest tu kompatybilność z Hibernatem.

Jspresso umożliwia także generowanie frontendu aplikacji. Poszczególne widoki również są definiowane za pomocą SJS, a dostępnymi technologiami docelowymi są: HTML+JS, Flex lub Java Swing.

Jspresso jest dostępne na licencji wolnego oprogramowania GNU LGPL 3.0 [16].

3.6 Generjee

Generjee prezentuje inne podejście niż w przypadku dotychczas opisanych scaffolderów. Z narzędzia korzysta się poprzez aplikację webową dostępną na stronie projektu [17]. Aplikacja oferuje interfejs graficzny (patrz Rysunek 3) w którym definiujemy model danych oraz parametry projektu. Po wybraniu wszystkich opcji rozpoczyna się pobieranie wygenerowanej struktury projektu z gotową konfiguracją w Maven. W ten sposób uzyskujemy kod zarówno frontendu w Java Server Faces z PrimeFaces, jak i backendu w czystej Javie EE.



Rysunek 3 Interfejs webowy prgoramu Generjee [17]

Mimo generowanego frontendu możliwe jest także automatyczne wystawienie RESTowych endpointów z podstawową funkcjonalnością CRUD (create, read, update i delete). Twórcy umożliwiają wygenerowanie typowych operacji dotyczących zarządzania danymi takich jak sortowanie, filtrowanie i eksport danych, operacji związanych z zarządzaniem użytkownikami, rejestracją nowych użytkowników i dostępami dla poszczególnych ról (używaną do tego biblioteką jest Apache Shiro), a także zarządzanie internacjonalizacją czy uploadem plików.

Wraz z wygenerowanym projektem dostarczane są metadane w formacie JSON, które pozwalają na ponowne zaimportowanie projektu do aplikacji Generjee i zmianę jej parametrów.

Tworzona w generatorze aplikacja używa JPA do dostępu do danych i jest skonfigurowana z bazą H2. Jeśli programista chce zintegrować projekt z bazą danych która rzeczywiście zapisuje rekordy na dysku twardym, musi to zrobić samodzielnie.

3.7 Simple Scaffolding

Celem przyświecającym stworzeniu Simple Scaffolding było udostępnienie lekkiego, łatwego w utrzymaniu narzędzia, które nie narzuca konkretnych rozwiązań architektonicznych [18].

Założeniem tego projektu nie jest generowanie kodu aplikacyjnego “od zera” na podstawie odgórnie zadanych parametrów. Jego zadaniem jest raczej ekstrakcja możliwych do ponownego wykorzystania szablonów na podstawie już istniejącej części projektu. Ma on być pomocny w momencie gdy jest już zaimplementowana jakaś architektura i funkcjonalności dla pewnej części obiektów modelu i chcemy dodać kolejne według ustalonego schematu.

Projekt nie posiada żadnego interfejsu, jest dostępny w postaci prostej klasy, którą trzeba umieścić w folderze `src/test/java` i pliku konfiguracyjnego w formacie JSON. Program należy uruchomić w dwóch następujących po sobie trybach: “read” i “write”. W trybie “read”, przed uruchomieniem, w pliku konfiguracyjnym należy umieścić nazwy encji, dla których już została zaimplementowana wzorcowa funkcjonalność (np. “User”). Dzięki temu po uruchomieniu Simple Scaffolding potraktuje wszystkie znalezione klasy zawierające nazwę podanego obiektu (np. “GetUserService” itp.) jako szablony. Następnie, przed ponownym uruchomieniem programu należy w pliku konfiguracyjnym podać nazwy nowych obiektów dla których klasy chcemy stworzyć. Dzięki temu Simple Scaffolding utworzy klasy wg. takiej struktury jak dla zapisanych wcześniej obiektów szablonowych.

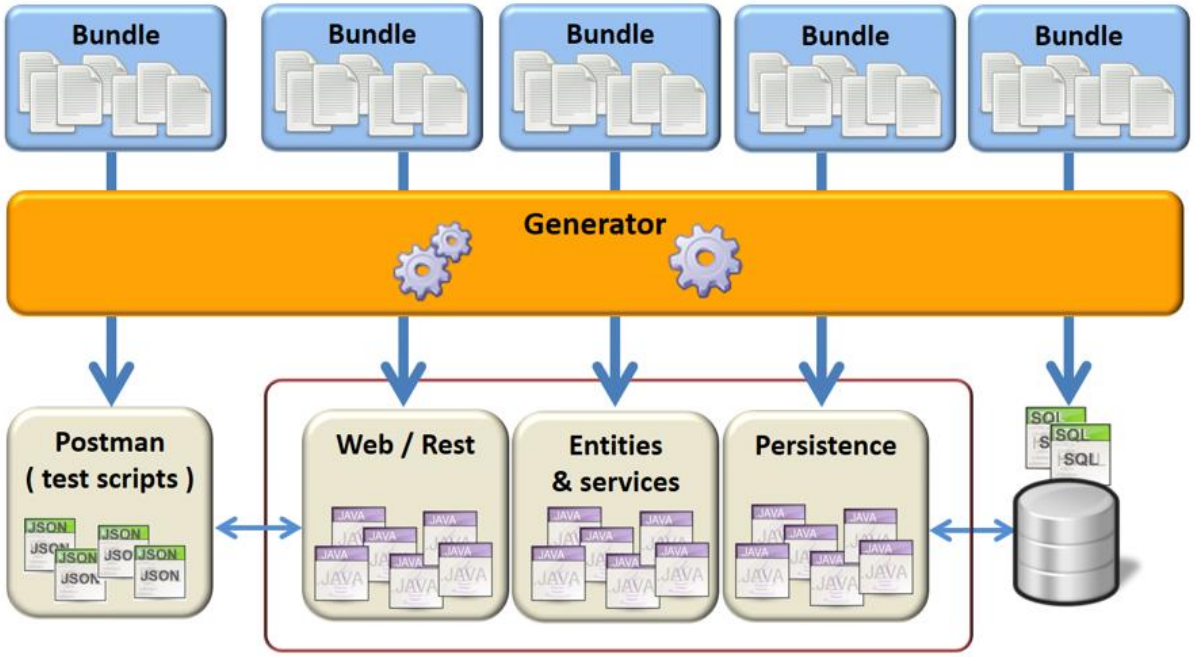
Ze względu na swoje założenia, Simple Scaffolding nie oferuje żadnej integracji z systemami baz danych ani innymi zewnętrznymi bibliotekami. Nie wspiera też w żaden sposób rozwoju frontendu aplikacji. Wszystkie te rzeczy leżą w gestii programisty.

Oprogramowanie to jest udostępnione na zasadach wolnej licencji Expat [19].

3.8 Telosys

Telosys jest dostępne zarówno jako narzędzie z interfejsem tekstowym, obsługiwanym z wiersza poleceń, jak i wtyczka do Eclipse [20]. Twórcy oferują także szereg wtyczek do najpopularniejszych edytorów tekstowych, takich jak Atom czy Visual Studio Code, wspomagających współpracę z programem terminalowym.

Założeniem tego projektu była duża generyczność, w związku z tym Telosys ma docelowo wspomagać tworzenie aplikacji w różnych językach i frameworkach, zarówno na frontendzie jak i backendzie. Narzędzie to działa na zasadzie procesora szablonów opartego na Apache Velocity. Kluczowym dla pracy z tym silnikiem jest repozytorium szablonów (patrz Rysunek 4). Znajdują się w nim wzorce dla generowanych projektów - backendowych, w tym Java EE, Spring MVC itp. i frontendowych, takich jak AngularJS. Szablony te są agnostyczne względem modelu danych i logiki biznesowej.



Rysunek 4 Schemat działania generatora Telosys. Różne, niezależne szablony są odpowiedzialne za poszczególne warstwy aplikacji [21]

Dane specyficzne dla projektu, to znaczy definicje obiektów domenowych można dostarczyć do Telosys na dwa sposoby. Pierwszym z nich jest zdefiniowanie encji i relacji w dedykowanym DSL (Domain Specific Language). Na podstawie plików zapisanych w tym języku, Telosys wczytuje dane i łącząc je z informacją o architekturze aplikacji, generuje kod projektu. Drugi sposób na załadowanie danych może być bardziej przydatny w przypadku gdy mamy już gotową relacyjną bazę danych ze zdefiniowanym modelem. W takim wypadku wystarczy skonfigurować połączenie i Telosys sam wczyta dane łącząc się z bazą.

Z tego narzędzia można korzystać na zasadach wolnej licencji LGPL 3.0 [22].

3.9 JBoss Forge UI Scaffolding

UI Scaffolding jest narzędziem należącym do zestawu funkcjonalności pakietu JBoss Forge - bazującego na Eclipse systemu wspierającego rozwój aplikacji Java EE [23]. Posiada on interfejs tekstowy i generuje aplikacje w czystej Java EE oraz Java Server Faces. Program wspiera generowanie widoków i funkcjonalności dla tworzenia, update'owania, usuwania, paginacji i szukania danych.

Praca z tym narzędziem polega na wpisywaniu odpowiednich komend w wiersz poleceń, zamiast wczytywania jednolitego pliku konfiguracyjnego. W ten sposób definiuje się encje, pola i relacje między obiektami. Możliwe jest tworzenie relacji jeden-do-wielu, wiele-do-wielu i jeden-do-jednego.

Narzędzie nie zapewnia automatycznej konfiguracji ani populacji bazy danych.

JBoss Forge jest rozpowszechniany na zasadach otwartej licencji Eclipse Public License 1.0 [24].

3.10 Wnioski z analizy stanu sztuki

Na podstawie analizy dostępnych na rynku rozwiązań można stwierdzić, że twórcy oprogramowania zajmują się zagadnieniem scaffoldingu dość intensywnie i od dłuższego czasu. Z

jednej strony fakt ten można tłumaczyć chęcią przyspieszenia pracy swojej i innych programistów - w przypadku niezależnych, generycznych projektów jak np. Simple Scaffolding. Z drugiej strony rozwijanie scaffoldera może mieć na celu promowanie swojej własnej technologii - im łatwiej i szybciej można stworzyć projekt z wykorzystaniem danego produktu, tym większa szansa że programiści z niego skorzystają. Tu przykładem może być jOOQ, dla którego scaffolding nie jest podstawową funkcjonalnością (narzędzie powstało jako wsparcie dla budowania zapytań do bazy danych z poziomu Javy [25]).

Opisywane narzędzia możemy podzielić według pięciu kryteriów: rodzaju interfejsu, sposobu dystrybucji, stopnia generyczności, wejścia danych oraz zakresu działania.

1. Ze względu na interfejs scaffoldery dzielą się na trzy kategorie:
    a. interfejs graficzny
        i. Open Xava
        ii. Generjee
        iii. Telosys
    b. interfejs tekstowy
        i. JHipster
        ii. Spring Roo
        iii. Telosys
        iv. UI Scaffolding
    c. brak interfejsu - aplikacja nie jest interaktywna
        i. jOOQ
        ii. Jspresso
        iii. Simple Scaffolding
2. Ze względu na sposób dystrybucji:
    a. niezależne aplikacje
        i. JHipster
        ii. Spring Roo
        iii. Generjee
        iv. Telosys
        v. UI Scaffolding
    b. wtyczki do IDE
        i. Spring Roo
        ii. Open Xava
        iii. Telosys
    c. biblioteki dostępne w repozytorium danego build-systemu np. archetypy Mavenowe
        i. jOOQ
        ii. Jspresso
    d. klasy umieszczane w projekcie
        i. Simple Scaffolding
3. Ze względu na stopień generyczności wyróżniamy projekty
    a. od narzędzi generujących kod pod konkretny framework
        i. Spring Roo
        ii. Open Xava
        iii. Jspresso
    b. lub w czystej Javie
        i. Generjee

ii. UI Scaffolding
iii. jOOQ

c. przez narzędzia wspierające pewien wachlarz technologii, jednakże odgórnie narzucony przez twórców scaffoldera
   i. JHipster

d. po w pełni generyczne narzędzia
   i. Simple Scaffolding
   ii. Telosys

4. Ze względu na sposób wprowadzania modelu danych mamy narzędzia:
   a. przyjmujące dane w dedykowanym domenowym języku
      i. JHipster
      ii. Open Xava[1]
      iii. Jspresso
      iv. Simple Scaffolding[2]
      v. Telosys
   b. w których model jest definiowany obiekt po obiekcie w wierszu poleceń/oknie aplikacji
      i. JHipster
      ii. Spring Roo
      iii. Generjee
      iv. UI Scaffolding
   c. pobierające model danych z bazy danych
      i. jOOQ
      ii. Telosys
5. Ze względu na zakres działania:
   a. generujące tylko backend
      i. jOOQ
      ii. Simple Scaffolding
   b. dostarczające również frontend
      i. JHipster
      ii. Spring Roo
      iii. Open Xava
      iv. Jspresso
      v. Generjee
      vi. Telosys
      vii. UI Scaffolding

Tabela 1 zawiera zestawienie niektórych cech generatorów.

[1] W przypadku Open Xava językiem definiowania obiektów domenowych jest Java

[2] W Simple Scaffolding de facto nie opisuje się obiektów modelowych, szablony są generowane na podstawie istniejącego kodu w Javie

Tabela 1 Zestawienie wybranych cech omawianych scaffolderów

Program	UI	Backend	Frontend	Połączenie z bazą danych
JHipster	NIE	TAK	TAK	NIE
Spring Roo	NIE	TAK	TAK	NIE
Open Xava	TAK	TAK	TAK	NIE
jOOQ	NIE	TAK	NIE	TAK
Jspresso	NIE	TAK	TAK	NIE
Generjee	TAK	TAK	TAK	NIE
Simple Scaffolding	NIE	TAK	NIE	NIE
Telosys	TAK	TAK	TAK	TAK
UI Scaffolding	NIE	TAK	TAK	NIE

3.11 Wybór technologii do prowadzenia porównania

Zgodnie z założeniami pracy zaimplementowano 3 projekty testowe. Każdy z tych projektów spełnia te same wymagania, ale każdy jest wykonany z pomocą innego scaffoldera. Na ich podstawie przeprowadzono dalszą analizę. Aby przygotowane porównanie dawało możliwie szeroki obraz omawianego zagadnienia, wybrane narzędzia powinny reprezentować różniące się między sobą podejścia do generowania kodu. W związku z tym stworzono projekty w następujących technologiach:

1. JHipster, ponieważ jest to najbardziej rozbudowane narzędzie pozwalające na generowanie kodu na podstawie modelu danych opisanego w dedykowanym DSL
2. jOOQ, jako przykład narzędzia pobierającego model danych z bazy danych
3. Simple Scaffolding przedstawiający podejście najbardziej generyczne

4 Użyte technologie

W celu przeprowadzenia praktycznego porównania wybranych technologii scaffoldingowych zaplanowano zaimplementowanie trzech aplikacji na podstawie określonego zestawu wymagań. Poniższy rozdział zawiera przegląd technologii w których powstały aplikacje testowe.

4.1 Języki programowania

Zgodnie z założeniami projektu implementację backendu wykonano w języku Java, natomiast frontend powstał w JavaScripcie.

4.1.1 Java

Java jest językiem programowania ogólnego przeznaczenia, który powstał w 1995 roku [26]. Założeniem przyjętym przez twórców tego języka było [27]:

- zapewnienie wieloplatformowości poprzez wykonywanie kodu na dedykowanej Maszynie Wirtualnej Javy (JVM – Java Virtual Machine)
- wykorzystanie paradygmatu obiektowego
- bezpieczeństwo
- niezależność od architektury
- wydajność

Programiści znaleźli dla Javy szeroki wachlarz zastosowań, od aplikacji webowych (takich jak te, które powstały w ramach niniejszej pracy), po oprogramowanie urządzeń mobilnych [28]. Jej popularność sprawia, że programiści mogą z łatwością uzyskać pomoc oraz znaleźć narzędzia stworzone z użyciem/na potrzeby tego języka.

4.1.2 JavaScript

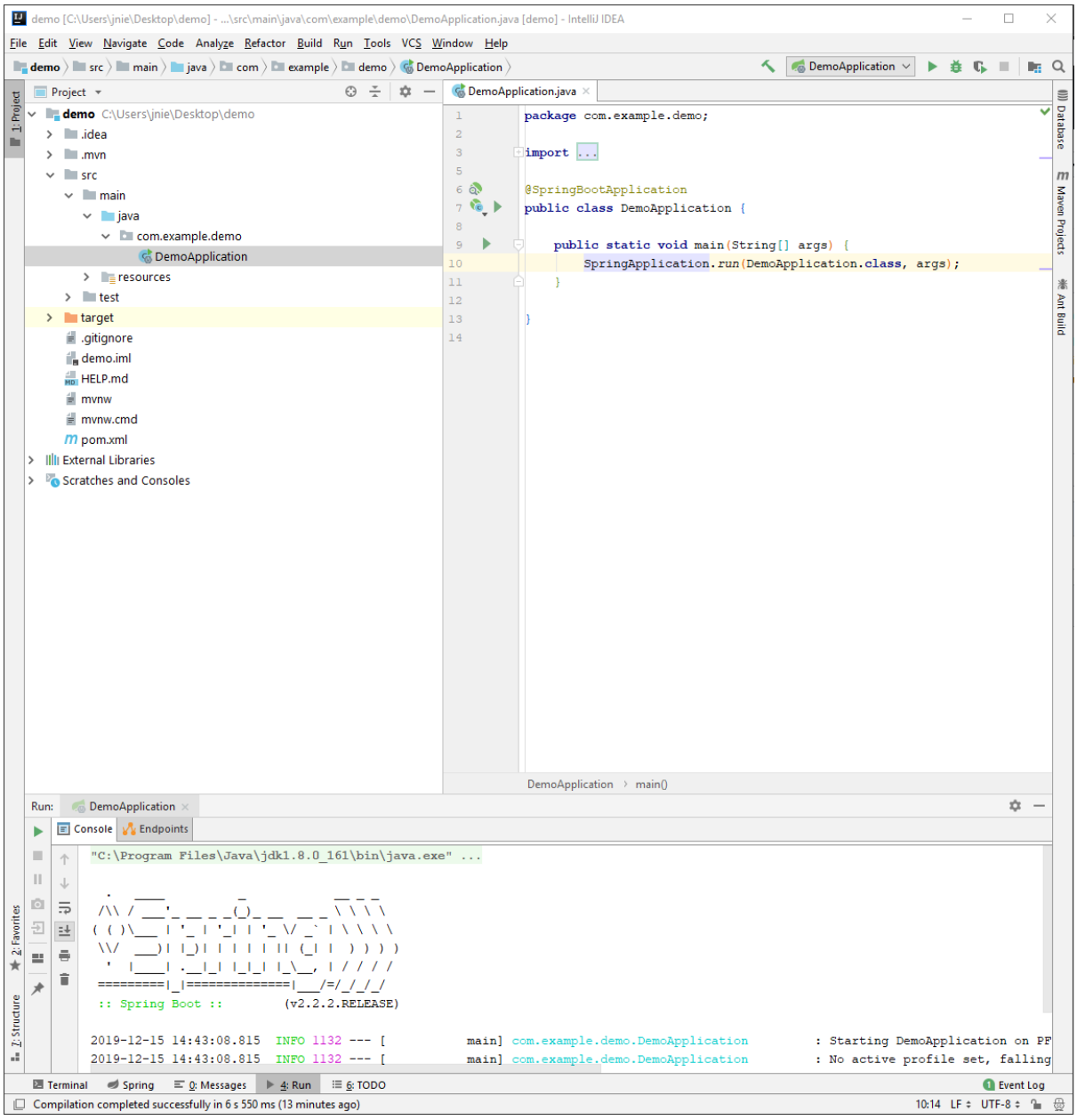
Język ten powstał w latach dziewięćdziesiątych [29] w celu umożliwienia dodawania skryptów na interaktywnych stronach HTML. Obecnie jest on bardzo szeroko stosowany w różnych technologiach internetowych, również w aplikacjach backendowych [30]. W miarę z upowszechnieniem się tego języka podjęto się jego standaryzacji i dziś funkcjonuje on pod oficjalną nazwą ECMAScript [31]. Głównymi cechami tego języka są:

- strukturalny paradygmat programowania, aczkolwiek wraz z rozwojem standardu wprowadzane są kolejne elementy obiektowości
- jest językiem interpretowanym (a nie kompilowanym)
- słabe typowanie, co oznacza, że pewne typy mogą być dedukowane implicite w na podstawie wykonywanej
- uniwersalność i wieloplatformowość, które są zapewniane przez przeglądarki implementujące standard ECMAScript

4.2 Środowisko programistyczne

Cały kod stworzony na potrzeby tej pracy został napisany w Zintegrowanym Środowisku Programistycznym (ang. Integrated Development Environment – IDE) IntelliJ IDEA firmy JetBrains

[32]. Oprogramowanie to posiada szereg narzędzi ułatwiających pracę programiście, takich jak: uzupełnianie kodu (interaktywne podpowiedzi w trakcie pisania), przeszukiwanie klas i metod, sprawdzanie stylu formatowania, integracja z systemem kontroli wersji, czy automatyczne rozwiązywanie błędów. Posiada również wbudowane wsparcie dla tworzenia aplikacji z użyciem frameworku Spring. Program ten jest dostępny między innymi na licencji otwartego oprogramowania Apache 2 [33]. Rysunek 5 przedstawia przykładowy projekt otwarty w programie IntelliJ.

