

Spis treści

1	Wprowadzenie	3
1.1	Cel pracy	3
1.2	Istniejące rozwiązania	3
2	Komponenty wykorzystane w projekcie	4
2.1	Język Scala	4
2.2	Framework aplikacji internetowych Play!	4
2.3	Baza danych Redis	5
2.4	Inne biblioteki wykorzystane w projekcie	6
2.4.1	Scalaz	6
2.4.2	scalaxb	6
2.4.3	Redis Client	6
2.4.4	play-scalaz	7
2.4.5	play-navigator	7
3	Architektura	8
3.1	Standard Maven	8
3.2	API	10
3.3	Autoryzacja	11
3.3.1	Schemat działania standardu OAuth	11
3.3.2	Kontrola dostępu do projektów	13
3.4	Struktura projektu	13
3.5	Przechowywanie danych	15
3.5.1	Sposób przechowywania danych	16
3.5.2	Zapis plików w standardzie Maven	16
3.5.3	Przeglądanie i wyszukiwanie projektów przez interfejs webowy	21
3.5.4	Przeglądanie drzewa pakietów	21

3.5.5	Użytkownicy i kontrola dostępu	24
3.6	Plugin sbt	24
4	Wdrożenie	26
4.1	Konfiguracja środowiska	26
4.1.1	Konfiguracja serwera nginx	27
4.2	Konfiguracja aplikacji	29
4.2.1	Konfiguracja GitHub OAuth	29
4.2.2	Pobranie kodu aplikacji	30
4.2.3	Konfiguracja parametrów aplikacji	30
4.2.4	Uruchomienie aplikacji	30
5	Podsumowanie	32
5.1	Napotkane problemy	32
5.2	Kierunki i możliwości rozwoju aplikacji	32
6	Bibliografia	33

1 Wprowadzenie

1.1 Cel pracy

Celem niniejszego projektu inżynierskiego jest zaprojektowanie oraz implementacja aplikacji internetowej służącej do publikacji projektów OpenSource w języku Scala¹.

Ze względu na duży wpływ środowiska języka Java biblioteki języka Scala są dystrybuowane jako pakiety `.jar`. Publikacja biblioteki najczęściej sprowadza się do utrzymania własnego serwera, a korzystanie z takich biblioteki wymaga znania dokładnego położenia danej biblioteki. Dla porównania, w środowiskach innych języków taki jak Ruby² czy Clojure³ istnieje jedno centralne miejsce dla bibliotek OpenSource.

1.2 Istniejące rozwiązania

Nieduże projekty, często prowadzone przez jedną osobę, w dużej części nie są publikowane w formie gotowych pakietów, a jedynie w formie kodu źródłowego. Powoduje to duże niedogodności z wykorzystaniem takich projektów i aktualizacją do nowych wersji. Niektórzy autorzy bibliotek publikują pakiety na własnych serwerach co wymaga od nich utrzymywania serwera i robienie ręcznych aktualizacji. Najczęściej jest to zwykły serwer plików nieposiadający możliwości przeszukiwania projektów, w innych przypadkach skomplikowane w instalacji, konfiguracji i użyciu narzędzia takie jak Artifactory⁴. Utrzymanie własne serwera wiąże się z kosztami oraz wymaga pamiętania lokalizacji pakietów projektu przez jego użytkowników. Sytuacja taka ma związek z trudnościami związanymi z publikacją pakietów w globalnych serwisach takich jak Maven Central⁵ czy Sonatype⁶. Celem projektu ScalaJars jest uwolnienie autorów projektów od konieczności utrzymywania własnej infrastruktury oraz uproszczenie publikacji pakietów w serwisie specjalnie przygotowanym dla bibliotek w języku Scala.