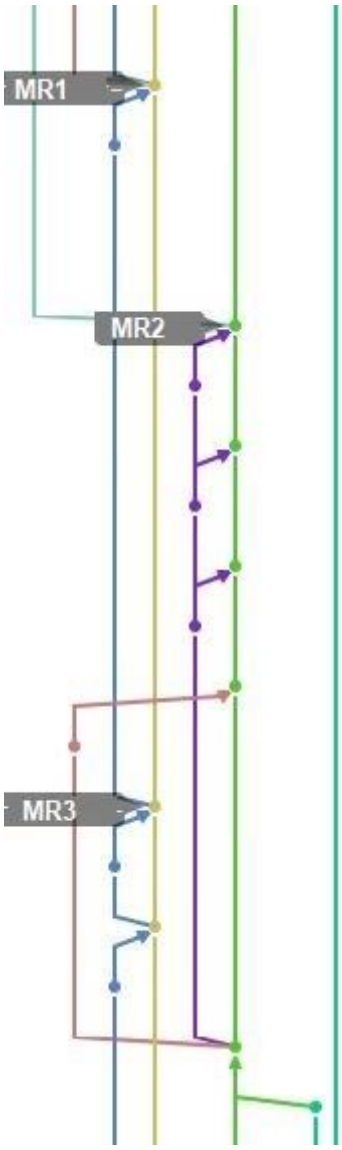


Rysunek 5 Przykładowy widok aplikacji IntelliJ IDEA

4.3 Repozytorium

Repozytorium kodu, które zostało wykorzystane przy rozwoju aplikacji testowych jest Git. Zostało ono stworzone przez twórcę jądra Linuxa – Linusa Thorvaldsa w 2005 r. [34] Pozwala na śledzenie zmian w kodzie i ułatwia współpracę w zespole dzięki możliwości tworzenia gałęzi (ang. branch) na których programiści mogą pracować niezależnie. Cechą, która wyróżnia ten system jest możliwość pracy rozproszonej – każdy użytkownik posiada na swojej maszynie lokalną wersję repozytorium i w

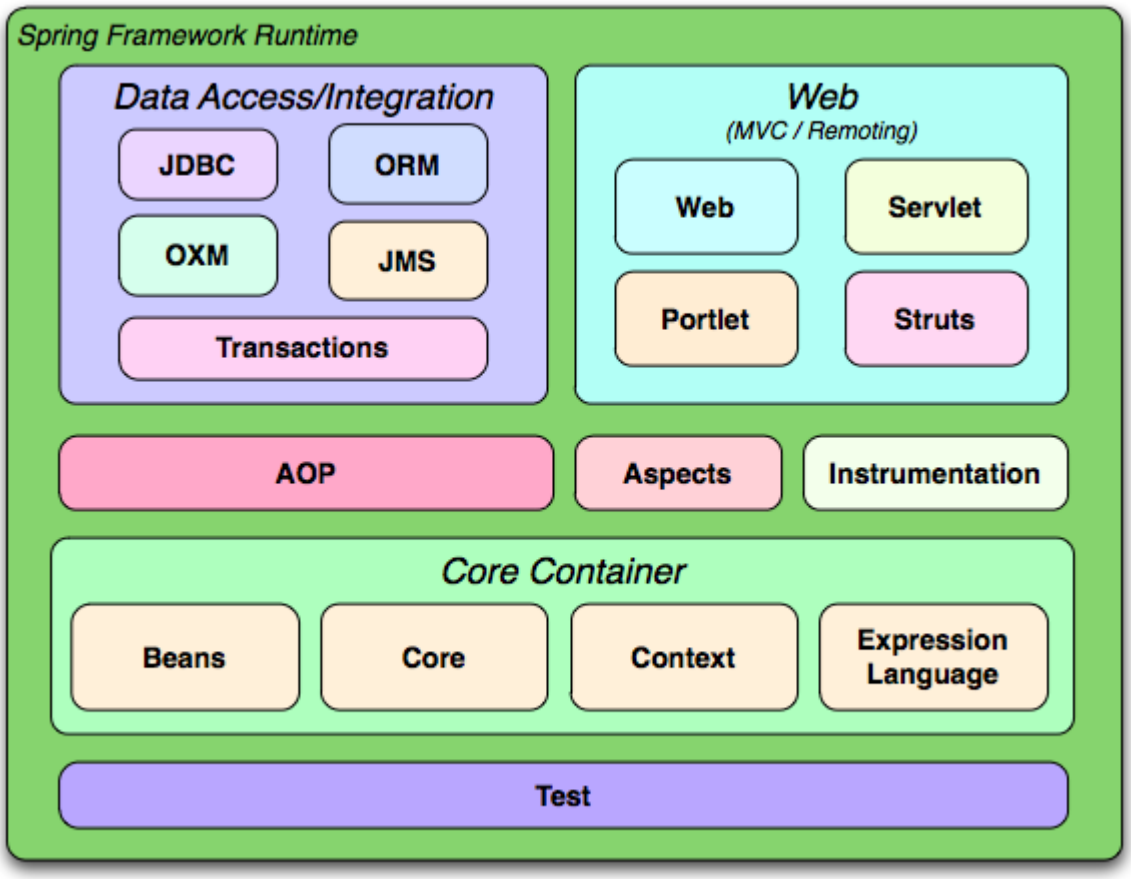
razie potrzeby synchronizuje jego stan z serwerem. Rysunek 6 przedstawia fragment przykładowego grafu rozgałęzień dla bardziej zaawansowanego projektu programistycznego. Linie prezentują poszczególne „gałęzie", kropki oznaczają pojedyncze paczki zmian, które zostały zatwierdzone przez programistę (ang. commit).



Rysunek 6 Fragment grafu repozytorium Git

4.4 Spring Boot

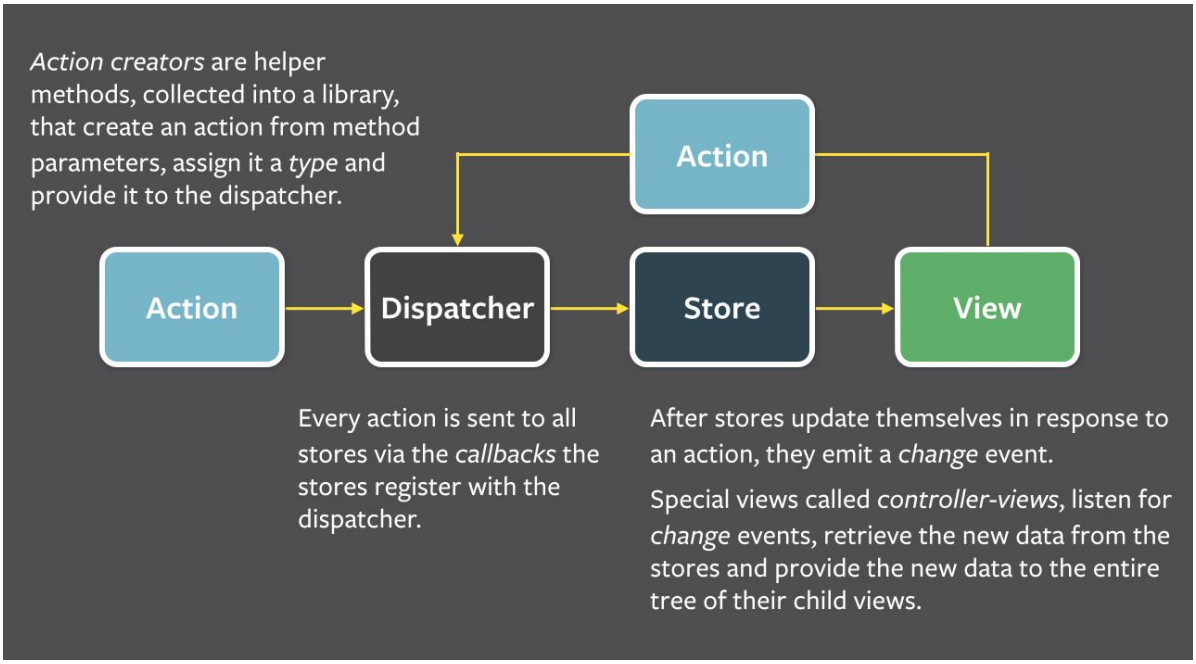
Wszystkie aplikacje backendowe zostały wykonane z użyciem technologii Spring Boot. Spring jest frameworkiem wykorzystywanym przy tworzeniu nowoczesnych aplikacji webowych w języku Java. Jego kluczowym zadaniem jest dostarczenie powtarzalnej struktury programu, sięgającej od poziomu dostępu do danych po widoki. Dzięki temu programiści korzystający ze Springa mogą skupić się na implementacji logiki biznesowej [35]. Spring Boot pozwala na szybką konfigurację projektu i integrację różnych bibliotek (Rysunek 7) z grupy Spring [36].



Rysunek 7 Moduły frameworku Spring [37]

4.5 React

Biblioteką, która została wykorzystana do stworzenia części frontendowej, jest React. Przy pracy z tym narzędziem programista tworzy kod w języku JavaScript, lub jego silnie typowanego odpowiednika – TypeScript. Strony tworzone w Reakcie składają się z niezależnych komponentów, które same zarządzają swoim stanem [38]. Nietypowym aspektem pisania aplikacji z użyciem Reacta jest możliwość użycia architektury aplikacyjnej Flux, opracowanej przez programistów z Facebooka. Główną cechą tego podejścia jest jednokierunkowy przepływ danych wewnątrz aplikacji (patrz Rysunek 8) co pozwala utrzymać czytelność i efektywność wykorzystania Reactowych komponentów [39].



Rysunek 8 Architektura Flux [39]

Z uwagi na fakt, że niniejsza praca dotyczy scaffolderów dla Javy, a z pośród narzędzi scaffoldingowych wykorzystanych do implementacji tylko jeden pozwala na generację frontendu aplikacji, dla dwóch pozostałych projektów napisano jedną, wspólną część odpowiedzialną za widoki. Listing 3 prezentuje przykładowy wygląd prostego komponentu Reactowego.

Listing 3 Hello World w bibliotece React

```
class HelloWorld extends React.Component {
    render() {
        return (
            <div>
                <h1>
                    Hello world and
                    {this.props.name}!
                </h1>
            </div>
        );
    }
}
// metoda render() odpowiada za
// ostateczny wygląd komonentu
// Przykład użycia: <HelloWorld name="Jakub" />
```

4.6 Docker

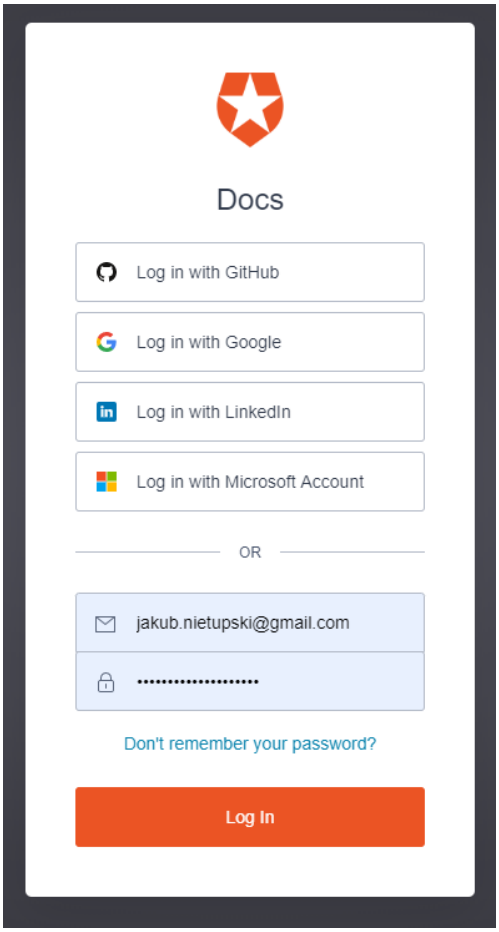
Bazy danych, z których korzystały aplikacje testowe zostały utworzone wewnątrz kontenerów dockerowych. Docker jest jedną z popularniejszych technologii zapewniających konteneryzację. Celem tego rodzaju oprogramowania jest ułatwienie konfiguracji środowiska poprzez konfigurację wielu małych, zamkniętych środowisk. Każde z tych środowisk (kontenerów) zawiera poszczególne elementy

(programy) składające się na stos technologiczny projektu. Wśród zalet tego podejścia można wymienić [40]:

- Przenośność – gdy cała konfiguracja jest zamknięta w kontenerze, końcowy użytkownik musi tylko pobrać i uruchomić odpowiedni obraz
- Luźne powiązanie – korzystając z niezależnych kontenerów łatwo jest je podmieniać lub modyfikować
- Skalowalność – kontenery są łatwe do powielania i rozpowszechniania
- Bezpieczeństwo – każdy kontener jest zamkniętym ekosystemem, więc awaria jednego z nich nie zagraża reszcie

4.7 Auth0

Auth0 jest to internetowy serwis zapewniający usługi autentykacji i autoryzacji (aaS – jako serwis) [41]. Auth0 implementuje szereg standardów związanych z zabezpieczeniem aplikacji webowych, takich jak OAuth, OpenID Connect, JWT. Korzystanie z tego oprogramowania pozwala programiście w łatwy sposób dodać bezpieczne mechanizmy kontroli dostępu i logowania do swojego projektu. Rysunek 9 przedstawia przykładowe okno logowania w Auth0. Jak widać możliwe jest ustawienie autentykacji zarówno za pomocą ustawionego loginu i hasła, jak i poprzez integrację z innymi portalami np. GitHub, Google itp.



Rysunek 9 Okno logowania Auth0

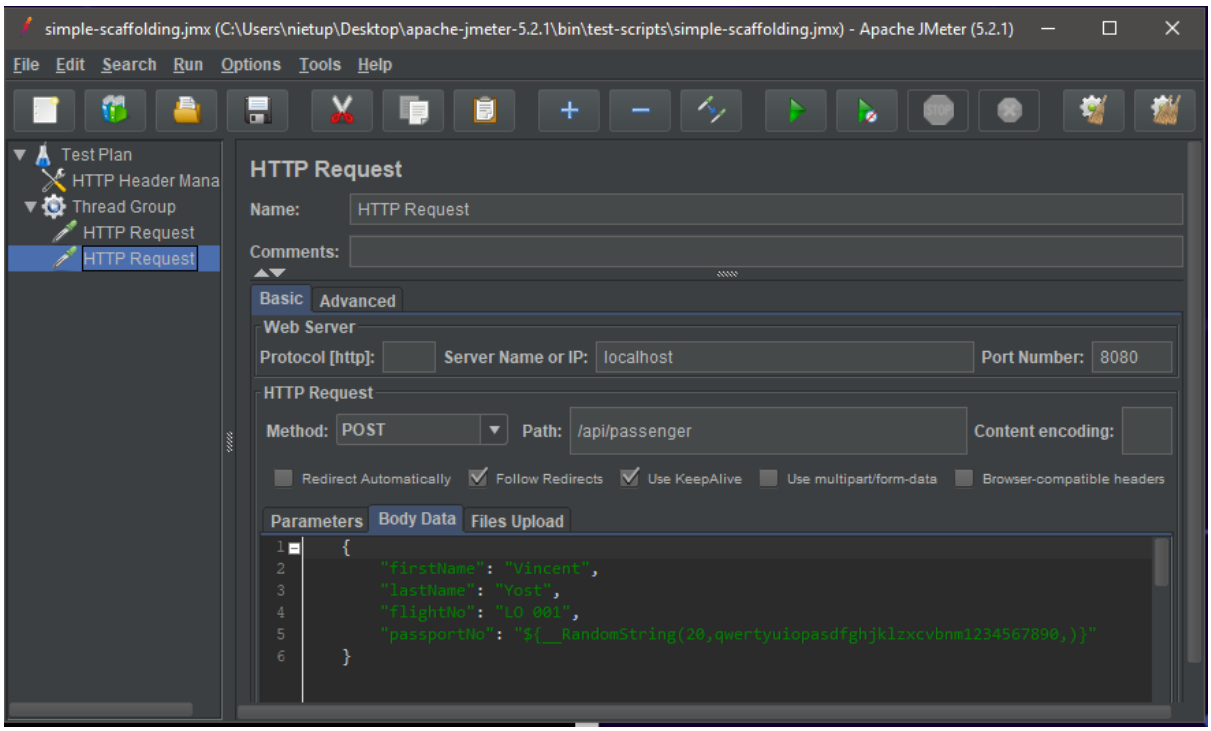
4.8 PostgreSQL

Jako system bazodanowy dla tworzonych projektów wybrano PostgreSQL. Jest to system open source z prawie trzydziestoletnią historią, który znalazł uznanie i zabezpieczył swoją pozycję na rynku baz danych [42].

4.9 JMeter

Apache JMeter jest narzędziem służącym do przeprowadzania testów wydajnościowych. Pierwotnie jego celem było testowanie aplikacji webowych, lecz z czasem możliwości tego programu zostały rozszerzone także na inne przypadki [43].

JMeter posiada intuicyjny interfejs graficzny (Rysunek 10) pozwalający na przygotowanie scenariuszy testowych. Dużą zaletą tego oprogramowania jest możliwość nagrywania zachowania użytkownika na stronie internetowej, a następnie odtwarzanie tych czynności w ramach testu. Program ten jednak nie może być wykorzystany do testowania interfejsu aplikacji, działa on w całości na poziomie protokołu (w tym przypadku HTTP).



Rysunek 10 Interfejs użytkownika programu JMeter

5 Opis wymagań

Tematem projektów testowych jest system do rezerwacji biletów lotniczych. Strona przeglądarkowa umożliwia wyszukiwanie i rezerwację lotu, a RESTowe API służy do zarządzania danymi lotów, przewoźników oraz pasażerów. W poniższym rozdziale zawarto szczegółowy opis wymagań funkcjonalnych oraz modelu danych projektu testowego, a następny rozdział zawira opis wykonania aplikacji z wykorzystaniem poszczególnych narzędzi.

5.1 Opis funkcjonalności

Główna funkcjonalność systemu to wyszukiwanie i rezerwacja biletów lotniczych. System ma umożliwić następujące czynności:

1. wprowadzenie przez użytkownika parametrów lotu: miasta startowego i docelowego oraz godziny startu
2. wyświetlenie użytkownikowi informacji o dostępnych połączeniach
3. zarezerwowanie miejsc przez użytkownika poprzez
    a. wybór połączenia
    b. wprowadzenie danych pasażera
4. poprzez REST API:
    a. pobieranie informacji o portach lotniczych
        i. `GET /airports/{kod portu}`
    b. pobieranie informacji o miastach i portach które się w nich znajdują
        i. `GET /cities/{nazwa}`
        ii. `GET /cities/{nazwa}/airports`
    c. pobieranie i edycja informacji o planowanych lotach
        i. `POST /flights`
        ii. `GET, PUT, DELETE /flights/{numer lotu}`
        iii. `GET /flights/{numer lotu}/passengers`
        iv. `POST /flightSearch`, ciało zapytania:
            1. miasto startowe
            2. miasto docelowe
            3. godzina startu
            4. tolerancja czasowa
    d. pobieranie i edycja informacji o pasażerach
        i. `POST /passengers`
        ii. `GET, PUT, DELETE /passengers/{id pasażera}`
            1. przy dodawaniu pasażera zachodzi weryfikacja czy lot ma jeszcze dostępne miejsca
5. logowanie

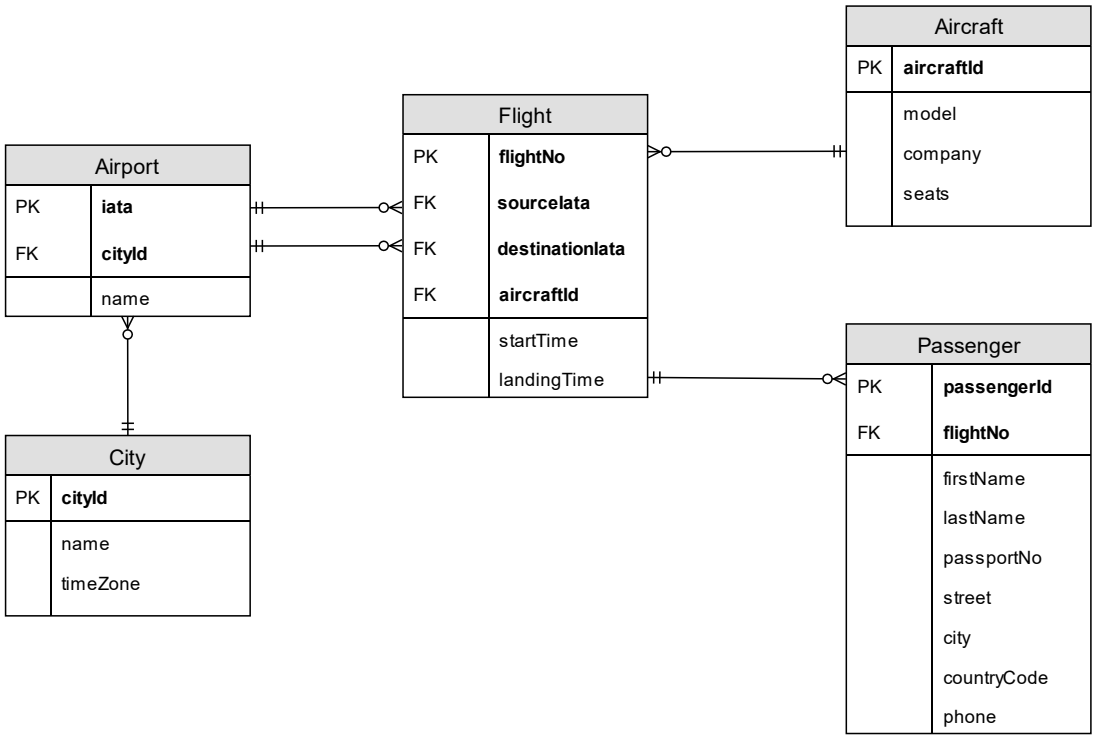
5.2 Model danych

System operuje na następujących obiektach:

1. Port lotniczy
    a. nazwa
    b. kod

    c. miasto w którym się znajduje
2. Miasto
    a. nazwa
    b. strefa czasowa
3. Lot
    a. numer lotu
    b. lotnisko startowe
    c. lotnisko docelowe
    d. godzina startu
    e. godzina przylotu
    f. samolot
4. Samolot
    a. nazwa modelu
    b. liczba miejsc
    c. nazwa przewoźnika
5. Pasażer
    a. imię
    b. nazwisko
    c. numer dokumentu tożstamości
    d. adres
    e. numer telefonu
    f. lot na który ma bilet

Rysunek 11 przedstawia schemat obiektów biznesowych oraz relacje pomiędzy nimi.



Rysunek 11 Model danych aplikacji testowej

6 Opis implementacji aplikacji testowych

Poniższy rozdział jest poświęcony opisowi implementacji wymienionych wcześniej wymagań w wybranych technologiach scaffoldingowych.

6.1 JHipster

Jako pierwsza została wykonana implementacja projektu przy użyciu narzędzia JHipster. Jest to najbardziej rozbudowane narzędzie spośród trzech wykorzystanych i pozwala na generację zarówno backendu jak i strony frontendowej. Ze względu na swoją złożoność praca z użyciem tego programu może być na początku przytłaczająca, jednak twórcy projektu przygotowali rozbudowany poradnik wideo, mający na celu wprowadzenie nowych użytkowników [44]. Zakres filmu instruktażowego obejmuje:

- zdefiniowanie przykładowego modelu danych,
- wybór opcji dotyczących architektury generowanej aplikacji,
- przegląd wyniku generacji
- wprowadzanie modyfikacji do otrzymanego kodu

6.1.1 Dane wejściowe generatora

Generację aplikacji należy rozpocząć od utworzenia folderu głównego, a następnie w jego środku wpisania komendy `>jhipster`, co spowoduje uruchomienie interaktywnej aplikacji konsolowej (patrz Rysunek 12).

Rysunek 12 Okno powitalne programu JHipster

W pierwszej kolejności dokonujemy wyboru rodzaju aplikacji, jaka ma być generowana. Do wyboru mamy:

- Monolit
- Aplikacja mikroserwisowa
- Brama sieciowa (gateway) aplikacji mikroserwisowej

- Jhipster UAA (User Account and Authentication – konta użytkowników i uwierzytelnianie) Server – używany do uwierzytelniania w aplikacjach mikroserwisowych

Na potrzeby naszego projektu wystarcza opcja pierwsza – aplikacja monolitowa.

Następnie definiujemy nazwę aplikacji oraz nazwę pakietu (package) javowego. W dalszej kolejności jesteśmy proszeni o wybór rodzaju autentykacji, jakiego chcemy użyć. Tu dostępnymi opcjami są:

- JWT [45] (JSON Web Token)
- HTTP Session [46] – domyślny mechanizm wykorzystywany przez Spring Security
- OAuth 2.0 [47] – popularny protokół implementowany przez wiele narzędzi np. Keycloak

Na potrzeby aplikacji testowej wybrano opcję pierwszą, ponieważ pozostałe projekty są zintegrowane z serwisem do zarządzania logowaniem Auht0, w którym ta opcja jest domyślnie wykorzystywana.

Kolejnym krokiem jest wybór bazy danych z której korzystać będzie aplikacja. Dostępnymi opcjami są:

- MySQL [48]
- MariaDB [49]
- PostgreSQL [42]
- Oracle [50]
- MS SQL Server [51]
- MongoDB [52]
- Cassandra [53]
- Couchbase [54]

Zgodnie z założeniami wybrano opcję „PostgreSQL”. Dodatkowo możliwy jest wybór bazy H2 w pamięci operacyjnej na potrzeby rozwoju i testowania aplikacji.

Później decydujemy czy chcemy korzystać z pamięci podręcznej (cache) implementowanej przez Spring i Hibernate. Dokonujemy wyboru narzędzia do kompilacji aplikacji: Maven lub Gradle.

W następnym etapie procesu możemy zdecydować się na integrację z dodatkowymi narzędziami:

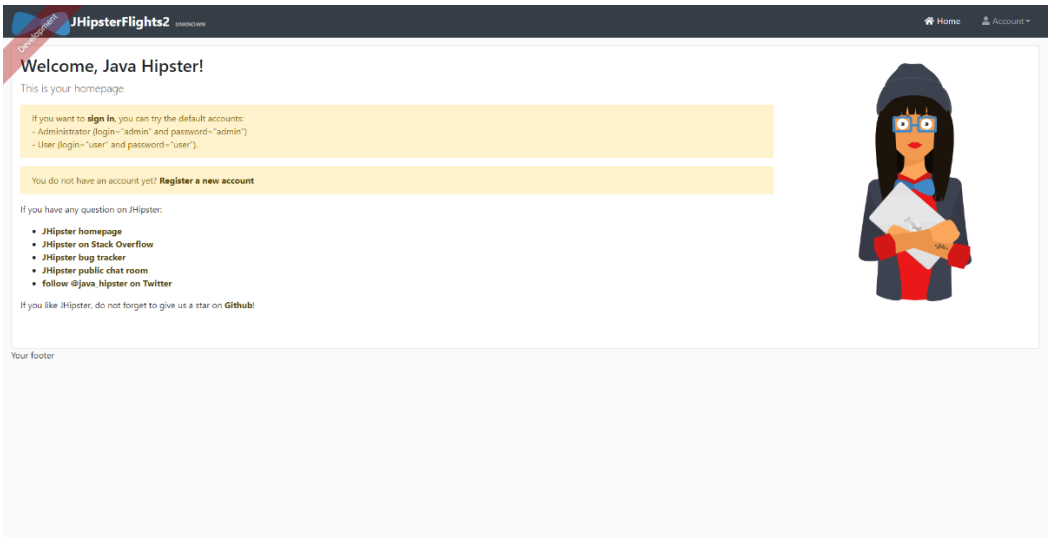
- Elasticsearch [55] do indeksowania oraz wyszukiwania encji
- WebSocket [56] implementowany przez Springa
- Asynchroniczne wiadomości przy użyciu Apache Kafka [57]
- Generatora OpenAPI, [58] pozwalającego na generowanie kodu kontrolerów na podstawie specyfikacji API zawartej w pliku konfiguracyjnym

Na potrzeby zadanej aplikacji nie jest potrzebna żadna z tych opcji. Dalej następuje wybór biblioteki frontendowej: Angular lub (tak jak w naszym przypadku) React. Później należy dokonać wyboru języków dla internacjonalizacji aplikacji. Na końcu oferowany jest wybór platformy do testowania:

- Gatling [59] – testowanie wytrzymałościowe (load testing)

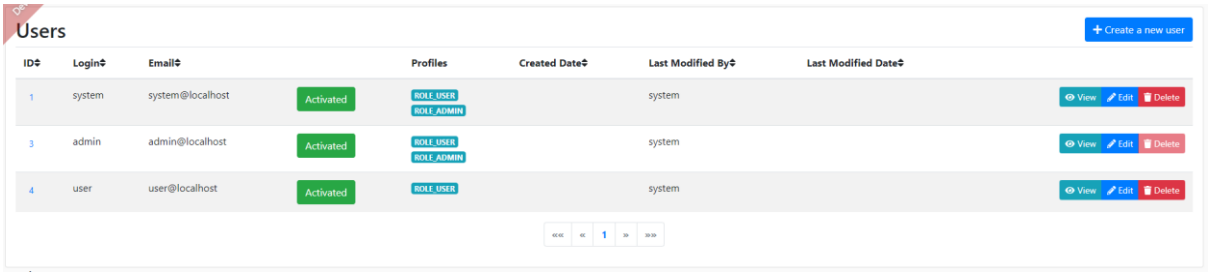
- Cucumber [60] – platforma testowa przystosowana do paradygmatu BDD (Behaviour Driven Development)
- Protractor [61] – testy „End to End”

Po ostatniej decyzji generator utworzy podstawową strukturę projektu, zgodnie ze wszystkimi konfiguracjami oraz integracjami, jakie wybrał użytkownik. Wytworzony projekt może zostać uruchomiony (Rysunek 13 przedstawia ekran startowy wygenerowanej aplikacji) i można korzystać z jego podstawowych funkcji takich jak:



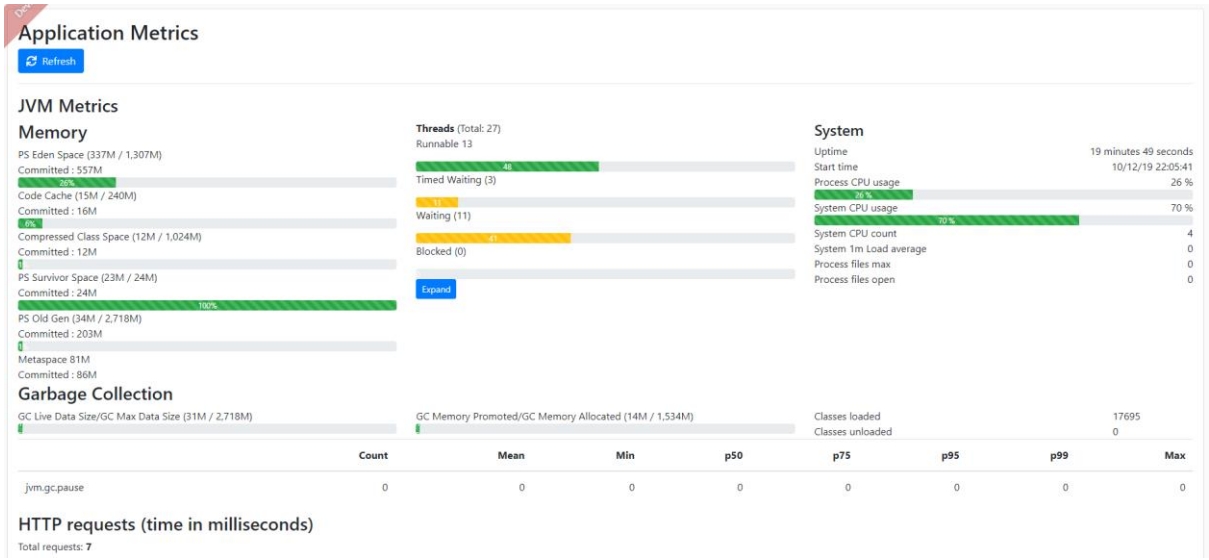
Rysunek 13 Ekran startowy widoczny po odpaleniu nowo wygenerowanej aplikacji w JHipster

1. Logowanie (od razu gotowe są testowe konta: admin i user)
2. Zarządzanie użytkownikami: dodawanie, usuwanie, zmiana danych lub uprawnień (Rysunek 14).



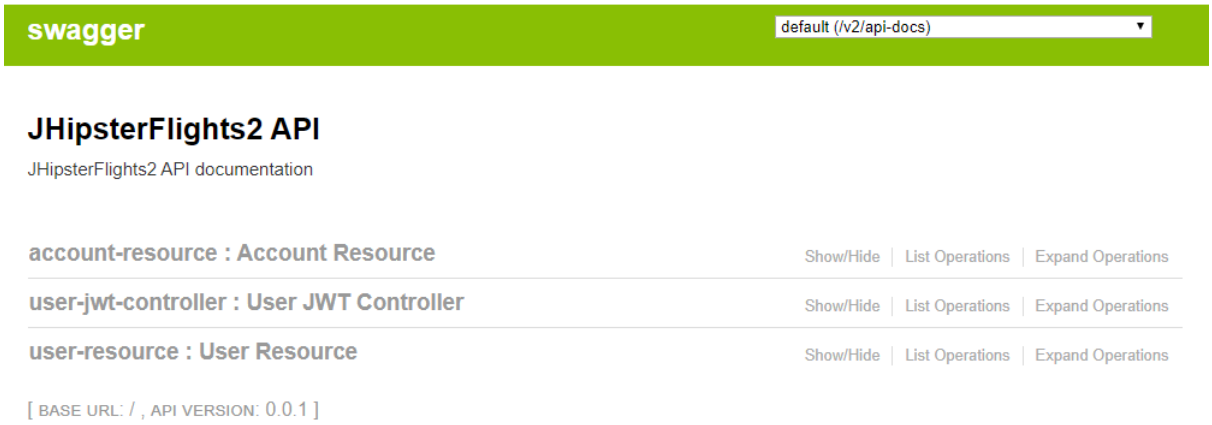
Rysunek 14 Panel zarządzania użytkownikami

3. Podgląd statystyk dotyczących zużycia pamięci, otrzymywanych zapytań czy pamięci podręcznej (Rysunek 15).



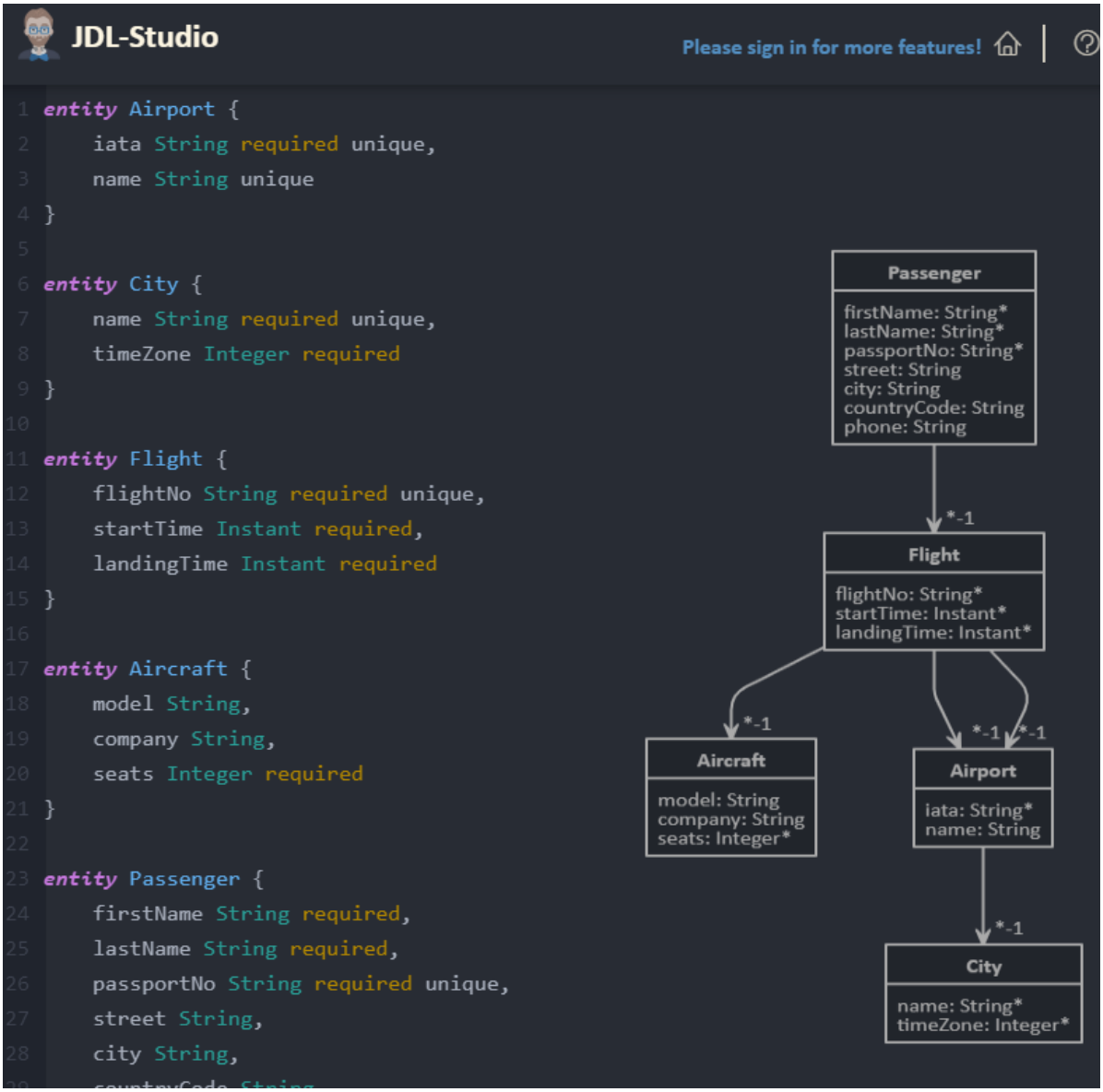
Rysunek 15 Statystyki wyświetlane w czasie rzeczywistym

4. Przeglądanie logów aplikacji oraz ustawienie poziomu logowania (trace, debug, warn, error) dla poszczególnych klas aplikacji.
5. Wyświetlenie automatycznie generowanej dokumentacji RESTowego API poprzez integrację z narzędziem Swagger (Rysunek 16). Generowana przez Swaggera strona pozwala również na testowanie poszczególnych endpointów poprzez wysyłanie zapytań z gotowymi przykładowymi ciałami zapytań.



Rysunek 16 Dokumentacja API wygenerowana przez program Swagger

Na tym etapie należy zdefiniować model danych. Może on być zapisany przy pomocy dowolnego edytora tekstu, jednak twórcy JHipster stworzyli narzędzie ułatwiające wykonanie tego zadania. Jest to JDL (JHipster Domain Language) Studio. Program ten jest dostępny za darmo w internecie, obsługuje się go przez przeglądarkę internetową. Jego główną funkcją jest wizualizacja w czasie rzeczywistym definiowanego modelu. Rysunek 17 przedstawia widok aplikacji JDL Studio dostępnej pod adresem https://start.jhipster.tech/jdl-studio/.



Rysunek 17 Zrzut ekranu z aplikacji JDL Studio

Opisanie modelu danych nie jest trudne. W pierwszej kolejności wyszczególniane są wszystkie encje biznesowych oraz ich atrybuty. Dla każdego atrybutu konieczne jest określenie nazwy i typu oraz możliwe jest dodanie słów kluczowych wskazujących na to, że atrybut jest wymagany lub musi być unikalny. Następnie definiowane są relacje pomiędzy obiektami. W tym przypadku określany jest rodzaj relacji: jeden-do-jednego lub jeden-do-wielu oraz to czy relacja jest wymagana. Obszerna dokumentacja [62] zawiera opis składni potrzebny do kompletnego zdefiniowania modelu.

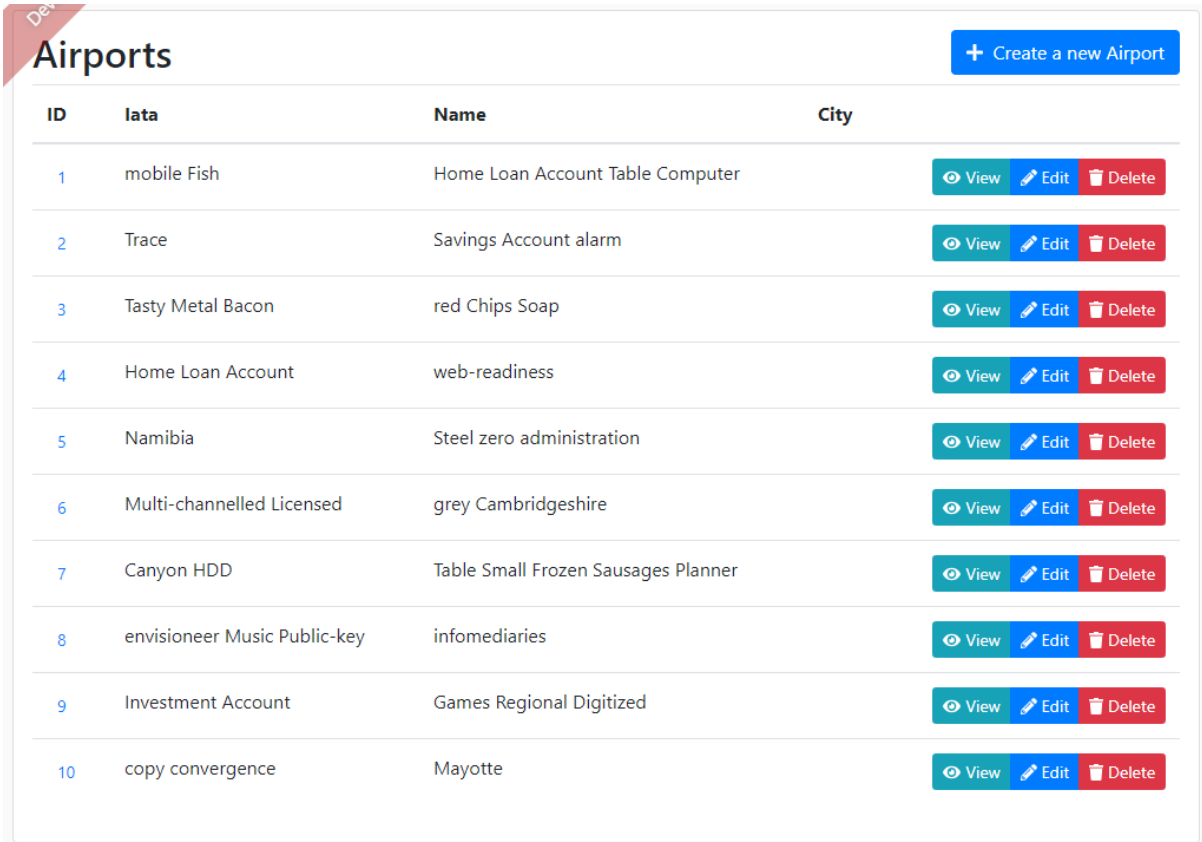
Trzeba jednak zaznaczyć, iż JHipster nie pozwala na definiowane własnych kluczy głównych – zawsze będzie to automatycznie generowane pole o nazwie „id” typu long [63]. W związku z tym nie można było wymusić by np. kluczem miasta była jego nazwa, a co za tym idzie aby został automatycznie wygenerowany endpoint pozwalający na znalezienie miasta na podstawie nazwy.

Gdy definiowanie modelu zostanie zakończone, należy zapisać stworzony kod w pliku o rozszerzeniu `.jdl` w głównym folderze projektu oraz zaimportować model do generatora. Dokonuje

się tego poprzez wpisanie w wierszu poleceń komendy `>jhipster import-jdl [nazwa pliku]`. Jest to ostatni etap związany z generacją kodu w JHipster.