¹Scala - <http://scala-lang.org/>

²RubyGems - <http://rubygems.org>

³Clojars - <http://clojars.org>

⁴Artifactory - http://www.jfrog.com/home/v_artifactory_opensource_overview#

⁵Maven Central - <http://search.maven.org>

⁶Sonatype - <https://repository.sonatype.org/index.html#welcome>

2 Komponenty wykorzystane w projekcie

2.1 Język Scala

Scala jest statycznie typowanym, hybrydowym językiem programowania łączącym paradygmaty programowania funkcyjnego oraz obiektowego stworzonym w celu sprostania wymaganiom pracy w środowisku rozproszonym. Główną platformą docelową języka jest Wirtualna Maszyna Javy (*ang. JVM - Java Virtual Machine*) ale trwają także prace nad kompilatorami pod platformę .NET oraz LLVM. Scala jest promowana jako “następca Javy” - język który poprawia błędy projektowe Javy, jest bezpieczniejszy (więcej błędów wykrytych podczas kompilacji), posiada abstrakcje zastępujące standardowe programowanie wielowątkowe (a co za tym idzie problemy związane z ręczną obsługą wątków). Scala posiada też wiele funkcjonalności znanych z języków funkcyjnych takich jak Scheme, ML czy Haskell - anonimowe funkcje, algebraiczne struktury danych, inferencja typów, typy wyższego rzędu (*ang. higher-order kinds*), dopasowanie do wzorca (*ang. pattern matching*), leniwa ewaluacja (*ang. lazy evaluation*). Ponadto Scala jest w pełni interoperacyjna z językiem Java co pozwala na łączenie kodu napisanego w obu tych językach w jednym projekcie.

Od paru lat Scala zdobywa coraz większą popularność zarówno w środowiskach akademickich/badawczych jak i w dużych korporacjach. Wiele firm używa Scali jako dodatek do istniejącego oprogramowania napisanego przy użyciu języka Java.

Serwisy internetowe używające Scali to m.in. Twitter⁷ (przejście z Ruby na Scale), Foursquare⁸ (od początku napisany w Scali), strona internetowa dziennika The Guardian⁹ (przejście z Javy na Scale), LinkedIn¹⁰ (nowe funkcjonalności napisane przy użyciu Scali), Meetup¹¹ (implementacja powiadomień w czasie rzeczywistym).

2.2 Framework aplikacji internetowych Play!

Aplikacja ScalaJars oparta jest na frameworku aplikacji internetowych Play!¹² napisanym w zdecydowanej większości przy użyciu języka Scala. W celu lepszej adaptacji projektu wśród programistów i firm posiada też API do języka Java.

Play! jest oparty o lekka, bezstanową architekturę zaprojektowaną tak, aby maksymalnie wykorzystać dostępne zasoby i pozwolić na skalowanie aplikacji w miarę wzrostu liczby użytkowników.

⁷Twitter - <http://twitter.com>

⁸Foursquare - <https://foursquare.com/>

⁹The Guardian - <http://guardian.co.uk/>

¹⁰LinkedIn - <http://linkedin.com/>

¹¹Meetup - <http://meetup.com/>

¹²Play! framework - <http://playframework.org/>

W odróżnieniu od standardowych bibliotek pozwalających na tworzenie aplikacji internetowych na platformie JVM, Play! implementuje architekturę MVC - Model, Widok, Kontroler. Głównymi założeniami tej architektury jest rozdzielenie logiki biznesowej, integracji z protokołem HTTP oraz warstwy prezentacji.

Play! jest udostępniony jako projekt OpenSource na licencji Apache 2.

2.3 Baza danych Redis

Ze względu na specyfikę aplikacji znacznie różniącą się od typowych aplikacji bazodanowych jako magazyn danych został wykorzystany Redis¹³.

Wspieranymi strukturami danych są:

- **String** - dowolny ciąg binarnych znaków, w odróżnieniu od najbardziej popularnej implementacji gdzie znak `\0` oznacza koniec ciągu Redis zapisuje osobno długość ciągu co pozwala na przechowywanie w bazie jakichkolwiek danych binarnych. Za pomocą komend bazy można m.in. odwołać się do dowolnego zakresu zachowanych danych. W przypadku gdy wartość ciągu reprezentuje liczbę możliwe są także operacje (de)inkrementacji czy operacje binarne.
- **List** - lista elementów typu **String**, baza pozwala m.in. na dodawanie/usuwanie elementów z obu stron listy oraz pobieranie określonego zakresu elementów.
- **Set** - zbiór unikalnych elementów typu **String**, baza udostępnia komendy pozwalające m.in. na dodawanie, usuwanie oraz pobieranie określonych elementów zbioru.
- **SortedSet** - podobnie jak w przypadku typu **Set** jest to zbiór unikalnych elementów typu **String** ale z dodatkową wartością numeryczną **score** określającą kolejność elementów w zbiorze. Dodatkowe operacje to m.in. pobranie elementów na podstawie parametru **score**.
- **Hash** - mapa o kluczach i wartościach typu **String** pozwalająca na wszystkie operacje na typie **String** dla konkretnego pola mapy