6.1.2 Wynik działania generacji

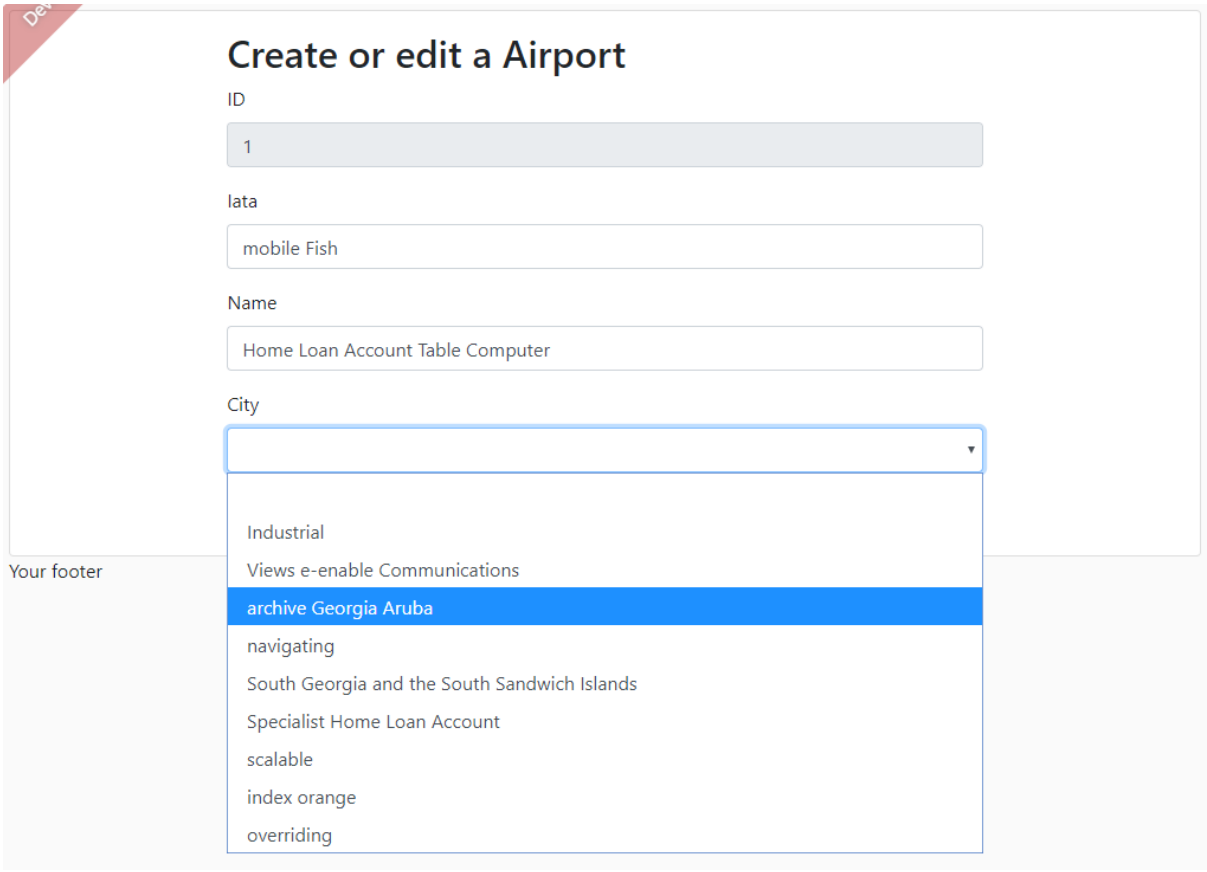
Po zakończeniu pracy z generatorem kodu użytkownik otrzymuje gotową do uruchomienia aplikację. Poza ekranami administracyjnymi, przedstawionymi w poprzednim rozdziale, po zaimportowaniu modelu danych, dodane zostały widoki poświęcone poszczególnym encjom umożliwiające wyświetlanie, tworzenie, edycję i usuwanie obiektów (patrz Rysunek 18 i Rysunek 19).



Rysunek 18 Ekran z automatycznie wygenerowanymi testowymi obiektami typu „Airports”

Gotowe jest także pełne API dla każdej encji wystawiające następujące endpointy:

- `GET /api/{nazwa encji}`
- `GET /api/{nazwa encji}/{id}`
- `POST /api/{nazwa encji}`
- `PUT /api/{nazwa encji}/{id}`
- `DELETE /api/{nazwa encji}/{id}`



Rysunek 19 Ekran przedstawiający okno edycji wybranego obiektu z listą umożliwiającą asocjację pomiędzy lotniskiem a miastem

Jeśli w pliku JDL zażądano dla danego obiektu obsługi kryteriów wyszukiwania (słowo kluczowe „filter") to do zapytania wysyłanego z nagłówkiem GET można dodać odpowiednie zapytanie w postaci parametru ścieżki (ang. path param). Dodatkowo generowany jest także endpoint:

- `GET /api/{nazwa encji}/count`

Który zwraca liczbę obiektów spełniających zadane kryterium.

6.1.3 Samodzielna implementacja brakujących elementów backendu

Porównanie wygenerowanej aplikacji z zestawem wymagań wskazuje na to, iż konieczna jest samodzielna implementacja następujących trzech z trzynastu endpointów:

- `GET /cities/{nazwa}/airports`
- `GET /flights/{numer lotu}/passengers`
- `POST /flightSearch`,
- dodanie do `POST /passengers` weryfikacji dostępności miejsc na wybrany lot

oraz przystosowanie wygenerowanych ekranów na potrzeby zadanej funkcjonalności.

Zaimplementowanie pierwszych dwóch endpointów było bardzo proste z wykorzystaniem wygenerowanego wyszukiwania za pomocą kryteriów. Zaimplementowane przez twórców JHipster

serwisy pozwalają na filtrowanie po dowolnym atrybucie danej klasy. Listing 4 przedstawia przykład wykorzystania filtrowania dla wyszukiwania lotnisk po relacji do miasta:

Listing 4 Użycie kryteriów na przykładzie wyszukiwania lotnisk

```
LongFilter idEqualsFilter = new LongFilter();
idEqualsFilter.setEquals(city.getId());

AirportCriteria airportCriteria = new AirportCriteria();
airportCriteria.setCityId(idEqualsFilter);

airports.addAll(airportQueryService.findByCriteria(airportCriteria));
```

Zaimplementowanie funkcjonalności do wyszukiwania lotów nie było już tak oczywiste, gdyż konieczne było stworzenie nowego endpointu oraz Obiektu Transferu Danych (ang. Data Transfer Object – DTO) który miałby być wysyłany w ciele zapytania. JHipster pozwala na „dogenerowanie" elementów do istniejącego projektu. Możliwe są dwie drogi przeprowadzenia takiej operacji. Pierwsza z nich polega na dodaniu wszystkich przekrojowych funkcjonalności związanych z nową encją: kontrolera, serwisu, repozytorium itp. Drugą możliwością jest utworzenie tylko konkretnej warstwy. Na potrzeby projektu nie było konieczne zapisywanie historii wyszukiwań w bazie danych, toteż zdecydowano się na drugą opcję. Osiągnąć ten efekt można modyfikując plik „.jh" lub korzystając z interaktywnego narzędzia konsolowego. Aby stworzyć kontroler za pomocą terminala należy użyć komendy `>jhipster spring-controller [nazwa kontrolera]`, a następnie udzielać odpowiedzi na zadawane pytania (patrz Rysunek 20).

```
C:\Users\nietup\Desktop\JHipsterFlights2>jhipster spring-controller search-request
INFO! Using JHipster version installed locally in current project's node_modules
INFO! Executing jhipster:spring-controller search-request
INFO! Options: from-cli: true
The spring-controller search-request is being created.
? Do you want to add an action to your controller? Yes
? What is the name of your action? search
? What is the HTTP method of your action? POST
? Do you want to add an action to your controller? No
adding Post action 'search' for /api/search-request/search
   create src\main\java\mgr\flights\jhipster\web\rest\Search-requestResource.java
   create src\test\java\mgr\flights\jhipster\web\rest\Search-requestResourceIT.java
INFO! Congratulations, JHipster execution is complete!
```

Rysunek 20 Przykład użycia pod-generatora JHipster

Co prawda JHipster umożliwia generowanie DTO, jednak dzieje się to zawsze jednocześnie z tworzeniem serwisu [64], a to nie było potrzebne do wykonania naszego zadania (wszystkie serwisy które były konieczne – lotniska oraz lotu – były już stworzone). Z tego względu zaimplementowano go ręcznie dodając takie pola, jakie znajdują się w formularzu wyszukiwania lotów w aplikacji (Listing 5).

Implementacja wyszukiwania lotów również została wykonana z rozległym wykorzystaniem kryteriów wyszukiwania. Po pierwsze wykorzystano je do wyszukania lotnisk na podstawie nazwy miasta (nazwa jest polem encji „City"). Mając przygotowane listy lotnisk startowych oraz docelowych (w jednym mieście może znajdować się wiele lotnisk) możliwe było przygotowanie kryteriów wyszukiwania lotów. Po pierwsze trzeba było wyciągnąć listy identyfikatorów poszczególnych lotnisk

i z ich wykorzystaniem stworzyć pierwsze filtry, a następnie przefiltrować godziny startu z uwzględnieniem tolerancji czasowej podanej przez użytkownika (Listing 6).

Listing 5 Klasa SearchRequestDTO

```
public class SearchRequestDTO implements Serializable {

    @NotNull
    private String source;

    @NotNull
    private String destination;

    @NotNull
    private Instant startTime;

    private Integer timeRange;

    //... gettery i settery
}
```

Listing 6 Przygotowanie kryteriów wyszukiwania lotów

```
private FlightCriteria getSearchCriteria(
    List<AirportDTO> sourceAirports,
    List<AirportDTO> destinationAirports,
    Instant startTime, Integer timeRange) {

    List<Long> sourceIds = sourceAirports
        .stream()
        .map(AirportDTO::getId)
        .collect(Collectors.toList());

    LongFilter sourceIdFilter = new LongFilter();
    sourceIdFilter.setIn(sourceIds);

    List<Long> destinationIds = destinationAirports
        .stream()
        .map(AirportDTO::getId)
        .collect(Collectors.toList());

    LongFilter destinationIdFilter = new LongFilter();
    destinationIdFilter.setIn(destinationIds);

    InstantFilter startTimeFilter = new InstantFilter();
    startTimeFilter.setGreaterThanOrEqual(
        startTime.minusSeconds(timeRange * 1800));
    startTimeFilter.setLessThanOrEqual(
        startTime.plusSeconds(timeRange * 1800));

    FlightCriteria criteria = new FlightCriteria();
    criteria.setSourceId(sourceIdFilter);
    criteria.setDestinationId(destinationIdFilter);
    criteria.setStartTime(startTimeFilter);

    return criteria;
}
```

Dodanie powyższego fragmentu kodu umożliwia wyszukiwanie lotów na podstawie zadanych warunków. Ostatnim krokiem było dodanie weryfikacji dostępności miejsc na wybrany przez użytkownika lot. Do tego celu potrzebne było zliczenie pasażerów już zapisanych na ten lot. Metodę przeznaczoną do tego celu przedstawia Listing 7.

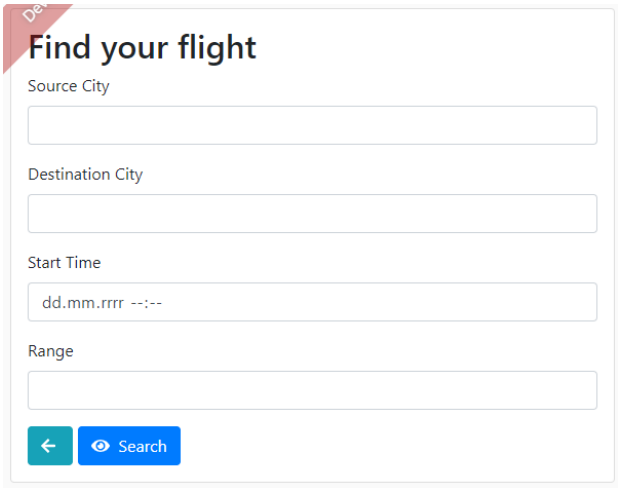
Listing 7 Zliczanie pasażerów zapisanych na dany lot

```
private Integer countPassengersByFlightId(Long flightId) {
    LongFilter flightIdFilter = new LongFilter();
    flightIdFilter.setEquals(flightId);
    PassengerCriteria passengerCriteria =
        new PassengerCriteria();
    passengerCriteria.setFlightId(flightIdFilter);
    return passengerQueryService
        .findByCriteria(passengerCriteria).size();
}
```

6.1.4 Samodzielna implementacja brakujących elementów frontendu

Jak pisano wcześniej, JHipster tworzy dla użytkownika widoki pozwalające na proste przeglądanie, dodawanie, edycję oraz usuwanie wszystkich zdefiniowanych obiektów. Konieczne było jednak przystosowanie aplikacji frontendowej na potrzeby zadanej logiki.

Pierwszą zmianą do wprowadzenia jest dodanie okna z formularzem do wyszukiwania lotów. W tym przypadku również można było skorzystać z generatora JHipster – trzeba było wygenerować widoki dla abstrakcyjnego obiektu „search”. Minusem tego podejścia jest to że generuje on pełen zestaw podstron: tabelę z encjami, podgląd, edycję, podczas gdy nam potrzebny jest jedynie formularz do wprowadzania danych. Jednakże nadmiarowe pozycje w menu oraz podstrony można było pochować i dzięki temu przy minimalnym nakładzie pracy otrzymać formularz wyszukiwania (Rysunek 21) wraz z przechowywaniem informacji o historii wyszukiwań.



Rysunek 21 Formularz wyszukiwania lotu

Drugą konieczną poprawką było dodanie okna z wynikami wyszukiwania. Kod tabeli był już w zasadzie gotowy – chodzi przecież o wyświetlanie znalezionych lotów, a właściwa tabela została wygenerowana jako element funkcjonalności związanej z encją „Flight". Trzeba było tylko zadbać o to, by w tabeli pojawiały się jedynie pożądane przez użytkownika loty, a nie wszystkie znajdujące się w bazie. Zadanie to było bardzo proste dzięki temu, że JHipster generuje aplikację Reactową zgodnie z wcześniej opisaną architekturą Flux. Wystarczyło dodać akcję komunikującą się z wcześniej zaimplementowanym endpointem wyszukiwania (Listing 8) do reducera lotów.

Listing 8 Akcja odpytująca endpoint wyszukiwania lotów

```
export const searchFlight: ICrudGetAllAction<IFlight> = body => ({
  type: ACTION_TYPES.FETCH_FLIGHT_LIST,
  payload: axios.post('api/search-request/search-flight', body)
});
```

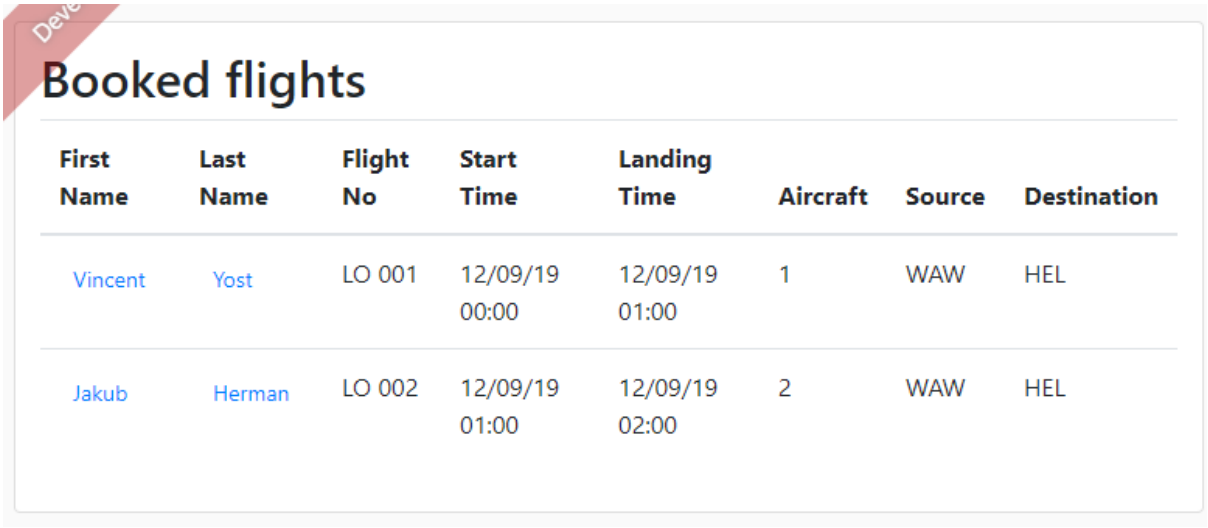
W efekcie możliwe było użycie tej akcji aby pobrać loty spełniające warunki ostatniego wyszukiwania (Listing 9).

Listing 9 Pobieranie lotów z backendu

```
export class SearchHistory extends React.Component<ISearchHistoryProps> {
  componentDidMount() {
    // metoda wykonująca się od razu po załadowaniu komponentu

    this.props.getEntities().then(data => {
      const i = data.value.data.length - 1;
      const src = data.value.data[i]['sourceCity'];
      const dest = data.value.data[i]['destinationCity'];
      const sT = data.value.data[i]['startTime'];
      const rng = data.value.data[i]['range'];

      this.props.searchFlight({
        "destination": dest,
        "source": src,
        "startTime": sT,
        "timeRange": rng
      });
    });
  }

  render() {
    // ... wyświetlenie wyników w tabeli
  }
}
```

Trzecim elementem jest rezerwacja lotu poprzez wprowadzenie danych pasażera. Do tego zadania przekształcono wygenerowany formularz do tworzenia pasażera. Wymagał on dodania informacji o wybranym w poprzednim kroku locie oraz zablokowania w formularzu możliwości zmiany numeru lotu. Cel ten osiągnięto przekazując informację o id lotu w parametrze źródła strony oraz pobraniu informacji o locie w podobny sposób jak na poprzednim listingu.

Na koniec konieczne było dodanie podglądu lotów zarezerwowanych przez zalogowanego użytkownika. W tym celu wprowadzono pewne zmiany do widoku zawierającego tabelę z użytkownikami. Zmieniono zestaw kolumn tabeli tak aby wyświetlał imię i nazwisko pasażera oraz dane dotyczące lotu (Rysunek 22). Wymagało to także pewnej zmiany po stronie backendowej: dodano DTO zawierający połączone informacje dotyczące pasażera i jego lotu oraz dodano nową metodę do repozytorium pasażerów, która pozwala na pobranie z bazy danych pasażerów dodanych tylko przez obecnie zalogowanego użytkownika. Dokonano tego poprzez użycie języka zapytań będącego częścią standardu Java Persistence API – JPQL (Listing 10).



Rysunek 22 Podgląd pasażerów utworzonych przez zalogowanego użytkownika i ich lotów

Listing 10 Kod repozytorium pasażerów

```
@SuppressWarnings("unused")
@Repository
public interface PassengerRepository
    extends JpaRepository<Passenger, Long>,
    JpaSpecificationExecutor<Passenger> {

    @Query(
        "select passenger " +
        "from Passenger passenger " +
            "where passenger.user.login = ?#{principal.username}"
    )
    List<Passenger> findByUserIsCurrentUser();

}
```

6.1.5 Wnioski z pracy z JHipster

Podsumowując tworzenie aplikacji przy użyciu JHipster, należy podkreślić, że jest to bardzo rozbudowane narzędzie o szerokich możliwościach. Pozwala na stworzenie działającego programu prawie bez dopisywania kodu. Czyni go to świetnym wsparciem do prototypowania i eksperymentowania z nowymi pomysłami na projekty. Trzeba jednak zauważyć, że jego rozległe możliwości generacyjne powodują, że (mimo pewnego stopnia elastyczności) użytkownik będzie tworzył nadmiarowy kod, który nie jest potrzebny w aplikacji i który, oprócz zwiększonych wymagań

np. pamięciowych, może tworzyć pewnie nieoczekiwane zależności w programie i przez to utrudniać dalszy, samodzielny rozwój budowanego systemu.

6.2 jOOQ

Kolejnym narzędziem, które wykorzystano do implementacji jest jOOQ. Głównym zadaniem narzędzia jOOQ jest umożliwienie alternatywnego podejścia do mapowania obiektowo-relacyjnego, jednakże aby osiągnąć ten cel, jOOQ posiada pewne mechanizmy generacji kodu.

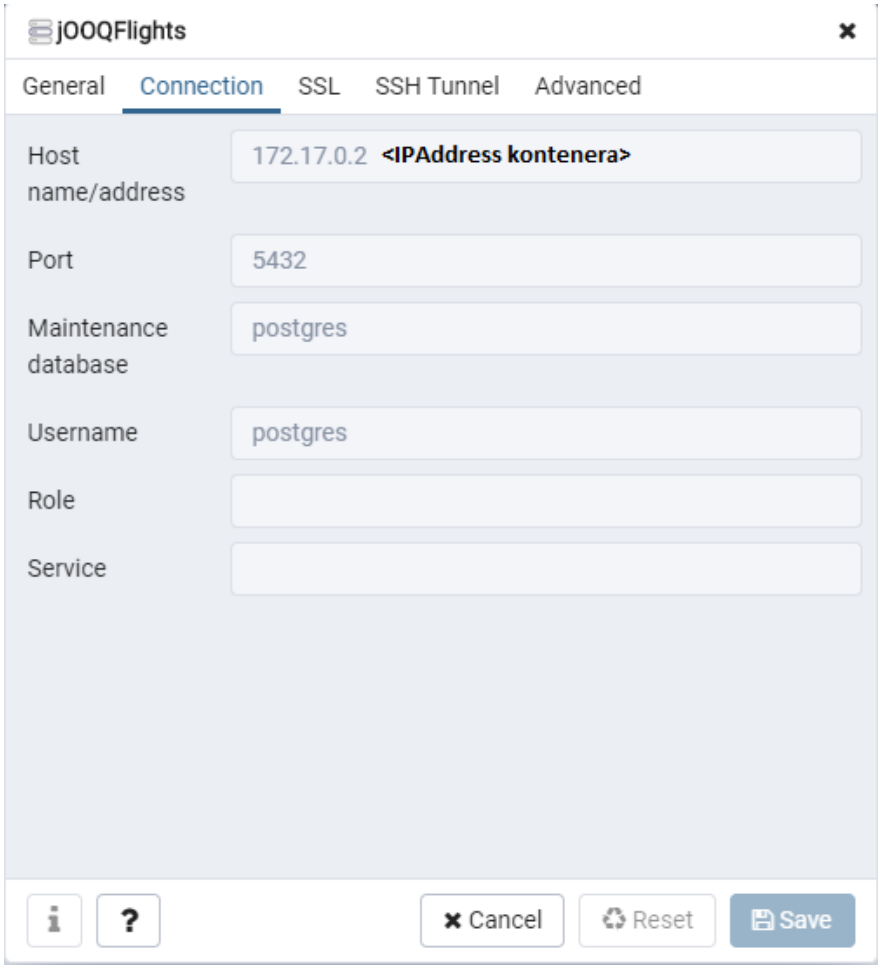
6.2.1 Dane wejściowe generatora

jOOQ generuje klasy na podstawie schematu bazy danych, a więc pierwszym krokiem w pracy z tym narzędziem jest konfiguracja bazy danych. jOOQ, w przeciwieństwie do JHipster, nie oferuje wsparcia w tym zakresie. W związku do uruchomienia bazy wykorzystano kontenery Dockerowe. Dzięki nim było to bardzo proste. Aby wystartować bazę PostgreSQL i zmapować ją na standardowy port `:5432` wystarczy pobrać obraz komendą `docker pull postgres` z konsoli, a następnie uruchomić go wpisując `docker run -e POSTGRES_PASSWORD=[wybrane hasło] -d -p 5432:5432 postgres`. Dodatkowo wykorzystano również pgadmin – narzędzie graficzne służące do zarządzania bazą PostgreSQL. Zainstalowano je przy użyciu komend:

```
docker pull dpage/pgadmin4
```

```
docker run -p 80:80 -e "PGADMIN_DEFAULT_EMAIL=[wybrany email]" -e "PGADMIN_DEFAULT_PASSWORD=[wybrane hasło]" -d dpage/pgadmin4
```

Dzięki tym krokom możemy połączyć się do panelu administracyjnego pgadmin za pośrednictwem przeglądarki internetowej pod adresem `localhost:80`. Następnie należy skonfigurować połączenie z bazą danych (patrz Rysunek 23) podając odpowiedni adres IP, który można odczytać wpisując komendę `docker inspect [nazwa kontenera]`.