Ponieważ typ **String** może zawierać dowolną wartość często jest wykorzystywany do przechowywania zserializowanych obiektów. W przypadku ScalaJars jako format serializacji został wybrany JSON. Jest to obecnie najbardziej popularny format danych używany w internecie. Zachowuje on czytelność zapisanych danych i jest wspierany przez praktycznie każdy język programowania.

Redis jest udostępniony jako projekt OpenSource na licencji BSD.

¹³Redis - <http://redis.io>

2.4 Inne biblioteki wykorzystane w projekcie

2.4.1 Scalaz

Scalaz¹⁴ jest biblioteką napisaną w języku Scala rozszerzającą funkcyjne możliwości biblioteki standardowej bazującą częściowo na bibliotece standardowej języka Haskell. Główne funkcjonalności biblioteki Scalaz wykorzystywane w projekcie to:

- operatory poprawiające czytelność kodu
- typy `Either` oraz `Validation` znacząco ułatwiające obsługę błędów i eliminujące konieczność korzystania z wyjątków
- monady i operacje z nimi związane (m.in. “Kleisli arrow”)
- transformatory monad

Scalaz jest udostępnione jako projekt OpenSource na licencji BSD.

2.4.2 scalaxb

scalaxb¹⁵ jest rozszerzeniem dla sbt generującym strukturę klas na podstawie pliku XSD. Wykorzystany został do wygenerowania reprezentacji struktury pliku POM standardu Maven.

scalaxb jest udostępnione jako projekt OpenSource na licencji MIT.

2.4.3 Redis Client

Jako biblioteka do obsługi połączenia z bazą danych Redis został wykorzystany pakiet `net.debasishg.redisclient`¹⁶. Biblioteka ta implementuje wszystkie komendy bazy danych oraz pozwala na używanie natywnych typów języka Scala jako danych trzymanych w bazie. W celu ułatwienia pracy nad projektem w aplikacji ScalaJars zostało stworzone rozszerzenie tej biblioteki wspierające (de)serializację obiektów z/do formatu JSON przy pomocy klas typów.

Redis Client jest udostępniony jako projekt OpenSource.

¹⁴Scalaz - <http://code.google.com/p/scalaz/>

¹⁵scalaxb - <http://scalaxb.org/>

¹⁶Scala Redis Client - <https://github.com/debasishg/scala-redis>

2.4.4 play-scalaz

play-scalaz¹⁷ jest pomostem między frameworkiem Play! a biblioteką scalaz. Lista funkcjonalności zapewnionych przez bibliotekę:

- implementacje klas typów dla najczęściej używanych klas frameworka (m.in. `Promise`, `Future`)
- (de)serializacja obiektów za pomocą statycznie typowanych typeclass z obsługą błędów opartą o `Validation`

play-scalaz jest udostępnione jako projekt OpenSource na licencji MIT.

2.4.5 play-navigator

play-navigator¹⁸ jest alternatywną implementacją routera HTTP dla frameworka Play!. Oryginalny router posiada wiele ograniczeń, które uniemożliwiłyby powstanie aplikacji ScalaJars. Szczegółowe wyjaśnienie powodów powstania projektu znajduje się na blogu Codetunes¹⁹.

play-navigator jest udostępniony jako projekt OpenSource na licencji MIT.

¹⁷play-scalaz - <https://github.com/teamon/play-scalaz>

¹⁸play-navigator - <https://github.com/teamon/play-navigator>

¹⁹Codetunes - <http://codetunes.com/2012/05/09/scala-dsl-tutorial-writing-web-framework-router>

3 Architektura

3.1 Standard Maven

W celu zachowania kompatybilności z istniejącymi narzędziami aplikacja implementuje protokół używany przez najpopularniejszy obecnie w środowisku JVM system publikacji projektów Maven²⁰. Standard ten określa w jaki sposób projekty są publikowane oraz pobierane. Wprowadza on koncept repozytoriów; każde repozytorium składa się z adresu URL który jest używany jako adres bazowy dla wszystkich operacji.

Przykładowy adres URL repozytorium TypeSafe²¹:

```
http://repo.typesafe.com/typesafe/releases/
```

Publikacja odbywa się poprzez wysłanie zapytania HTTP PUT na określony adres URL którego ciało zawiera treść pliku. Adres URL składa się z pakietu, nazwy, wersji oraz nazwy pliku.

Schemat adresu:

```
PUT http://repozytorium/pakiet/nazwa/wersja/nazwa-artefaktu-wersja-rozszerzenie
```

Dla przykładu skompilowany moduł (`.jar`) `lucene-core` z projektu Apache Lucene²² w wersji 3.6.1 zostanie wysłany pod adres

```
PUT http://repozytorium/org/apache/lucene/lucene-core/3.6.1/lucene-core-3.6.1.jar
```

Standard Maven pozwala na zagnieżdżanie projektów w podprojekty. Każdy moduł definiuje artefakt. Definicja artefaktu jest dość szeroka, z reguły jest to plik `.jar` ze skompilowanym kodem. W przypadku projektów zawierających aplikacje webowe używa się rozszerzenia `.war`. Podczas publikacji przysyłane są też inne pliki takie jak:

- `.pom` - Plik XML zawierający informacje o artefakcie, m.in. jego zależności
- `-sources.jar` - Skompresowane źródła projektu (opcjonalne)
- `-javadoc.jar` - Skompresowana dokumentacja artefaktu (opcjonalne)

Przykładowo, pełna publikacja projektu Apache Lucene składającego się z modułów `lucene-core` oraz `lucene-queries` we wersji 3.6.1 wymagałaby następujących operacji:

²⁰Maven - <http://maven.apache.org/>

²¹Typesafe - <http://typesafe.com/>

²²Apache Lucene - <http://lucene.apache.org/core/>


```
PUT http://.../org/apache/lucene/lucene-core/3.6.1/lucene-core-3.6.1.pom
PUT http://.../org/apache/lucene/lucene-core/3.6.1/lucene-core-3.6.1.jar
PUT http://.../org/apache/lucene/lucene-core/3.6.1/lucene-core-3.6.1-sources.jar
PUT http://.../org/apache/lucene/lucene-core/3.6.1/lucene-core-3.6.1-javadoc.jar
PUT http://.../org/apache/lucene/lucene-queries/3.6.1/lucene-queries-3.6.1.pom
PUT http://.../org/apache/lucene/lucene-queries/3.6.1/lucene-queries-3.6.1.jar
PUT http://.../org/apache/lucene/lucene-queries/3.6.1/lucene-queries-3.6.1-sources.jar
PUT http://.../org/apache/lucene/lucene-queries/3.6.1/lucene-queries-3.6.1-javadoc.jar
```

Każda wersja kompilatora języka Java jest kompatybilna wstecz, nie ma problemów z używaniem projektu skompilowanego za pomocą kompilatora `javac 1.3` w projekcie który wykorzystuje kompilator `javac 1.4` lub nowszy. W przypadku projektów w języku Scala pojawiła się pewna trudność w wersjonowaniu artefaktów. Ze względu na bardzo dynamiczny rozwój języka nowe wersje kompilatora `scalac` są wydawane stosunkowo często i nie są ze sobą kompatybilne binarnie. Kompatybilność wsteczna jest zachowana jedynie w przypadku numeru wydania²³.

Oznacza to tyle, że projekt skompilowany kompilatorem `scalac` w wersji `2.7.x` nie może być użyty w innym projekcie używającym kompilatora w wersji `2.8.x`. Zależność ta działa w obie strony, projekt skompilowany pod `2.8.x` nie może być wykorzystany pod `2.7.x`.