Rysunek 23 Okno konfiguracji połączenia w pgadmin

Po tych krokach można przejść do definiowania schematu zgodnego z wcześniej opisanym modelem danych przy użyciu języka SQL. Po utworzeniu modelu w bazie przygotowano bazowy szkielet projektu aplikacji przy użyciu Spring Initializr. Jest to inicjalizator tworzący strukturę folderów oraz klasę `main` przygotowaną na potrzeby rozwijania aplikacji Spring Boot. Zapisuje on również plik `pom.xml`, który zawiera dane potrzebne do budowania projektu przez Maven. Tam też znajduje się wykaz zależności aplikacji oraz ich konfiguracja, a zatem to właśnie tam umieszczono parametry generatora jOOQ (Listing 11). Informacje potrzebne do wprowadzenia odpowiednich zmian w pliku uzyskano z oficjalnej dokumentacji [65], jednak miejscami jest ona dosyć lakoniczna. Na przykład można tam przeczytać o tym, że „generacja DAO (ang. Data Access Object – klasa odpowiadająca za komunikację aplikacji ze źródłem danych) może zostać aktywowana przy użyciu odpowiedniej flagi”, przy czym stosowny przykład musi zostać znaleziony we własnym zakresie.

Listing 11 Konfiguracja wtyczki jOOQ w pliku `pom.xml`

```
<plugin>
    <!-- Id edycji jOOQ opartej na
    zasadach otwartego oprogramowania -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>9.4.1211</version>
        </dependency>
    </dependencies>
    <configuration>
        <jdbc>
            <driver>org.postgresql.Driver</driver>
            <url>jdbc:postgresql://0.0.0.0:5432/Flights</url>
            <user>postgres</user>
            <password>admin</password>
        </jdbc>
        <generator>
            <database>
                <name>org.jooq.meta.postgres.PostgresDatabase</name>
                <includes>.*</includes>
                <excludes></excludes>
                <inputSchema>public</inputSchema>
            </database>
            <generate>
                <pojos>true</pojos>
                <daos>true</daos>
            </generate>
            <target>
                <packageName>mgr.flights.jooq</packageName>
                <directory>target/generated-sources/jooq</directory>
            </target>
        </generator>
    </configuration>
</plugin>
```

Dzięki takiej konfiguracji kod zostanie wygenerowany w podfolderze `target/generated-sources/jooq` przy każdym pełnym budowaniu projektu, czyli np. po wykonaniu komendy `mvn clean install`.

6.2.2 Wynik działania generacji

Przy wyżej wymienionych parametrach wynikiem generacji są, dla każdego obiektu domenowego:

1. Klasy typu POJO (ang. Plain Old Java Object – proste klasy, zawierające tylko pola, konstruktor, gettery oraz setery)

2. Klasy DAO, które zawierają metody do tworzenia, aktualizowania, usuwania oraz wyszukiwania obiektów na podstawie wartości atrybutu. Operują one na obiektach POJO
3. Klasy reprezentujące tabele w bazie danych. Więcej informacji na ich temat znajduje się poniżej
4. Klasy reprezentujące pojedynczy rekord w bazie danych, które są wykorzystywane przez klasy tabel

Z jOOQ można korzystać na dwa sposoby. W przypadku prostych zapytań wystarczy klasyczne DAO. Jeśli jednak programista chce zaimplementować bardziej złożoną logikę przy odpytywaniu źródła danych, powinien użyć zasadniczego składnika projektu jOOQ czyli generowanej reprezentacji tabel.

Celem jOOQ jest dostarczenie warstwy abstrakcji odpowiedzialnej za interakcję z bazą danych [66]. Stosuje inne podejście niż biblioteki ORM (mapowania obiektowo-relacyjnego). Programista sam jest odpowiedzialny za „przetłumaczenie" zapisywanego grafu obiektów na język tabel i rekordów. O ile może to być utrudnieniem przy zapisie danych, daje duże możliwości w przypadku złożonych zapytań [67], dzięki reprezentacji struktury bazy danych oraz zapewnieniu bezpieczeństwa typowania (ang. type safety) budowanych zapytań. Efekt ten zostaje osiągnięty z pomocą generowanych klas, opisanych wyżej. Listing 12 prezentuje prosty przykład wykorzystania generowanych klas jOOQ.

Listing 12 Przykład kwerendy z wykorzystaniem jOOQ

```
//...
// klasa reprezentująca kontekst jOOQ
private final DSLContext create;

@Autowired
public PassengerController(DSLContext create, /* ... */) {
    this.create = create;
    //...
}
//...

// ciało metody
create.select()
        .from(PASSENGER)
        .where(PASSENGER.FIRST_NAME
                .eq("Andrzej"))
        .fetch();
//...
```

6.2.3 Samodzielna implementacja brakujących elementów backendu

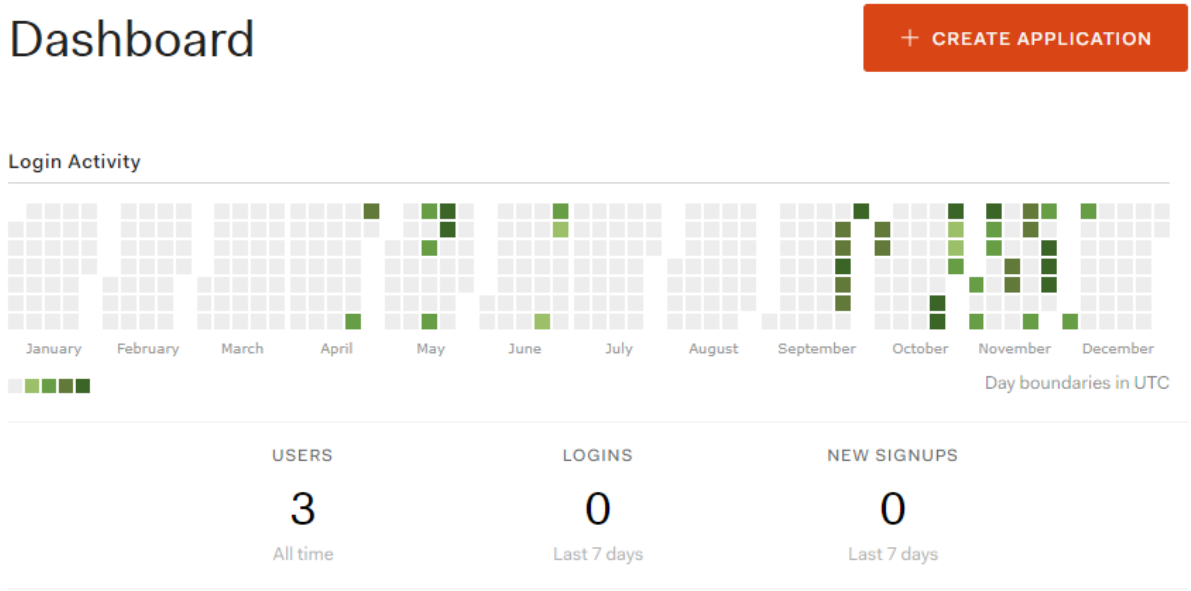
Po dokonaniu generacji projektu dysponujemy jedynie klasami reprezentującymi obiekty modelu danych oraz interfejsem dostępu do bazy danych. Nie zostały wygenerowane żadne endpointy RESTowego API, ani logika dotycząca logowania. Mimo to implementacja nie była bardzo pracochłonna. Zaprogramowanie autoryzacji zostało wykonane przy pomocy Spring Security. Spring oferuje różne możliwości konfiguracji zabezpieczeń, jednak w przypadku tego projektu zdecydowano się na użycie opcji pod nazwą „Resource Server" [68]. Jest to podejście w którym użytkownik aplikacji uwierzytelnia się za pomocą tokenu uzyskanego z zewnętrznego zasobu. Dokonano tego wyboru ze

względu na użycie Auth0. Program ten zajmuje się dostarczeniem autoryzacji i autentykacji jako usługi i wykorzystuje tokeny w formacie JSON (JSON Web Token – JWT).

Kluczowymi elementami konfiguracji zabezpieczeń od strony Springa są:

- Dodanie adnotacji `@EnableResourceServer` na klasie konfiguracyjnej
- Ustawienie następujących zmiennych w pliku konfiguracyjnym `application.properites`:
  - `auth0.domain`
  - `auth0.clientId`
  - `security.oauth2.resource.id`
  - `security.oauth2.resource.jwk.keySetUri`

`security.oauth2.resource.id` powinien wskazywać na adres chronionego API, natomiast pozostałe wartości wynikają z konfiguracji Auth0. Tej z kolei dokonuje się za pomocą interfejsu graficznego dostępnego na stronie Auth0. Aby rozpocząć, należy zalogować się do kokpitu dostępnego pod adresem `https://manage.auth0.com/` a następnie wybrać opcję CREATE APPLICATION (patrz Rysunek 24).

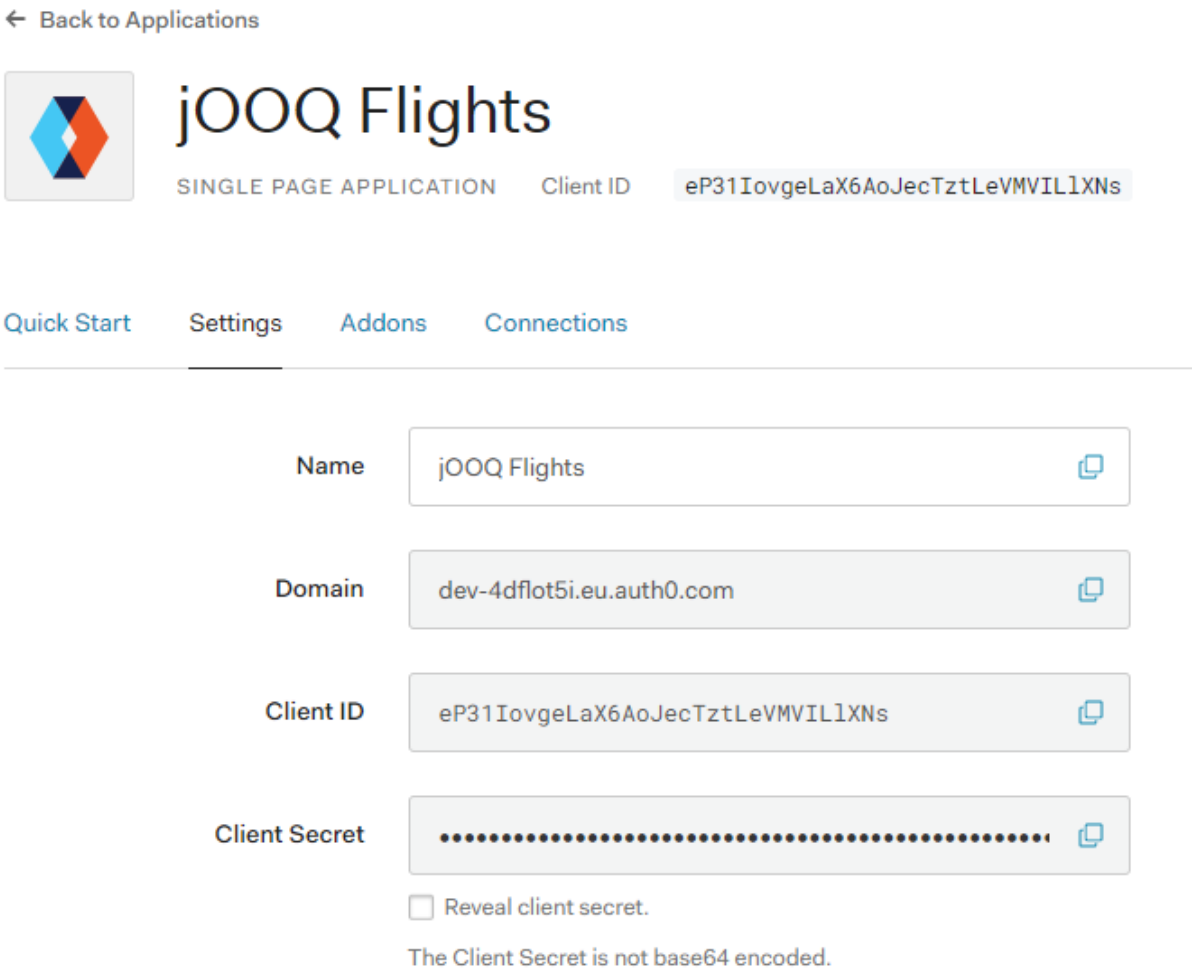


Rysunek 24 Wygląd kokpitu zarządzania Auth0

Następnie użytkownik jest proszony o wprowadzenie nazwy aplikacji oraz wybór typu spośród:

- Native – wybierane w przypadku aplikacji okienkowych oraz mobilnych
- Single Page Application – wybierany przy aplikacjach webowych tworzonych z wykorzystaniem nowoczesnych bibliotek takich React lub Angular (opcja użyta w niniejszej pracy)
- Regular Web Application – opcja dla tradycyjnych aplikacji w których frontend generowany jest na serwerze i wysyłany do użytkownika
- Machine to Machine – wykorzystywany przy aplikacjach bez interfejsu graficznego, porozumiewającymi się z innymi aplikacjami

W następnym etapie użytkownik jest przenoszony do formularza konfiguracyjnego w którym widoczne są wartości (Rysunek 25) które należy wpisać do pliku `application.properties`.



Rysunek 25 Podgląd utworzonej aplikacji w Auth0

Auth0 dostarcza wiele opcji w swoim panelu, najważniejszymi z nich, które koniecznie trzeba ustawić aby zapewnić aplikacji funkcjonowanie, są:

- Allowed Callback URLs
- Allowed Web Origins
- Allowed Logout URLs

W tych polach należy dodać adresy aplikacji frontendowej ponieważ określają z jakich adresów można wysyłać zapytania do Auth0 aby uzyskać token autoryzacyjny, oraz na jakie adresy ma nastąpić przekierowanie po poprawnym zalogowaniu.

Po ustawieniu odpowiednich zabezpieczeń można było przejść do implementacji odpowiednich endpointów. Podstawowe odczyty i zapisy do bazy były w zasadzie gotowe dzięki generowanym DAO. Listing 13 prezentuje odczyt na podstawie miasta, a Listing 14 zapis pasażera.

Listing 13 Kontroler pozwalający na odczyt miasta

```
@GetMapping("/cities/{cityName}")
public ResponseEntity<City>
getCityByName(@PathVariable String cityName) {
    City city = cityDao
            .fetchOneByName(cityName);
    if (city == null)
        throw new ResponseStatusException(
                HttpStatus.NOT_FOUND, "City: " +
                cityName + " was not found.");
    return ResponseEntity.ok().body(city);
}
```

Listing 14 Kontroler służący do zapisu pasażera

```
@PostMapping
public ResponseEntity<Passenger>
create(@RequestBody Passenger passenger) {
    if (passenger.getPassengerId() != null)
        throw new ResponseStatusException(
                HttpStatus.BAD_REQUEST,
                "New passenger cannot have an id");

    if (!flightService
            .hasFreeSeats(passenger.getFlightNo()))
        throw new ResponseStatusException(
                HttpStatus.BAD_REQUEST,
                "Chosen flights has no seats available");

    passengerDao.insert(passenger);
    Passenger result = passengerDao
            .fetchOneByPassportNo(passenger.getPassportNo());

    return ResponseEntity.ok().body(result);
}
```

Przy zapisie pasażera korzystano z kolejnej metody zaimplementowanej przy użyciu zapytań jOOQ, czyli `hasFreeSeats`, sprawdzającej, czy na wybranym locie są jeszcze dostępne miejsca. Jest ona przedstawiona przez Listing 15.

Listing 15 Implementacja metody `hasFreeSeats`

```
public Boolean hasFreeSeats(String flightNo) {
    int passengerCount = (int) create
            .selectCount()
            .from(PASSENGER)
            .where(PASSENGER.FLIGHT_NO.eq(flightNo))
            .fetch()
            .getValue(0, 0);

    int seats = (int) create
            .select(AIRCRAFT.SEATS)
            .from(AIRCRAFT)
            .where(AIRCRAFT.AIRCRAFT_ID.eq(create
                    .select(FLIGHT.AIRCRAFT_ID)
                    .from(FLIGHT)
                    .where(FLIGHT.FLIGHT_NO.eq(flightNo))))
            .fetch()
            .getValue(0, 0);

    return seats > passengerCount;
}
```

W tym przypadku najpierw następuje zliczenie pasażerów już zapisanych na dany lot, następnie wybranie miejsc w samolocie znalezionym na podstawie asocjacji z lotem i wreszcie porównanie uzyskanych wartości.

W podobny sposób zaimplementowano wszystkie pozostałe endpointy wraz z wymaganą logiką i regułami wyszukiwania.

6.2.4 Samodzielna implementacja frontendu

Jako że zarówno jOOQ jak i Simple Scaffolding nie wspierają generowania frontendu aplikacji, to do obu projektów wykorzystano ten sam program w Reakcie. W związku z tym poniższy rozdział opisuje proces powstawania aplikacji frontendowej dla obu aplikacji.

Pracę nad frontendem rozpoczęto od wygenerowania szkieletu aplikacji. Dokonano tego za pomocą pakietu stworzonego przez programistów z Facebook [69] poprzez wpisanie komendy `npx create-react-app [nazwa aplikacji]`. W efekcie wygenerowany zostaje folder zawierający podstawowe pliki składające się na projekt React, takie jak `App.js`, `App.css` itp.

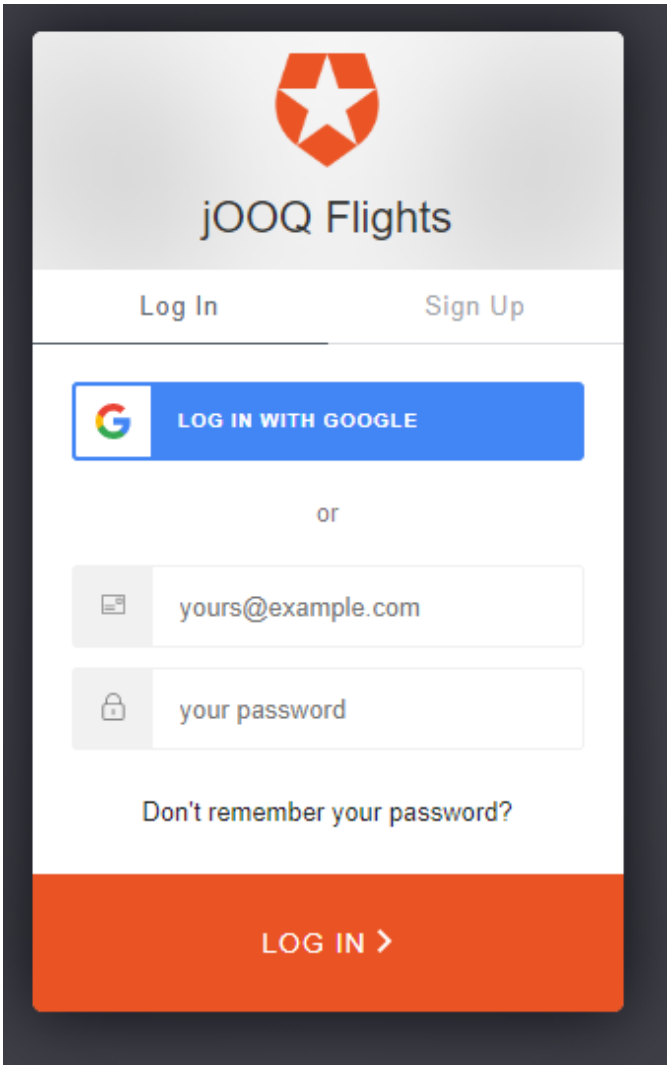
Pierwszym krokiem w implementacji była integracja z Auth0. Zasadniczym elementem tego zadania jest stworzenie klasy Auth.js, odpowiedzialnej za uzyskiwanie i odnawianie tokenu autentykującego. Właściwa zawartość jej metod jest dostępna w samouczku Auth0. W tej klasie tworzony jest obiekt auth0 (Listing 16). Jako argument przy tworzeniu przyjmuje on parametry konfiguracji połączenia z odpowiednią aplikacją na stronie Auth0. Odpowiedni JSON może być pobrany z przeglądarkowego kokpitu.