W celu rozwiązania problemu niekompatybilności między wersjami kompilatora (nie tracąc przy tym ogólnej kompatybilności z systemem Maven) zastosowano prostą konwencję w nazewnictwie artefaktów.

```
nazwa-artefaktu-do-publicacji = nazwa-artefaktu_wersja-kompilatora
```

Dla przykładu biblioteka `scalaz` w wersji `7.0.0` skompilowana kompilatorem `scalac` w wersjach `2.8.0` oraz `2.9.1` zostanie opublikowana z następującymi artefaktami:

- `scalaz-core_2.8.0-7.0.0.jar`
- `scalaz-typelevel_2.8.0-7.0.0.jar`
- `scalaz-core_2.9.1-7.0.0.jar`
- `scalaz-typelevel_2.9.1-7.0.0.jar`

W celu ułatwienia publikowania projektów poprzez `sbt` wprowadzone zostało kompilowanie projektu dla wielu wersji języka Scala jednocześnie (`cross-compiling`).

²³Numeracja wersji oprogramowania - http://pl.wikipedia.org/wiki/Numeracja_wersji_oprogramowania

Standard Maven określa także sposób pobierania projektów jako zależności w innych projektach. Odbywa się to dwuetapowo. W celu pobrania artefaktu Maven przeszukuje dostępne repozytoria za pomocą zapytań HTTP HEAD.

Schemat adresu:

```
HEAD http://repozytorium/pakiet/nazwa/wersja/nazwa-artefaktu-wersja-rozszerzenie
```

Serwer powinien zwrócić ściśle określona odpowiedź:

- w przypadku gdy dany artefakt znajduje się w repozytorium
 - Status HTTP: 200 Found
 - Nagłówki HTTP:
 - * Content-Type²⁴ - typ pliku
 - * Content-Length - rozmiar pliku
- w przypadku gdy danego artefaktu nie ma w repozytorium
 - Status HTTP: 404 Not Found

Gdy serwer repozytorium odpowie statusem 200 Found Maven wysyła zapytanie HTTP GET na ten sam adres URL co przy zapytaniu HEAD. Serwer powinien wysłać wtedy dokładnie takie same nagłówki co w przypadku zapytania HEAD oraz treść pliku w ciele odpowiedzi.

W przypadku gdy w żadnym z dostępnych repozytoriów nie ma dostępnego danego artefaktu Maven zwróci błąd użytkownikowi.

Ponadto Maven najpierw pobiera plik .pom, aby sprawdzić zależności danego artefaktu i pobrać je w dokładnie taki sam sposób jak opisany powyżej.

3.2 API

Ponieważ publikowanie projektu składa się z wielu etapów, każdy plik jest przesyłany osobno, a w adresie URL jest informacja tylko o jednym artefakcie poprawne przypisanie afterkatu do projektu wymagałoby odczytywania odpowiedniego pliku POM przy każdym zapytaniu. Ponadto Maven nie udostępnia prostego sposobu autoryzacji użytkowników poprzez token. Obejściem tych ograniczeń okazało się wykorzystanie specjalnego adresu bazowego repozytorium. W przeciwieństwie do

²⁴MIME Content-Type - <http://en.wikipedia.org/wiki/MIME#Content-Type>

standardowych repozytoriów, ScalaJars wymaga podania specjalnego adresu URL zamierającego nazwę projektu oraz token użytkownika.

Schemat adresu bazowego repozytorium do zapisu:

```
http://scalajars.org/publish/nazwa-projektu/token/
```

Dzięki temu aplikacja nie musi sprawdzać plików POM przy każdym zapytaniu, wiadomo od razu do jakiego projektu należy artefakt. Ponadto można od razu wykryć jaki użytkownik dokonuje publikacji oraz czy ma dostęp do danego projektu.

W przypadku sprawdzania dostępności i odczytu projektów przez standard Maven adres repozytorium jest bliższy standardowi:

```
http://scalajars.org/repository/
```

3.3 Autoryzacja

W celu maksymalnego uproszczenia procesu publikacji projektów aplikacja nie posiada standardowego mechanizmu rejestracji oraz logowania użytkowników poprzez login/email i hasło. Użytkownicy niechętnie podają swój adres email w nowych serwisach internetowych ze względu na obawy przed niechcianą pocztą. Wiązałoby się to też z koniecznością wymyślenia i zapamiętania kolejnego hasła przez użytkownika.