Listing 16 Tworzenie obiektu Auth0

```
auth0 = new auth0.WebAuth({
    domain: AUTH_CONFIG.domain,
    clientID: AUTH_CONFIG.clientId,
    redirectUri: AUTH_CONFIG.callbackUrl,
    audience: AUTH_CONFIG.apiUrl,
    responseType: 'token id_token',
    scope: 'openid profile read:messages'
});
```

Po stworzeniu, obiekt ten udostępnia metody pozwalające na takie operacje jak:

- Przekierowanie na stronę logowania Auth0 (Rysunek 26)
- Odnowienie tokenu w ramach bieżącej sesji
- Zakończenie sesji



Rysunek 26 Ekran logowania Auth0

W kolejnym kroku dodano formularz do wyszukiwania lotów. Sam React nie posiada wbudowanych gotowych narzędzi służących do tworzenia zaawansowanych formularzy posiadających

walidację pól, przesyłanie formularza czy śledzenie zmian (chociaż możliwości tej biblioteki jak najbardziej pozwalają na stworzenie mechanizmów zarządzających formularzami) [70]. Z tego względu do implementacji formularzy wykorzystano pakiet Formik. Celem twórców tego narzędzia było stworzenie biblioteki do zarządzania formularzami, który byłby prosty, wydajny i zapewniający tylko najważniejsze funkcje potrzebne do stworzenia formularza (zarządzenie stanem, submisja, walidacja) [71].

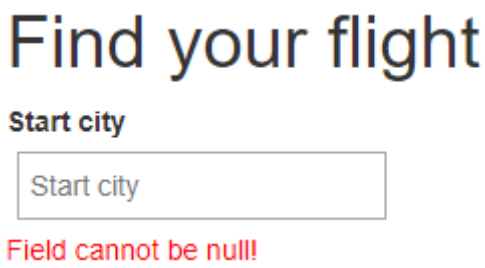
Listing 17 Fragment formularza do wyszukiwania lotów

```
<Formik
    initialValues={{startCity: "",
        destinationCity: "",
        startDate: 0,
        startTime: 0,
        timeRange: 0}}
    onSubmit={this.handleSubmit}
    render={({errors, status, touched, isSubmitting}) => (
        <Form>
            <label style={{display: 'block'}}>
                Start city
            </label>
            <Field
                type="text"
                name="startCity"
                placeholder="Start city"
                style={{display: 'block',
                    padding: ".5rem",
                    margin: ".5rem"}}
                validate={this.validateNotNull}
            />
```

Listing 17 przedstawia początek kodu formularza odpowiedzialnego za wyszukiwanie lotów. Jak widać, Formik wymaga przekazania zmiennych zawierających: domyślne wartości pól, referencję na metodę odpowiedzialną za przetwarzanie zapytania oraz metodę zawierającą kod wyświetlenia formularza (tu zdefiniowaną in-line). Dodatkowo, dla każdego pola formularza można przekazać referencję na metodę służącą do walidacji. Listing 18 przedstawia przykładowy kod walidatora wykorzystany w projekcie. Taka metoda powinna zwracać zmienną typu string. Jeżeli zwracana wartość nie jest pusta, będzie to oznaczało, że wykryto jakiś błąd a zwrócona wartość zawiera opis problemu. Ten z kolei zostaje wyświetlony na formularzu (patrz Rysunek 27). Walidacja jest przeprowadzana przy próbie zatwierdzenia formularza oraz przy „dotknięciu” pola – sytuacji w której kursor użytkownika znajdował się wewnątrz danego pola i został z niego przesunięty gdzieś indziej.

Listing 18 Metoda sprawdzająca czy dane pole nie jest puste (`null`)

```
validateNotNull = (value) => {
    let error;
    if (value === '' || value === 0) {
        error = 'Field cannot be null!';
    }
    return error;
};
```



Rysunek 27 Przykładowa wiadomość walidatora

Następnie zaimplementowano widok wyników wyszukiwania. Aby móc wyświetlić odpowiednie loty, należało uprzednio wysłać odpowiednie zapytanie do backendu. W tym celu wykorzystano bibliotekę Axios. Pozwala ona na łatwe wysyłanie asynchronicznych zapytań do RESTowego API. Listing 19 przedstawia przykład takiego zapytania. Po uzyskaniu wyników zostały one zapisane w wewnętrznym stanie komponentu, skąd mogły być wyświetlone w standardowej tabeli zdefiniowanej przy pomocy HTML.

Listing 19 Przykład zapytania przy użyciu Axios

```
axios.post(`${API_URL}/flight-search`, {
    "sourceCity": values.startCity,
    "destinationCity": values.destinationCity,
    "startTime": values.startDate + "T" + values.startTime,
    "timeRange": values.timeRange
})
    .then(response => this.setState({response: response.data}))
    .catch(error => this.setState({message: error.message}));
```

Na koniec stworzono formularz do wprowadzenia danych pasażera i stronę z podglądem wybranych lotów. Stworzono je na takiej samej zasadzie jak poprzednie komponenty: z wykorzystaniem pakietów Formik oraz Axios.

6.2.5 Wnioski z pracy z jOOQ

jOOQ jest narzędziem o zdecydowanie mniejszych możliwościach generacyjnych niż JHipster. Trzeba jednak zauważyć, że to nie rozbudowany scaffolding był głównym celem przyświecającym twórcom tego projektu. Siła jOOQ ujawnia się najbardziej, gdy wymagania projektowe sprawiają, że zapisy do bazy są raczej proste, natomiast odczyt jest bardziej zaawansowany. W przypadku aplikacji testowej tworzonej na potrzeby tej pracy zaistniały zbliżone warunki – zapis odbywał się praktycznie

na pojedynczych encjach: lot, pasażer itp., natomiast do odczytów potrzebne były trochę bardziej złożone kwerendy (np., połączenie różnych warunków przy wyszukiwania lotów). Dzięki temu implementacja zapisów była praktycznie natychmiastowa dzięki generowanym DAO, a odczyt był bardzo łatwy dzięki rozbudowanym możliwościom tworzenia zapytań.

6.3 Simple Scaffolding

Ostatnim z wykonanych projektów testowych był projekt stworzony przy użyciu Simple Scaffolding. Jest to najprostsze narzędzie z rozważanych. Całość kodu samego scaffoldera mieści się praktycznie w jednej klasie.

6.3.1 Dane wejściowe generatora

Simple Scaffolding wymaga najwięcej przygotowań przed przeprowadzeniem generacji. Programista musi samodzielnie nie tylko stworzyć strukturę projektu i skonfigurować połączenie z bazą danych, ale także zaimplementować przekrój tworzonej aplikacji, to znaczy utworzyć po jednej klasie z każdej z warstw: kontroler, serwis, DAO, itp.

Baza danych, projekt w Springu oraz zabezpieczenia zostały skonfigurowane w ten sam sposób jak w przypadku jOOQ, z tą różnicą, że w tej aplikacji wykorzystano Hibernate'a do mapowania obiektowo-relacyjnego. Hibernate pozwala również na generowanie struktury bazy na podstawie odpowiednich adnotacji umieszczonych na klasach projektu w Javie. Z tego względu nie było konieczne pisanie kodu DDL (ang. Data Definition Language – kod definiujący tabele i relacje w bazie) i możliwe było od razu rozpoczęcie implementacji. Do stworzenia przekroju wybrano miasto, ponieważ nie wymaga ono zaprogramowania żadnych dodatkowych reguł biznesowych. Zaimplementowano następujące klasy:

- Klasę reprezentującą encję
- Repozytorium
- Serwis
- Kontroler
- DTO (Data Transfer Object – obiekt zwracany na wyjściu kontolera)
- Maper pomiędzy klasą encji a DTO

W klasie reprezentującej encję (Listing 20) dodano odpowiednie adnotacje należące do pakietu `javax.persistence`. To właśnie one zostaną wykorzystane przez Hibernate do odpowiedniego wygenerowania tabel. Adnotacje te to: `Entity`, `Id`, `Column`, `GeneratedValue` oraz `NotNull`. Pozostałe adnotacje, czyli `Data`, `AllArgsConstructor` i `NoArgsConstructor` należą do biblioteki Lombok, która pozwala na przyśpieszenie tworzenia kodu w Javie poprzez generowanie powtarzalnych metod, np. adnotacja `AllArgsConstructor` powoduje wygenerowanie publicznego konstruktora klasy `City` przyjmującego jako argumenty wszystkie pola tej klasy.

Listing 20 Klasa `City`

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class City {
    @Id
    @Column(name = "city_id")
    @GeneratedValue
    @NotNull
    private Integer cityId;

    @NotNull
    @Column(name = "name",
            nullable = false,
            unique = true)
    private String name;

    @NotNull
    @Column(name = "time_zone",
            nullable = false)
    private Integer timeZone;
}
```

Pozostałe klasy wykonano przy użyciu standardowego schematu tworzenia aplikacji w Springu: mapowanie zapytań za pomocą adnotacji `RequestMapping` w kontrolerach, czy rozszerzenie interfejsu `JpaRepository` przy tworzeniu repozytorium. Ważne jest aby pokryć możliwie szeroki zakres funkcjonalności, to znaczy że na przykład mimo, że w wymaganiach nie jest zawarty endpoint do tworzenia miast, to i tak został on zaimplementowany, ponieważ stworzony kod zostanie powielony na inne encje.

Plik konfiguracyjny generatora `scaffolding.json` należy umieścić w folderze głównym projektu. Jego zawartość dla opisywanego projektu przedstawia Listing 21. Jak widać, zawiera on pola dotyczące nazw folderów i pakietów projektów. Pole `read` określa tryb działania program. W pierwszej kolejności jest to odczyt. Ostatnie pole określa nazwę szukaną w obiektach wzorcowych.

Listing 21 Plik konfiguracyjny Simple Scaffolding

```
{
  "profile": "default",
  "input_directory":
  "src/main/java/mgr/flights/simplescaffolding",
  "output_directory":
  "src/main/java/mgr/flights/simplescaffolding",
  "template_location":
  "src/test/resources/scaffolding",
  "base_package":
  "mgr.flights.simplescaffolding",
  "read": true,
  "entities": [
    "City"
  ]
}
```

Po uruchomieniu kodu scaffoldera, przy takiej konfiguracji, zapisane zostaną szablony utworzone ze wszystkich plików zawierających `City` w nazwie. Każde wystąpienie wyznaczonej nazwy w pliku zostanie zamienione na znacznik. Dzięki temu, po zmianie trybu pracy programu na `write`, w miejsce znacznika zostaną wprowadzone odpowiednie podane nazwy innych encji. Listing 22 oraz Listing 23 prezentują fragment kodu w kontrolerze `CityController`, i odnośny fragment wygenerowanego szablonu.

Listing 22 Metoda kontrolera odpowiedzialna za tworzenie miasta

```
@PostMapping
public ResponseEntity<CityDto> createCity
        (@RequestBody CityDto cityDto) {
    if (cityDto.getCityId() != null) {
        throw new ResponseStatusException(
                HttpStatus.BAD_REQUEST,
                "CityId must be null when creating city");
    }

    CityDto savedCityDto =
            cityService.createCity(cityDto);
    return ResponseEntity.ok().
            body(savedCityDto);
}
```

Listing 23 Szablon metody kontrolera

```
@PostMapping
public ResponseEntity<{{entity-class}}Dto>
create{{entity-class}}
(@RequestBody {{entity-class}}Dto {{entity-variable}}Dto) {
    if ({{entity-variable}}Dto.get{{entity-class}}Id() != null) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            "{{entity-class}}Id must be null when creating
            {{entity-variable}}");
    }

    {{entity-class}}Dto saved{{entity-class}}Dto =
    {{entity-variable}}Service
        .create{{entity-class}}({{entity-variable}}Dto);
    return ResponseEntity.ok()
        .body(saved{{entity-class}}Dto);
}
```

6.3.2 Wynik działania generacji

W wyniku generacji otrzymano zestaw metod dla obiektów: `Airport`, `Aircraft`, `Passenger` oraz `Flight`. Pozwalały one na podstawowe operacje CRUD (create, read, update i delete) tak jak dla encji `City`. Konieczne jednak było zmodyfikowanie pól wygenerowanych klas modelowych, DTO, oraz maperów, tak aby odpowiadały rzeczywistym atrybutom danego typu (czyli np. w klasie

`Passenger` dodać pola `firstname`, `lastname` itp). Po wykonaniu tej jednej poprawki dysponujemy działającą aplikacją backendową, która wymaga jedynie zaimplementowania odpowiednich reguł biznesowych.

6.3.3 Samodzielna implementacja brakujących elementów backendu

Wszystkie pozostałe wymagania projektowe trzeba było zaimplementować z użyciem standardowych metod dostępu do danych oferowanych przez JPA i Hibernate. Listing 24 przedstawia przykład sprawdzenia dostępności miejsc na dany lot. Jak widać, różni się on od sposobu, w jaki ta metoda została zaimplementowana w projekcie jOOQ. Tutaj wykonywane są oddzielne zapytania do repozytorium lotów, samolotów oraz pasażerów.

Listing 24 Metoda sprawdzająca dostępność lotów w projekcie Simple Scaffolding

```
public boolean hasFreeSeats(String flightNo) {
    Flight flight = flightRepository
            .findByFlightNo(flightNo)
            .orElseThrow(NotFoundException::new);

    Integer allSeats = aircraftService
            .getAircraftById(flight.getAircraft().getAircraftId())
            .orElseThrow(NotFoundException::new)
            .getSeats();

    Integer takenSeats = passengerService
            .getPassengersByFlightNo(flightNo)
            .size();

    return allSeats > takenSeats;
}
```

Wyszukiwanie lotów, przedstawia Listing 25, zostało stworzone na takiej samej zasadzie. Tu jednak musiały zostać wykorzystane bardziej złożone reguły filtrowania wyników na podstawie zadanych kryteriów. Zaimplementowanie ich nie było trudne z użyciem Java Stream Api, jednak trzeba zauważyć, że to filtrowanie jest wykonywane już po pobraniu danych z bazy, ma więc negatywny wpływ na wydajność.

Tak jak napisano wyżej, jako że Simple Scaffolding nie wspiera tworzeniu frontendu, to do tej aplikacji wykorzystano kod JavaScript stworzony wcześniej na potrzeby projektu jOOQ.

Listing 25 Wyszukiwanie lotów w Simple Scaffolding

```
public List<FlightDto> searchFlight(SearchRequest searchRequest) {
    LocalDateTime minTime = searchRequest
            .getStartTime()
            .minusHours(searchRequest.getTimeRange() / 2);
    LocalDateTime maxTime = searchRequest
            .getStartTime()
            .plusHours(searchRequest.getTimeRange() / 2);

    List<String> startingIatas = cityService
            .getAirportsByCityName(searchRequest
                    .getSourceCity())
            .stream()
            .map(AirportDto::getIata)
            .collect(Collectors.toList());

    List<String> landingIatas = cityService
            .getAirportsByCityName(searchRequest
                    .getDestinationCity())
            .stream()
            .map(AirportDto::getIata)
            .collect(Collectors.toList());

    List<FlightDto> resultFlights = startingIatas
            .stream()
            .map(flightRepository::findBySourceIata)
            .flatMap(Collection::stream)
            .filter(flight -> landingIatas.contains(
                    flight.getDestination().getIata()))
            .filter(flight -> flight
                    .getStartTime().isAfter(minTime))
            .filter(flight -> flight
                    .getStartTime().isBefore(maxTime))
            .map(flightMapper::toDto)
            .collect(Collectors.toList());

    return resultFlights;
}
```

6.3.4 Wnioski z implementacji

Simple Scaffolding jest narzędziem bardzo prostym, ale to właśnie prostota jest jego główną zaletą i założeniem przyjętym przez jego twórcę. Do jego użycia potrzebny jest już pewien gotowy fragment kodu, ale dzięki temu łatwo może być wykorzystany także kiedy projekt jest już dosyć rozbudowany. Nie generuje niczego „od zera”, ale dzięki temu nie narzuca wyboru żadnej konkretnej architektury ani narzędzi. Nie oferuje żadnych opcji konfiguracji. Jego celem jest uwolnienie programisty od zapisywania powtarzalnych, podobnych do siebie fragmentów kodu, dzięki czemu programista może skupić się na implementacji logiki biznesowej. Włączenie Simple Scaffolding do projektu nie jest poważną decyzją architektoniczną i nie niesie ze sobą daleko idących konsekwencji, tak jak w przypadku JHipster czy jOOQ. Jest on raczej narzędziem, które programista pracujący nad rozwojem kodu może wybrać, aby ułatwić swoją pracę. Decyzja ta nie wpłynie na produktywność innych osób, gdyż Simple Scaffolding praktycznie nie pozostawia śladów po swoim działaniu.

7 Analiza porównawcza

Poniższy rozdział zawiera zestawienie statystyk, pomiarów oraz wniosków wynikających z pracy z wybranymi technologiami scaffoldingowymi. Wykonane aplikacje przebadano pod względem czasów generacji, objętości kodu, wydajności stworzonych aplikacji oraz łatwości użycia.

7.1.1 Czasy generacji

Tabela 2 przedstawia wyniki pomiarów czasów generacji dla trzech wykonanych aplikacji testowych. Dla JHipster oraz Simple Scaffolding wyszczególniono wyniki cząstkowe dla dwóch etapów generacji, natomiast w jOOQ podany jest tylko łączny wynik, gdyż w przypadku jOOQ generacja jest jednoetapowa – następuje tylko przy pełnej kompilacji projektu.

Tabela 2 Czasy generacji dla poszczególnych aplikacji testowych

JHipster	**Czas (mm:ss:ms)**	**jOOQ**	**Czas (mm:ss:ms)**	**Simple Scaffolding**	**Czas (mm:ss:ms)**
Gen. bez modelu	04:28:94	-	-	Tryb odczytu	00:11:34
import jdl	01:08:56	-	-	Tryb zapisu	00:05:42
Razem	**05:37:50**	**Razem**	**00:24:52**	**Razem**	**00:16:76**

Jak widać, czas generacji jest zdecydowanie najdłuższy dla JHipster. Wynik nie jest zaskoczeniem, ponieważ program ten generuje zdecydowanie najwięcej kodu (należy uwzględnić, że JHipster tworzy od zera strukturę projektu, a dodatkowo generuje również frontend). Trzeba jednak zauważyć, że generacja jest procesem jednorazowym i we wszystkich przypadkach jest ona na tyle szybka, że skuteczna praca z narzędziem nie jest zagrożona.

7.1.2 Objętość kodu

Tabela 3 przestawia sumaryczną wielkość projektów oraz liczbę linijek kodu, które musiały zostać dopisane na potrzeby realizacji wymagań. W zestawieniu nie zawarto frontendu aplikacji jOOQ oraz Simple Scaffolding, ponieważ, jak wspominano wcześniej, musiał on być napisany „od zera". Pomijając strukturę programu React wygenerowaną przez narzędzie Create React App (opisane w rozdziale 6.2.4 „Samodzielna implementacja frontendu"), liczba dopisanych linii kodu frontendu dla tych aplikacji wyniosła 867.

Tabela 3 Objętość poszczególnych projektów wyrażona w liniach kodu

Projekt	**Łączna objętość kodu w linijkach**	**Liczba dopisanych linii kodu**
JHipster backend	8 414	331
JHipster frontend	7 702	27
jOOQ backend	1 680	722
Simple Scaffolding	3 036	653

Tu znów wyraźnie największym projektem spośród utworzonych jest program napisany przy pomocy JHipster. Jego objętość przekracza osiem tysięcy linii. Wielkość ta nie budzi jednak zastrzeżeń ze względu na największą liczbę funkcjonalności, które dostarcza JHipster (zabezpieczenia, filtrowanie danych, logowanie, monitorowanie stanu aplikacji itp). Aplikacja jOOQ jest znacząco mniejsza nawet od Simple Scaffolding. Fakt ten jest efektem ubocznym tego, że jOOQ nie generuje struktur serwisów i kontrolerów, a więc siłą rzeczy nie jest tworzony nadmiarowy kod.

Z kolei, jeśli przyjrzeć się wielkości dopisanego kodu, wtedy JHipster jest zdecydowanym faworytem pod względem minimalizacji wymaganego nakładu pracy. Można dzięki temu wyciągnąć wniosek, że mimo iż JHipster generuje dużo kodu, z czego znacząca część może rzeczywiście okazać się nadmiarowa, to kod ten jest użyteczny, jak również łatwy do modyfikacji i do przystosowania oraz wykorzystania na potrzeby bieżących wymagań projektowych.

Liczba koniecznych do dopisania linii kodu jest podobna w przypadku jOOQ i Simple Scaffolding, jednak w przypadku jOOQ nakład pracy był odrobinę większy. Tak samo jak w wyniku łącznej objętości kodu wynika to z faktu, że jOOQ dostarcza najmniej kodu odpowiedzialnego za ostateczne funkcjonowanie aplikacji webowej, w związku z czym programista miał najwięcej pracy tworząc aplikację przy użyciu tego narzędzia.

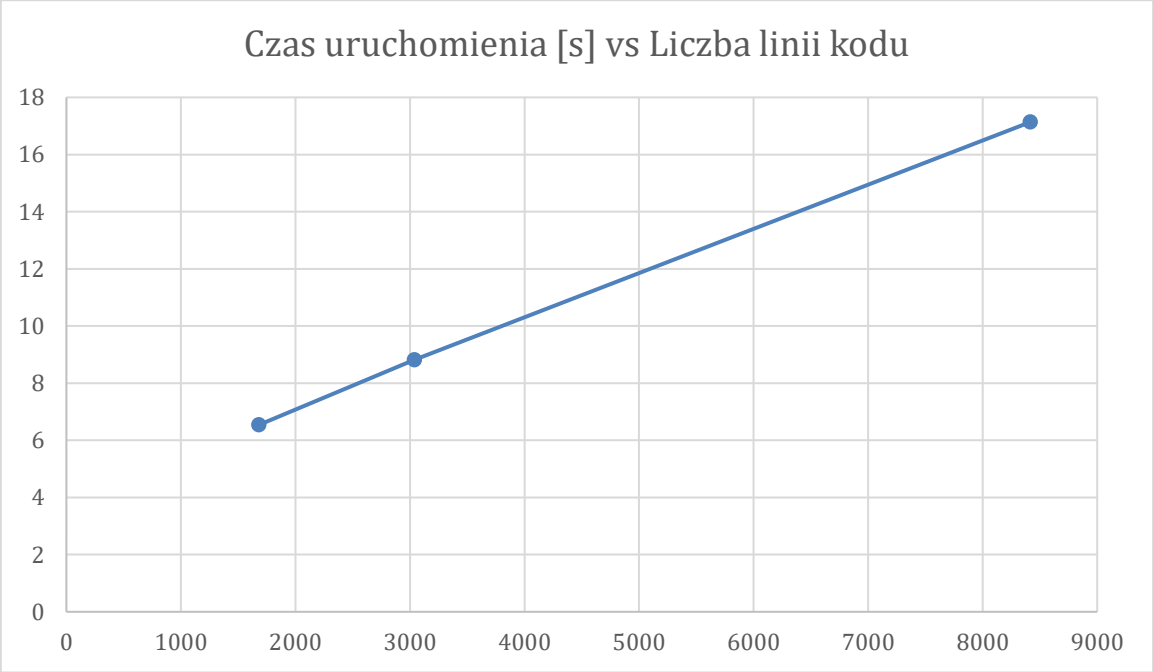
7.1.3 Wydajność

Czas potrzebny do uruchomienia każdej z przygotowanych aplikacji prezentuje Tabela 4. Nie jest zaskoczeniem, że najwięcej czasu na uruchomienie potrzebuje największa aplikacja, czyli JHipster.

Tabela 4 Czasy uruchomień aplikacji testowych

Aplikacja	**Czas uruchomienia [s]**
JHipster	17.133
jOOQ	6.539
Simple Scaffolding	8.818

Czasy uruchomienia nie zajmują bardzo długo i nie utrudniają sprawnego rozwoju aplikacji. Jeśliby pokusić się o zbadanie zależności pomiędzy wielkością aplikacji a czasem jej uruchomienia, można stwierdzić, że druga wielkość jest niemalże wprost proporcjonalna do pierwszej (patrz Rysunek 28). Z tego wynika, że wynik JHipster relatywnie nie jest gorszy od pozostałych aplikacji.



Rysunek 28 Czas uruchomienia aplikacji w zależności od jej wielkości

Na potrzeby porównania wydajności stworzonych programów wykonano testy z użyciem narzędzia JMeter. Wszystkie trzy sesje testowe przebiegały według tego samego scenariusza:

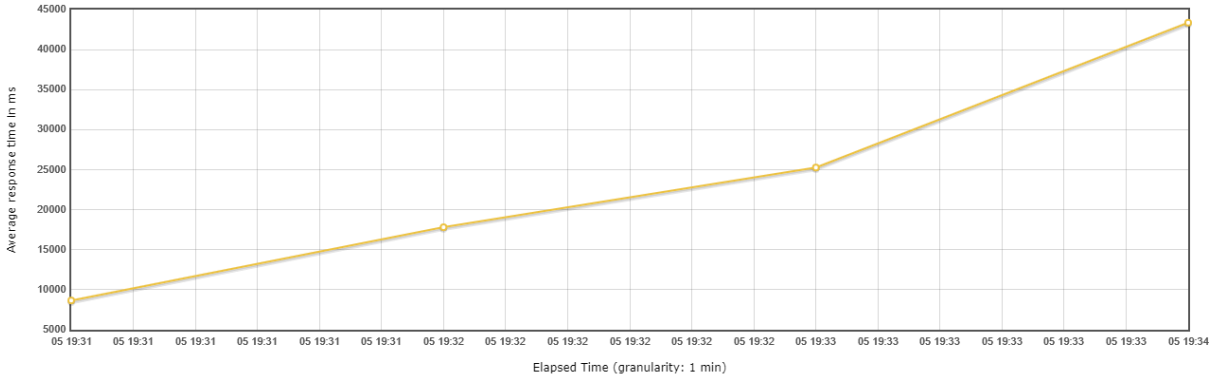
1. Na początek utworzonych zostaje 1 000 wątków symulujących użytkowników aplikacji
2. Każdy użytkownik najpierw wyszukuje lot według przykładowych kryteriów
3. Następnie użytkownik zapisuje pasażera
4. Całość jest powtarzana pięciokrotnie

Zatem w sumie każda z aplikacji zostanie obciążona dziesięcioma tysiącami zapytań w dość krótkim czasie. Tabela 5 przedstawia dokładne czasy trwania testów.

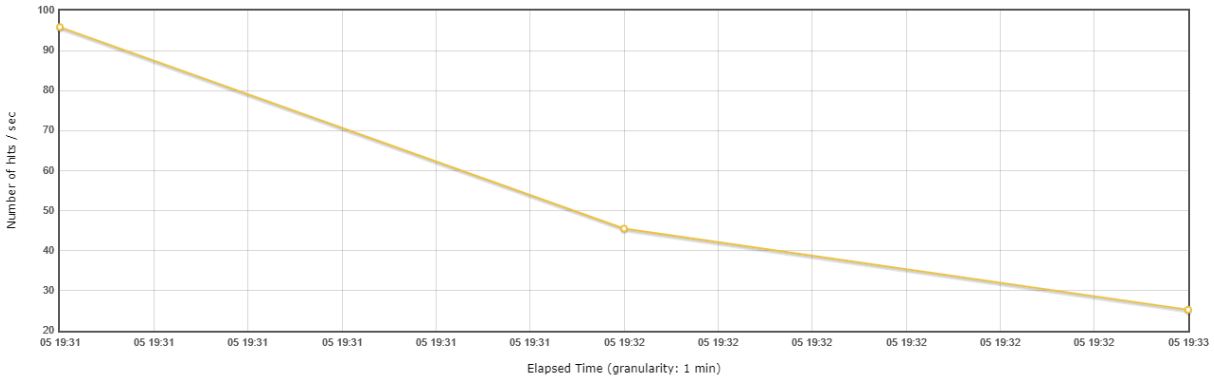
Tabela 5 Czasy wykonywania testów wydajnościowych przez aplikacje testowe

Aplikacja	**Czas wykonywania testu [ms]**
JHipster	141 159 ≈ 2.35 min
jOOQ	29 358 ≈ 0.48 min
Simple Scaffolding	161 751 ≈ 2.69 min

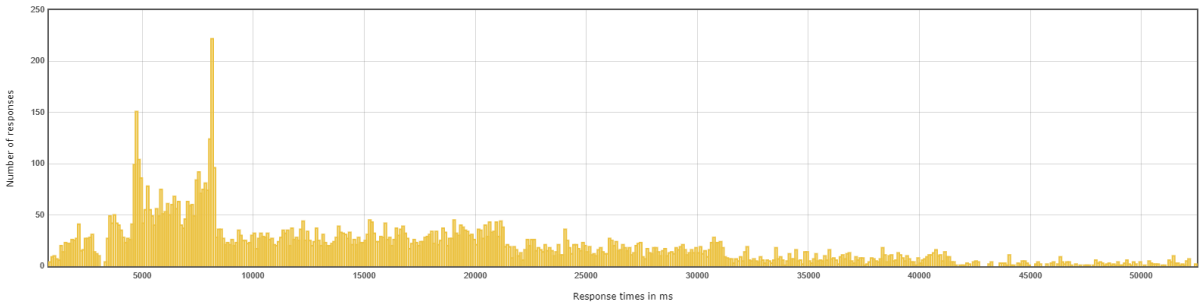
Na grafikach opatrzonych etykietami od „Rysunek 29” do „Rysunek 37” zawarto wybrane wykresy wygenerowane przez JMeter na podstawie przeprowadzonych eksperymentów. Pierwszy wykres przedstawia średni czas odpowiedzi wyliczany w kolejnych momentach trwania testu. Warto porównać go z drugim wykresem, na którym przedstawiona jest liczba zapytań wysyłanych w danej chwili, aby zobaczyć, jak liczba zapytań wiązała się z czasem odpowiedzi aplikacji. Trzeci wykres w serii prezentuje rozkład zapytań na podstawie czasu odpowiedzi. Pozwala on porównać ogólną wydajność programów.



Rysunek 29 JHipster: Średni czas odpowiedzi w zależności od czasu

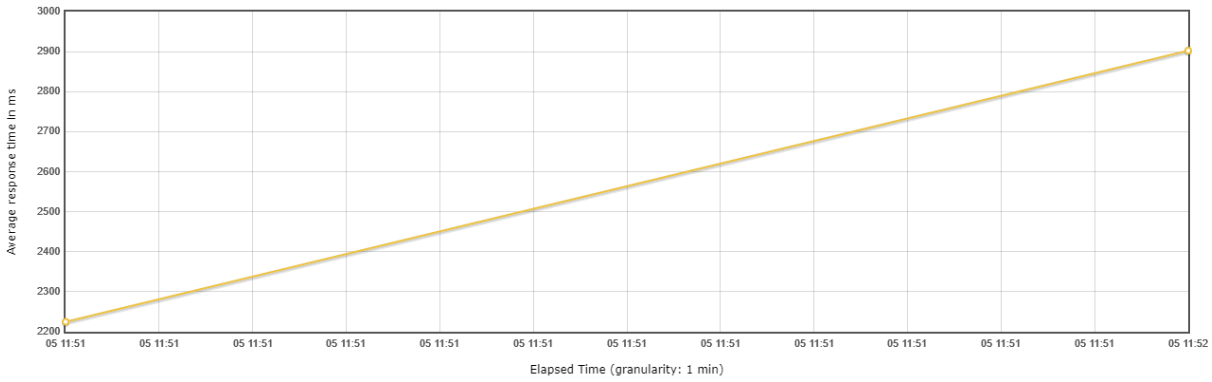


Rysunek 30 JHipster: Liczba zapytań w zależności od czasu

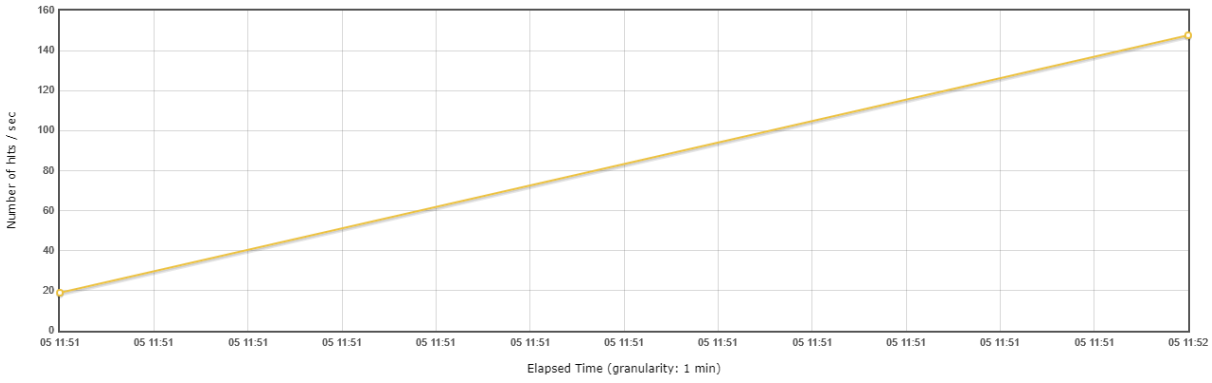


Rysunek 31 JHipster: Histogram przedstawiający rozkład czasów odpowiedzi

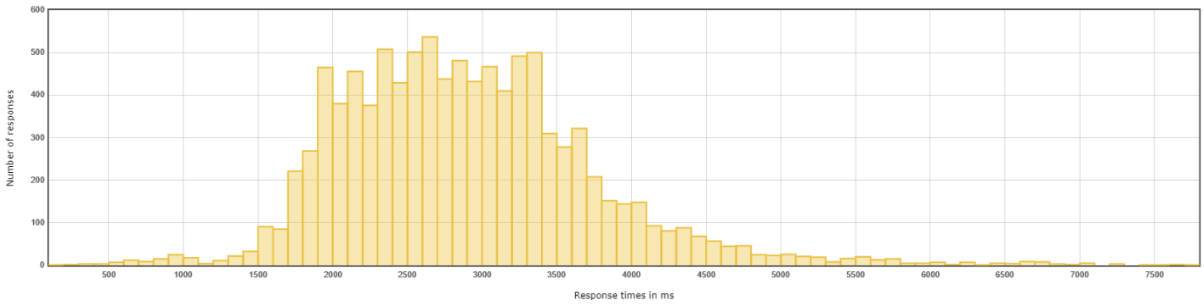
Jak widać z wykresu pierwszego (Rysunek 29), czas odpowiedzi ze strony aplikacji JHipster rósł wraz z przebiegiem eksperymentu, mimo że liczba zapytań malała (Rysunek 30). Można więc wnioskować, że w początkowej fazie testu aplikacja została „zalana" zapytaniami, których nie nadążała obsługiwać, w związku z czym czas odpowiedzi wzrastał. Liczba zapytań malała stopniowo, dlatego że pierwsi klienci, którzy wysłali zapytania zostali obsłużeni najszybciej. Z trzeciego wykresu (Rysunek 31) można wyczytać, że najwięcej zapytań oscyluje wokół wartości 5 000 ms i 8 000 ms.



Rysunek 32 jOOQ: Średni czas odpowiedzi w zależności od czasu

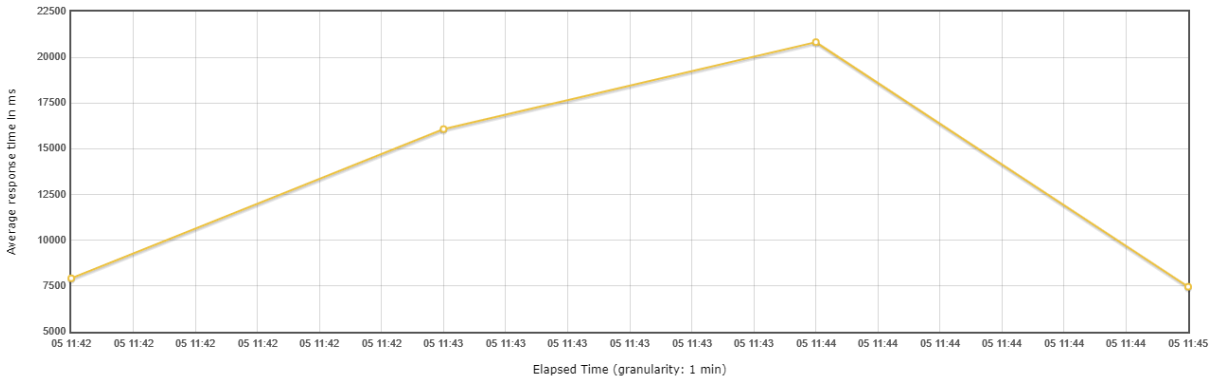


Rysunek 33 jOOQ: Liczba zapytań w zależności od czasu

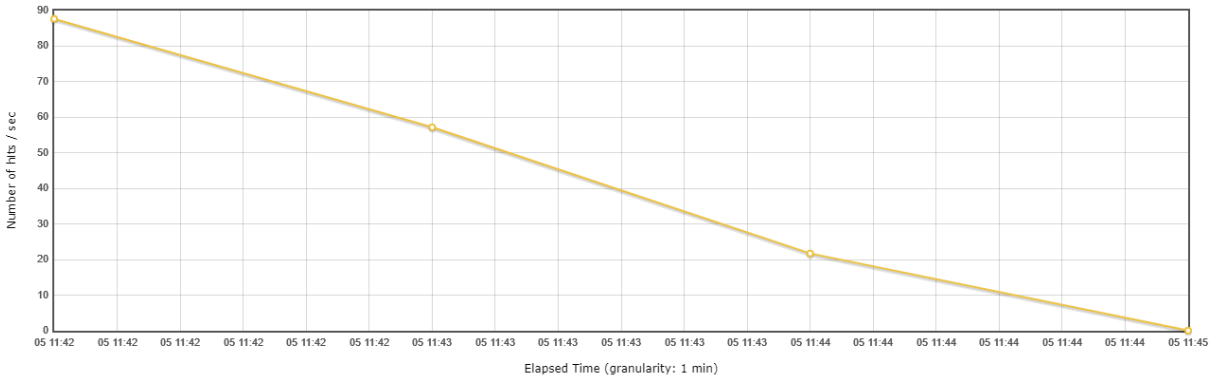


Rysunek 34 jOOQ: Histogram przedstawiający rozkład czasów odpowiedzi

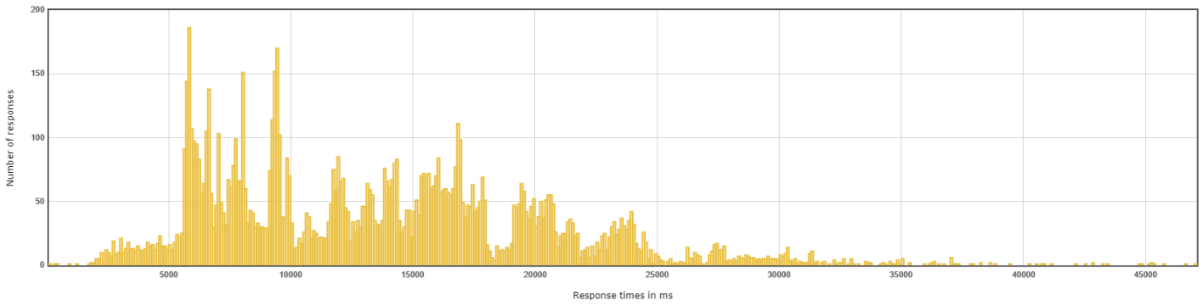
W przypadku jOOQ mamy do czynienia z najlepszą wydajnością aplikacji, co widać zarówno w najniższym czasie wykonywania eksperymentu (Tabela 5), jak i na rozkładzie czasów zapytań (Rysunek 34), który mieści się pomiędzy wartościami 2 000 ms i 3 000 ms. Tak jak przy poprzedniej aplikacji, średni czas odpowiedzi wzrastał (Rysunek 32), jednak nie przekroczył poziomu 3 000 ms. Pozwala to przypuszczać, że wszystkie zapytania były obsługiwane na bieżąco i w przeciwieństwie do JHipster nie doświadczyliśmy stopniowego zmniejszania się liczby zapytań w czasie (Rysunek 33).



Rysunek 35 Simple Scaffolding: Średni czas odpowiedzi w zależności od czasu



Rysunek 36 Simple Scaffolding: Liczba zapytań w zależności od czasu



Rysunek 37 Simple Scaffolding: Histogram przedstawiający rozkład czasów odpowiedzi

Czas wykonywania testów dla Simple Scaffolding był nieznacznie dłuższy niż w przypadku JHipster. Mniejszą wydajność widać również na histogramie (Rysunek 37) – większość czasów reakcji przekracza 5 000 ms. Przebieg liczby zapytań (Rysunek 36) wygląda podobnie do analogicznego wykresu dla JHipster, natomiast na wykresie czasu oczekiwania (Rysunek 35) widać, że osiągnął on w pewnym momencie swój szczyt, po czym zaczął spadać.

Dlaczego jOOQ wykazał największą wydajność? Na część tego wyniku może składać się fakt, że odpytywanie bazy danych jest lepiej zoptymalizowane w przypadku jOOQ niż Hibernate (przynajmniej przy domyślnych ustawieniach i dla tego konkretnego przypadku testowego). Jednakże dostępne w sieci porównania wydajności między jOOQ a czystym JDBC oraz między Hibernate a JDBC wskazują na to, że pod tym względem różnica powinna być niewielka [72] [73]. Najprawdopodobniej główny wkład w wydajność aplikacji jOOQ leżał w liczbie przesyłanych obiektów. Podczas gdy pozostałe aplikacje dokonywały przynajmniej części filtrowania w pamięci operacyjnej, jOOQ przygotował zapytanie SQL pozwalające na uzyskanie wyników gotowych do zwrócenia użytkownikowi.

7.1.4 Udział generatora w spełnieniu wymagań

Poniższy podrozdział zawiera oszacowanie tego, jak duży udział miał każdy z generatorów w tworzeniu funkcjonalności zdefiniowanych w rozdziale 5.1. Tabela 6 przedstawia podsumowanie wymagań wraz z luźnym szacunkiem tego, jak bardzo dane narzędzie pomogło spełnić każde z nich.

Tabela 6 Udział scaffolderów w spełnieniu wymagań aplikacji

Funkcjonalność	**JHipster**	**jOOQ**	**Simple Scaffolding**
formularz wyszukiwania lotu	90%	-	-
tabela do przeglądania wyników wyszukiwania	90%	-	-
formularz rezerwacji lotu	50%	-	-
podgląd zarezerwowanych lotów	50%	-	-
pobieranie informacji o portach lotniczych	90%	30%	90%
pobieranie informacji o miastach i portach które się w nich znajdują	90%	30%	50%
pobieranie i edycja informacji o planowanych lotach	90%	30%	90%
wyszukiwanie lotów	-	-	-
pobieranie i edycja informacji o pasażerach	90%	30%	90%
logowanie	100%	-	-

W kolumnie JHipster oraz Simple Scaffolding wartością 90% oznaczono zadania, które można było wykonać dokonując niewielkich przeróbek, takich jak dodanie kryterium wyszukiwania, czy przekierowanie adresu strony. Przy jOOQ szereg wymagań oznaczono wartością 30%, gdyż dla każdego obiektu zostały wygenerowane klasy dostępu do danych (DAO), ale dla żadnej nie uzyskano na drodze generacji serwisów ani kontrolerów.

7.1.5 Jakość generowanego kodu

Porównanie jakości kodu w odniesieniu do implementowanych wzorców projektowych (np. MVC) byłoby trudne, ze względu na to, że omawiane projekty generują aplikację w różnych zakresach. Cała struktura aplikacji powstaje tylko w przypadku JHipster i tu można powiedzieć, że widoczny jest wyraźny podział klas na komponenty odpowiedzialne za model, przetwarzanie i składowanie modelu, obsługa zapytań i wyświetlanie widoku (w postaci strony React). Frontend podporządkowuje się dobrym praktykom tworzenia architektury Flux (opisanej w rozdziale 4.5) używając biblioteki Redux do przechowywania stanu aplikacji.

Co do kodu generowanego w jOOQ, można odnieść się do ogólnych paradygmatów tworzenia poprawnego kodu obiektowego, takich jak SOLID. Teorię stanowiącą podwaliny tych zasad sformułował Robert Martin [74]. SOLID to akronim, którego litery pochodzą od pięciu kryteriów kierujących tworzeniem czytelnego, efektywnego i reużywalnego oprogramowania. Te zasady to:

1. Single Responsibility Principle – według tej zasady każda klasa powinna być odpowiedzialna za tylko jeden rodzaj operacji przeprowadzanych na danych
2. Open-Closed Principle – zasada ta stanowi, że tworzone moduły oprogramowania powinny umożliwiać i być otwarte na rozszerzenie funkcjonalności bez konieczności swojej modyfikacji
3. Liskov Substitution Principle – zgodnie z tą zasadą, obiekty klasy bazowej powinny być zastępowalne przez obiekty podklasy; innymi słowy, metody klasy pochodnej nie powinny zmieniać funkcjonalności metod klasy bazowej
4. Interface Segregation Principle – jest to reguła mówiąca o tym, by unikać dużych interfejsów reprezentujących szeroki wachlarz funkcjonalności i zamiast tego faworyzować mniejsze, reprezentujące pojedyncze zadania
5. Dependency Inversion Principle – zasada zalecająca odwrócenie zależności przy projektowaniu kodu, tak aby planując projekt mieć na względzie w pierwszej kolejności wysokopoziomowe abstrakcje, a dopiero potem konkretne implementacje

jOOQ generując DAO dla konkretnych encji korzysta z opisanej poniżej struktury. Kontrakt opisujący funkcjonowanie charakterystyczne dla tego typu klas jest zawarty w najbardziej abstrakcyjnym interfejsie. W związku z tym spełniona zostaje Dependency Inversion Principle, jako że każda klasa implementująca ten interfejs musi dostarczać odpowiednich metod. Dyskusyjnym mogłoby być to czy spełnione są zasady Single Responsibility Principle oraz Interface Segregation Principle gdyż interfejs DAO zawiera różne typy operacji na danych (zapis, zmiana, usuwanie, odczyt). Jest to jednak powszechnie stosowana praktyka przy tworzeniu aplikacji biznesowych. Metody, których działanie jest wspólne dla wszystkich rodzajów encji, takie jak `getById`, `count` itp. są implementowane przez ogólną klasę `DAOImpl`, co znowu wzmacnia stosowanie Dependency Inversion Principle w projekcie. Dodatkowo umożliwia to stosowanie Open-Closed Principle, ponieważ poszerzenie tych ogólnych funkcjonalności zostanie wykonane w konkretnych klasach `AirportDAO` itp. które rozszerzają `DAOImpl`. W związku z tym można stwierdzić, że kod generowany przez jOOQ jest wysokiej jakości programem obiektowym.

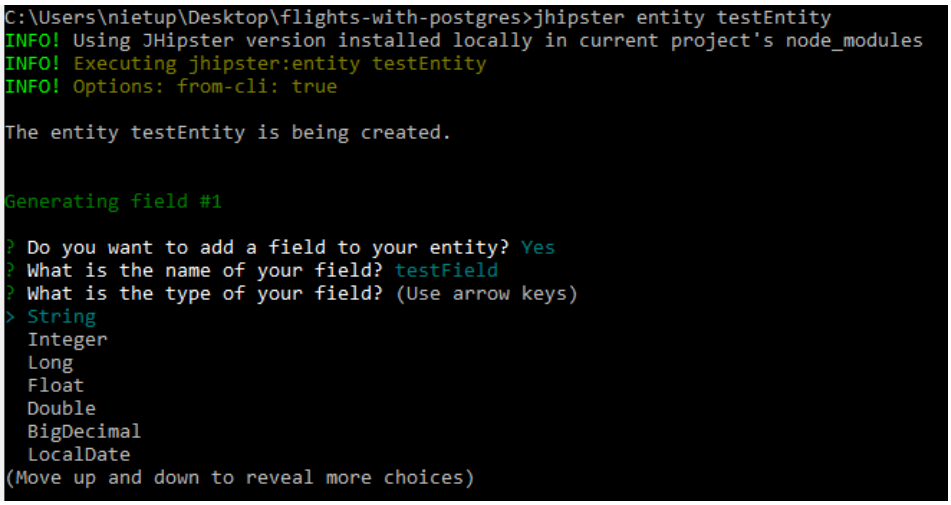
7.1.6 Łatwość użycia – dokumentacja

Dokumentacje użytych narzędzi różnią się swoją obszernością, ponieważ różny jest także poziom ich złożoności i liczba możliwych opcji. Jednak dla każdego narzędzia dokumentacja jest dość dokładna by bez przeszkód móc korzystać z tego oprogramowania. Dla wszystkich trzech projektów pełna dokumentacja jest dostępna online [4] [12] [18].

7.1.7 Łatwość modyfikacji

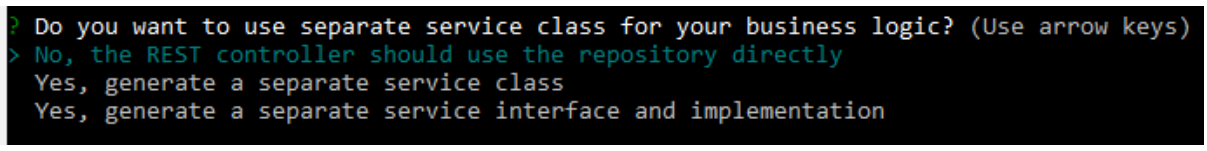
Poniższy podrozdział opisuje łatwość modyfikacji rozumianej jako użycie danego generatora po wprowadzeniu pewnych zmian do projektu – na przykład, gdy po pewnym czasie okazałoby się, że trzeba dodać nową encję do modelu.

JHipster oferuje możliwość „dogenerowania" dodatkowych encji oraz zmiany własności istniejących, np. dodanie atrybutu. Trzeba jednak pamiętać, że modyfikowanie istniejących encji wiąże się z regeneracją wszystkich plików z nimi związanych – a więc także ze stratą zmian, które zostały wprowadzone ręcznie.



Rysunek 38 Okno interfejsu generacji encji w JHipster

Rysunek 38 przedstawia proces dodawania nowej encji w trybie interaktywnym (w przeciwieństwie do modyfikowania pliku JDL). Użytkownikowi zadawana jest seria pytań dotyczących nazwy encji, nazw i typów jej pól itp. Użytkownik ma też wybór tego, jakie warstwy aplikacji chce wygenerować (Rysunek 39).



Rysunek 39 Okno interfejsu generacji encji w JHipster (c.d.)

W przypadku jOOQ sytuacja jest odmienna. Tu klasy generowane są na podstawie tabel w bazie danych. Model jest re-generowany przy każdej pełnej kompilacji projektu (`mvn clean install`). Dodanie nowych/zmiana istniejących encji jest więc uzależnione tylko od tego, czy pozwala na to przyjęty system bazodanowy i zależności w modelu danych. Po ponownej generacji żadne z klas stworzonych przez programistę nie zostaną zmodyfikowane. Błąd może wystąpić tylko w przypadku,

gdy zmiana naruszy istniejące w kodzie operacje dotyczące generowanych obiektów, na przykład, gdy usunięte zostanie jakieś pole wykorzystywane w programie.

W Simple Scaffolding generacja wykonywana jest na żądanie, poprzez uruchomienie klasy scaffoldera. Nie narzuca on żadnych ograniczeń dotyczących momentu ani liczby użyć. Programista może w dowolnym momencie utworzyć nowe szablony według swoich potrzeb lub generować kolejne obiekty na podstawie zapisanych szablonów. Jeśli nazwy generowanych plików kolidowałyby z istniejącymi, program zatrzyma generację tych konkretnych plików, aby nie utracić danych (Rysunek 40).