Standardowe podejście wymagałoby zaimplementowania mechanizmu rejestracji, logowania, przypomnienia hasła oraz jego zmiany. Również przechowywanie hasel użytkowników jest nietrywialne ze względów bezpieczeństwa.

Zamiast normalnej rejestracji autoryzacja użytkowników w serwisie odbywa się przy pomocy mechanizmu OAuth²⁵ w wersji 2.0. Pozwala on na autoryzację użytkowników jak i dzielenie się prywatnymi informacjami przechowywanymi w jednym serwisie z innym. OAuth jest obecnie najpopularniejszym otwartym standardem autoryzacji używanym w internecie. Znaczna część serwisów z których każdy korzysta na co dzień umożliwia rejestrację za pomocą standardu OAuth.

3.3.1 Schemat działania standardu OAuth

W standardzie OAuth2 występują trzy główne elementy:

- Użytkownik

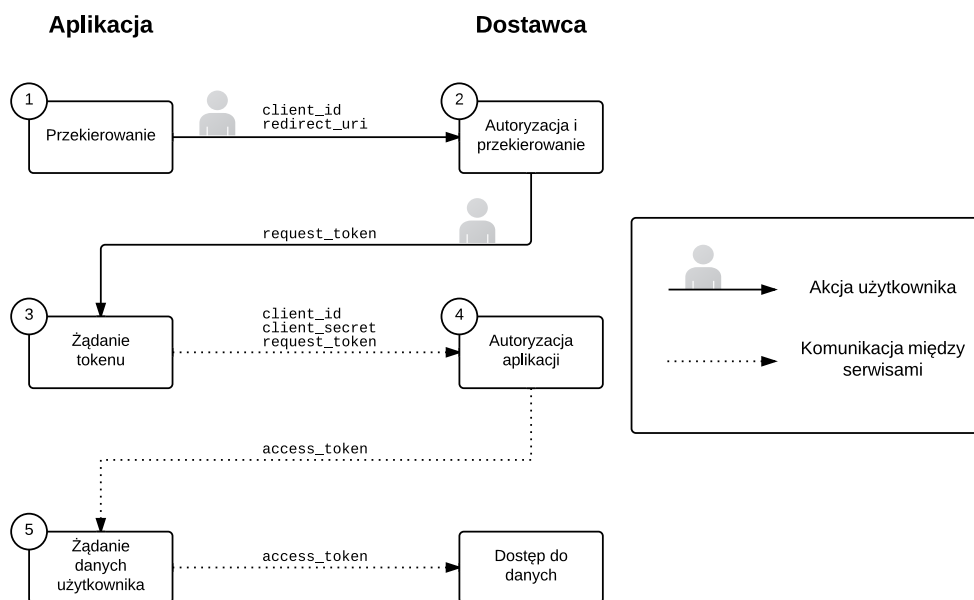
²⁵OAuth - <http://oauth.net>

- Aplikacja - serwis który chce uzyskać dostęp do informacji o użytkowniku
- Dostawca (*ang. provider*) - serwis w którym użytkownik posiada konto

Najpopularniejszymi dostawcami OAuth są m.in. Google²⁶, Facebook²⁷ oraz Twitter²⁸. Ze względu na charakter aplikacji jest ona skierowana głównie do programistów, dlatego też jako dostawca OAuth został wybrany serwis skupiający największą społeczność OpenSource - GitHub²⁹.

Aby aplikacja mogła uzyskać dostęp do danych użytkownika przechowywanych u konkretnego dostawcy należy ją zarejestrować u dostawcy. Konfiguracja aplikacji polega na pobraniu unikalnego identyfikatora (`client_id`) oraz tajnego klucza (`client_secret`).

Schemat działania znajduje się na rysunku 1.