```
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Writing new code from templates
Extracted templates: 6
profilePath:default/
Skipping 'src/main/java/mgr/flights/simplescaffolding/flight/FlightService
 .java' to prevent an overwrite.
```

Rysunek 40 Simple Scaffolding powstrzyma programistę przed nadpisaniem swoich danych

8 Podsumowanie

W ramach niniejszej pracy przeprowadzono analizę scaffoldingu jako wsparcia w tworzeniu aplikacji internetowych przy wykorzystaniu Javy jako głównego języka implementacji. Zbadano obecny stan sztuki w tym zakresie oraz wykonano trzy aplikacje testowe pozwalające na porównanie różnych podejść do tworzenia narzędzi scaffoldingowych. Projekty, z wykorzystaniem których stworzono prototypy, to: JHipster, jOOQ oraz Simple Scaffolding.

JHipster jest narzędziem bardzo zaawansowanym. Pozwala na generację rozbudowanej aplikacji posiadającej frontend, autoryzację, metody zapisu i modyfikacji danych, rozbudowane możliwości wyszukiwania obiektów i wiele innych. Tworzy ciężki projekt, zawierający od początku dużą ilość kodu. Jako że każdy projekt informatyczny ma swoje unikalne potrzeby i wymagania, część kodu tworzonego przez JHipster musi okazać się nadmiarowa. Mimo to, analiza nakładu pracy potrzebnego do spełnienia założeń aplikacji testowej pokazała, że metody generowane przez JHipster są użyteczne i łatwe do wykorzystania przy wprowadzaniu zmian. Dzięki temu stosunkowo niskim kosztem można było uzyskać gotowy program. Kolejnym aspektem wynikającym z faktu, że JHipster generuje dużą aplikację początkową jest to, że tworzy on przez to wiele zależności pomiędzy poszczególnymi modułami i klasami, przez co rozwój aplikacji może być utrudniony. Powyższa charakterystyka sprawia, że JHipster wydaje się przede wszystkim świetnym narzędziem do szybkiego tworzenia działającego prototypu aplikacji webowej.

jOOQ jest projektem oferującym alternatywne podejście do współpracy z bazą danych. Pozwala to w szczególności na efektywne prowadzenie odczytów i zapytań analitycznych. Dzięki temu aplikacja powstała przy użyciu jOOQ odznaczała się zdecydowanie najwyższą wydajnością. Filozofia tego projektu (baza danych żyje dłużej niż aplikacja; model danych kształtuje design programu) unaocznia się również w możliwościach scaffoldera załączonego do jOOQ – jego celem jest odzwierciedlenie struktury bazy danych. Nie pozwala na generację pewnych funkcjonalności niezbędnych dla aplikacji webowej, takich jak wystawienie interfejsu sieciowego. Dzięki temu praca z tym narzędziem może być połączona np. z Simple Scaffolding

Simple Scaffolding to bardzo lekki program. Powinien być traktowany bardziej jako doraźna pomoc dla programisty niż zaplanowany element stosu technologicznego projektu, ponieważ jego wykorzystanie jest praktycznie „niewidzialne” – nie nakłada żadnych wymogów odnośnie architektury ani sposobu pracy programistów biorących udział w projekcie. Jego celem jest umożliwienie szybkiego powielania istniejących struktur w kodzie, przez co prawdopodobnie zapewnia największy wzrost produktywności w momencie, gdy przygotowany został tzw. vertical slice (ang. przekrój pionowy – pełen zakres funkcjonalności aplikacji przygotowany dla wąskiego przypadku lub pojedynczego typu obiektów). W takiej sytuacji pozwoli on na szybkie przystosowanie metod na resztę encji/obsługiwanych sytuacji.

Podsumowując należy podkreślić, że scaffolding może stanowić istotne wsparcie w tworzeniu aplikacji internetowych. Architekt powinien jednak zdawać sobie sprawę z różnych podejść przyjętych przez twórców narzędzi scaffoldingowych i wybrać projekt odpowiadający swoim potrzebom. Celem niniejszej pracy było ułatwienie tego wyboru.

9 Bibliografia

[1] "TIOBE programming community index," [Online]. Url: https://www.tiobe.com/tiobe-index/. [Dostęp 05 12 2019].

[2] D. Acemoglu, D. Autor, D. Dorn, G. H. Hanson and B. Price, Return of the Solow Paradox?, vol. NBER Working Paper Series, Cambridge: NBER Working Paper Series, 2014.

[3] "Scaffold (programming)," [Online]. Url: https://en.wikipedia.org/wiki/Scaffold_(programming). [Dostęp 05 09 2019].

[4] "Dokumentacja JHipster," [Online]. Url: https://www.jhipster.tech/. [Dostęp 05 09 2019].

[5] "JHipster License Agreement," [Online]. Url: https://github.com/jhipster/generator-jhipster/blob/master/LICENSE.txt. [Dostęp 06 09 2019].

[6] "Dokumentacja Spring Roo," [Online]. Url: https://projects.spring.io/spring-roo/. [Dostęp 05 09 2019].

[7] "Screenshot Roo Shell," [Online]. Url: https://docs.spring.io/spring-roo/docs/2.0.x/reference/html/images/sts-project-settings.png. [Dostęp 15 12 2019].

[8] "Spring Roo License Agreement," [Online]. Url: https://github.com/spring-projects/spring-roo/blob/master/LICENSE.TXT. [Dostęp 06 09 2019].

[9] "Dokumentacja Open Xava," [Online]. Url: https://openxava.org/. [Dostęp 05 09 2019].

[10] "Modeling with Java - Open Xava Documentation," [Online]. Url: https://www.openxava.org/OpenXavaDoc/docs/modeling_en.html. [Dostęp 15 12 2019].

[11] "Open Xava License Agreement," [Online]. Url: https://github.com/openxava/openxava/blob/master/OpenXava/license.txt. [Dostęp 06 09 2019].

[12] "Dokumentacja jOOQ," [Online]. Url: https://www.jooq.org/. [Dostęp 05 09 2019].

[13] "jOOQ Licenses," [Online]. Url: https://www.jooq.org/legal/licensing. [Dostęp 06 09 2019].

[14] "Dokumentacja Jspresso," [Online]. Url: http://www.jspresso.org/. [Dostęp 05 09 2019].

[15] "Authoring SJS source files - Examples," [Online]. Url: https://jspresso.gitbooks.io/sjs-doc/en/chapter1.html#examples. [Dostęp 15 12 2019].

[16] "Jspresso License Agreement," [Online]. Url: https://github.com/jspresso/jspresso-ce/blob/master/LICENSE.txt. [Dostęp 06 09 2019].

[17] "Dokumentacja Generjee," [Online]. Url: http://generjee.com/. [Dostęp 05 09 2019].

[18] "Simple Scaffolding," [Online]. Url: http://gary-rowe.com/agilestack/2013/05/19/simple-scaffolding/. [Dostęp 06 09 2019].

[19] "Simple Scaffolding License Agreement," [Online]. Url: https://github.com/gary-rowe/SimpleScaffolding/blob/master/MIT-LICENSE. [Dostęp 06 09 2019].

[20] "Dokumentacja Telosys," [Online]. Url: http://www.telosys.org/. [Dostęp 06 09 2019].

[21] "Schemat architektury Telosys," [Online]. Url: https://www.telosys.org/img/templatesPage/multi-bundles.png. [Dostęp 15 12 2019].

[22] "Telosys License Agreement," [Online]. Url: https://github.com/telosys-tools-bricks/telosys-tools-all/blob/master/LICENSE. [Dostęp 06 09 2019].

[23] "Dokumentacja UI Scaffolding," [Online]. Url: https://forge.jboss.org/1.x/docs/important_plugins/ui-scaffolding.html. [Dostęp 06 09 2019].

[24] "JBoss Forge License Agreement," [Online]. Url: https://github.com/forge/core/blob/master/LICENSE. [Dostęp 06 09 2019].

[25] "jOOQ as a SQL builder with code generation," [Online]. Url: https://www.jooq.org/doc/3.12/manual/getting-started/use-cases/jooq-as-a-sql-builder-with-code-generation/. [Dostęp 06 09 2019].

[26] A. Binstock, "Java's 20 Years Of Innovation," 23 05 2015. [Online]. Url: https://www.forbes.com/sites/oracle/2015/05/20/javas-20-years-of-innovation/#431faa11d7cb. [Dostęp 15 12 2019].

[27] "The Java Language Environment," [Online]. Url: https://www.oracle.com/technetwork/java/intro-141325.html. [Dostęp 15 12 2019].

[28] "Poradnik Programowania na systemy Android," [Online]. Url: https://developer.android.com/guide. [Dostęp 15 12 2019].

[29] "Netscape and Sun announce JavaScript," 12 4 1995. [Online]. Url: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html. [Dostęp 15 12 2019].

[30] "Dokumentacja Node.js," [Online]. Url: https://nodejs.org/en/docs/. [Dostęp 15 12 2019].

[31] "Dokumentacja ECMAScript," [Online]. Url: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources. [Dostęp 15 12 2019].

[32] "IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains," [Online]. Url: https://www.jetbrains.com/idea/. [Dostęp 15 12 2019].

[33] "Intellij License Agreement," [Online]. Url: https://github.com/JetBrains/intellij-community/blob/master/LICENSE.txt. [Dostęp 15 12 2019].

[34] "A Short History of Git," [Online]. Url: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git. [Dostęp 15 12 2019].

[35] "Spring Framework," [Online]. Url: https://spring.io/projects/spring-framework. [Dostęp 07 12 2019].

[36] "Dokumentacja Spring Boot," [Online]. Url: https://docs.spring.io/spring-boot/docs/2.2.2.RELEASE/reference/html/getting-started.html#getting-started. [Dostęp 07 12 2019].

[37] "Introduction to Spring Framework," [Online]. Url: https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/overview.html. [Dostęp 07 12 2019].

[38] "React - A JavaScript library for building user interfaces," [Online]. Url: https://reactjs.org/. [Dostęp 07 12 2019].

[39] "Flux in depth overview," [Online]. Url: https://facebook.github.io/flux/docs/in-depth-overview. [Dostęp 08 12 2019].

[40] "Docker concepts," [Online]. Url: https://docs.docker.com/get-started/. [Dostęp 29 12 2019].

[41] "Auth0 Overview," [Online]. Url: https://auth0.com/docs/getting-started/overview. [Dostęp 29 12 2019].

[42] "Dokumentacja PostgreSQL," [Online]. Url: https://www.postgresql.org/docs/12/index.html. [Dostęp 08 12 2019].

[43] "Apache JMeter," [Online]. Url: https://jmeter.apache.org/. [Dostęp 6 1 2020].

[44] "JHipster Video Tutorial," [Online]. Url: https://www.jhipster.tech/video-tutorial/. [Dostęp 09 12 2019].

[45] "JSON Web Token (JWT) RFC 7519 IETF Standard," [Online]. Url: https://tools.ietf.org/html/rfc7519. [Dostęp 15 12 2019].

[46] "Dokumentacja Spring Session," [Online]. Url: https://docs.spring.io/spring-session/docs/current/reference/html5/guides/java-rest.html. [Dostęp 15 12 2019].

[47] "OAuth 2.0 Authorization Framework RFC 6749 IETF Standard," [Online]. Url: https://tools.ietf.org/html/rfc6749. [Dostęp 15 12 2019].

[48] "Dokumentacja MySQL," [Online]. Url: https://dev.mysql.com/doc/.

[49] "Dokumentacja MariaDB," [Online]. Url: https://mariadb.org/documentation/. [Dostęp 15 12 2019].

[50] "Dokumentacja Oracle Database," [Online]. Url: https://docs.oracle.com/en/database/oracle/oracle-database/19/index.html. [Dostęp 15 12 2019].

[51] "Dokumentacja MS SQL Server," [Online]. Url: https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15. [Dostęp 15 12 2019].

[52] "Dokumentacja MongoDB," [Online]. Url: https://docs.mongodb.com/manual/. [Dostęp 15 12 2019].

[53] "Dokumentacja Apache Cassandra," [Online]. Url: https://cassandra.apache.org/doc/latest/. [Dostęp 15 12 2019].

[54] "Dokumentacja Couchbase," [Online]. Url: https://docs.couchbase.com/home/index.html. [Dostęp 15 12 2019].

[55] "Dokumentacja Elasticsearch," [Online]. Url: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html. [Dostęp 15 12 2019].

[56] "Dokumentacja Spring WebSocket," [Online]. Url: https://docs.spring.io/spring-framework/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html. [Dostęp 15 12 2019].

[57] "Dokumentacja Apache Kafka," [Online]. Url: https://kafka.apache.org/documentation/. [Dostęp 15 12 2019].

[58] "Dokumentacja OpenAPI," [Online]. Url: https://swagger.io/docs/specification/about/. [Dostęp 15 12 2019].

[59] "Dokumentacja Gatling," [Online]. Url: https://gatling.io/docs/current/. [Dostęp 15 12 2019].

[60] "Dokumentacja Cucumber," [Online]. Url: https://cucumber.io/docs/cucumber/. [Dostęp 15 12 2019].

[61] "Dokumentacja Protractor," [Online]. Url: https://www.protractortest.org/#/toc. [Dostęp 15 12 2019].

[62] "Dokumentacja JHipster Domain Language," [Online]. Url: https://www.jhipster.tech/jdl/. [Dostęp 10 12 2019].

[63] "Custom Entity primary key - name and type," [Online]. Url: https://github.com/jhipster/generator-jhipster/issues/4224. [Dostęp 14 12 2019].

[64] "Dokumentacja DTO JHipster," [Online]. Url: https://www.jhipster.tech/using-dtos/. [Dostęp 13 12 2019].

[65] "Configuration and setup of the generator - jOOQ," [Online]. Url: https://www.jooq.org/doc/3.12/manual/code-generation/codegen-configuration/. [Dostęp 16 12 2019].

[66] "Differen use cases form jOOQ," [Online]. Url: https://www.jooq.org/doc/3.12/manual/getting-started/use-cases/. [Dostęp 18 12 2019].

[67] "jOOQ vs. Hibernate: When to Choose Which," [Online]. Url: https://blog.jooq.org/2015/03/24/jooq-vs-hibernate-when-to-choose-which/. [Dostęp 19 12 2019].

[68] "Dokumentacja Spring Resource Server," [Online]. Url: https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/html/boot-features-security-oauth2-resource-server.html. [Dostęp 21 12 2019].

[69] "Create React App," [Online]. Url: https://github.com/facebook/create-react-app. [Dostęp 21 12 2019].

[70] "React Main Concepts - Forms," [Online]. Url: https://reactjs.org/docs/forms.html. [Dostęp 21 12 2019].

[71] "Formik Overview," [Online]. Url: https://jaredpalmer.com/formik/docs/overview. [Dostęp 22 12 2019].

[72] "jOOQ | Performance benchmark," [Online]. Url: https://github.com/jOOQ/jOOQ/issues/1625. [Dostęp 11 1 2020].

[73] "Hibernate forum | Performance comparison with JDBC," [Online]. Url: https://forum.hibernate.org/viewtopic.php?f=1&t=931708. [Dostęp 11 1 2020].

[74] R. Martin, "Design Principles and Design Patterns," [Online]. Url: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. [Dostęp 11 1 2020].

10 Wykaz rysunków

11 Wykaz listingów

12 Wykaz tabel