Rysunek 1: Schemat działania standardu OAuth

1. Aplikacja przekierowuje użytkownika na stronę dostawcy. Parametry przekierowania to:

- `client_id` - unikalny identyfikator aplikacji

²⁶Google - <http://google.com>

²⁷Facebook - <http://facebook.com>

²⁸Twitter - <http://twitter.com>

²⁹GitHub - <http://github.com>

- `redirect_uri` - adres URL na który ma być przekierowany użytkownik po udanej autoryzacji
 - `scope` (opcjonalny) - określa zakres danych do których aplikacja chce uzyskać dostęp
2. Użytkownik loguje się na stronie dostawcy, dostawca generuje nowy token autoryzacji (`request_token`) i przekierowuje użytkownika na podany przez aplikację adres URL (`redirect_uri`) wraz z załączonym tokenem.
 3. Aplikacja wysyła żądanie do dostawcy o token dostępu (`access_token`). Wymagane parametry zapytania to:
 - `client_id` - ten sam co w punkcie 1
 - `client_secret` - tajny klucz aplikacji
 - `request_token` - token autoryzacji użytkownika
 4. Dostawca autoryzuje aplikację i w odpowiedzi odsyła token dostępu (`access_token`)
 5. Aplikacja wykorzystuje token dostępu (`access_token`) aby uzyskać informacje o użytkowniku.

3.3.2 Kontrola dostępu do projektów

Po pomyślnej autoryzacji każdemu użytkownikowi zostaje przypisany unikalny token pozwalający na publikację projektów. Token ten jest dostępny dla użytkownika po zalogowaniu. W razie potrzeby istnieje także możliwość ponownego wygenerowania tokena.

3.4 Struktura projektu

Ze względu na różnice w wersjonowaniu i nazewnictwie projektów obecnie żaden serwis umożliwiający publikację nie jest przystosowany do projektów w języku Scala. W większości tego typu rozwiązań brakuje nawet logicznego podziału na projekty i podprojekty.

Najczęściej spotykana struktura artefaktów:

```
* scalaz-core_2.8.0
  * 7.0.0
    * scalaz-core_2.8.0-7.0.0.jar
    * scalaz-core_2.8.0-7.0.0.pom
* scalaz-core_2.9.1
```

- * 7.0.0
 - * scalaz-core_2.9.1-7.0.0.jar
 - * scalaz-core_2.9.1-7.0.0.pom
- * scalaz-typelevel_2.8.0
 - * 7.0.0
 - * scalaz-typelevel_2.8.0-7.0.0.jar
 - * scalaz-typelevel_2.8.0-7.0.0.pom
- * scalaz-typelevel_2.9.1
 - * 7.0.0
 - * scalaz-typelevel_2.9.1-7.0.0.jar
 - * scalaz-typelevel_2.9.1-7.0.0.pom

Brakuje tutaj wspomnianego podziału na podprojekty, przeszukiwanie takiego repozytorium projektów pozostawia wiele do życzenia.

W niektórych serwisach można spotkać następującą strukturę:

- * scalaz
 - * scalaz-core_2.8.0
 - * 7.0.0
 - * scalaz-core_2.8.0-7.0.0.jar
 - * scalaz-core_2.8.0-7.0.0.pom
 - * scalaz-core_2.9.1
 - * 7.0.0
 - * scalaz-core_2.9.1-7.0.0.jar
 - * scalaz-core_2.9.1-7.0.0.pom
 - * scalaz-typelevel_2.8.0
 - * 7.0.0
 - * scalaz-typelevel_2.8.0-7.0.0.jar
 - * scalaz-typelevel_2.8.0-7.0.0.pom
 - * scalaz-typelevel_2.9.1
 - * 7.0.0
 - * scalaz-typelevel_2.9.1-7.0.0.jar
 - * scalaz-typelevel_2.9.1-7.0.0.pom

W tym przypadku artefakt o tej samej nazwie ale innej wersji kompilatora scala jest traktowany jak dwa osobne artefakty.

Poprawna struktura projektu:

```
* scalaz
  * 7.0.0
    * 2.8.0
      * scalaz-core
        * scalaz-core_2.8.0-7.0.0.jar
        * scalaz-core_2.8.0-7.0.0.pom
      * scalaz-typelevel
        * scalaz-typelevel_2.8.0-7.0.0.jar
        * scalaz-typelevel_2.8.0-7.0.0.pom
    * 2.9.1
      * scalaz-core
        * scalaz-core_2.9.1-7.0.0.jar
        * scalaz-core_2.9.1-7.0.0.pom
      * scalaz-typelevel
        * scalaz-typelevel_2.9.1-7.0.0.jar
        * scalaz-typelevel_2.9.1-7.0.0.pom
```

Powyższa struktura uwzględnia podział na podprojekty oraz wersje języka Scala. Umożliwia ona sprawne wyszukanie konkretnej wersji projektu dla wymaganej wersji kompilatora. Właśnie taka struktura została wykorzystana przy tworzeniu aplikacji ScalaJars.

3.5 Przechowywanie danych

Zadanie postawione przed aplikacją ScalaJars dalece odbiega od typowych zadań aplikacji bazodanowych.

Główne zadania stawiane przed bazą danych:

- Wylistowanie wszystkich projektów po nazwie
- Przeglądanie drzewa pakietów
- Wyszukiwanie projektów
- Zapis i pobranie skomplikowanej struktury projektu
- Pobranie listy ostatnio zaktualizowanych projektów
- Przechowanie danych dostępowych użytkownika
- Możliwość pracy w środowisku rozproszonym

Ze względu na złożoną strukturę projektów w standardzie Maven relacyjna baza danych zawierałaby znaczną ilość tabel. Baza relacyjna byłaby także znacznie wolniejszym rozwiązaniem, a także o wiele trudniejszym w skalowaniu na wiele serwerów. Zamiast bazy relacyjnej w aplikacji została wykorzystana baza danych Redis³⁰. Redis jest rozproszonym magazynem danych typu klucz-wartość. Ze względu na architekturę tej bazy pozwala ona na bardzo szybkie operacje zapisu oraz odczytu przez wiele klientów równocześnie bez potrzeby blokowania tabel w transakcjach jak ma to miejsce w przypadku baz relacyjnych.

Aplikacja pobiera informacje o projektach w dwóch wersjach: podstawowej - tylko nazwa projektu lub pełnej - pełny obiekt projektu wraz ze wszystkimi obiektami podrzędnymi. W przypadku bazy relacyjnej pobranie pełnej wersji projektu wymagałoby wielu skomplikowanych operacji.

3.5.1 Sposób przechowywania danych

W standardowych rozwiązaniach bazujących na bazach relacyjnych model danych jest wspólny dla całej aplikacji, wszystko jest oparte na relacjach między danymi w bazie. W wielu przypadkach prowadzi to do nieefektywnych operacji na bazie danych w celu uzyskania wyniku wymaganego przez funkcjonalność aplikacji. W podejściu KEY-VALUE to funkcjonalność aplikacji narzuca sposób przechowywania danych.

W aplikacji ScalaJars funkcjonalność, a więc także model danych w bazie został podzielony na kilka kluczowych elementów:

- Zapis plików w standardzie Maven
- Przeglądanie i wyszukiwanie projektów przez interfejs webowy
- Przeglądanie drzewa pakietów
- Użytkownicy i kontrola dostępu

3.5.2 Zapis plików w standardzie Maven

Dla danego artefaktu standard zakłada wysyłanie plików sekwencyjnie w określonej kolejności. Pierwszy zawsze zostanie wysłany plik POM, później `.jar` lub `.war`, na końcu `-sources.jar` oraz `-javadoc.jar`. Nie chroni to jednak aplikacji przed koniecznością przyjęcia kilku artefaktów jednocześnie. W przypadku publikowania projektów dla kilku wersji języka Scala może zajść sytuacja kiedy wiele artefaktów w różnych wersjach będzie publikowanych jednocześnie (w szczególności jeżeli publikacja następuje np. w rezultacie pomyślnego przejścia testów w systemie Continous Integration).

³⁰Redis - <http://redis.io>

Struktura projektu jest wielopoziomowym rozległym drzewem. Każde zapytanie publikacji artefaktu modyfikuje jedną konkretną ścieżkę w drzewie od korzenia do ostatniego poziomu. Schemat struktury znajduje się na Rysunku 2. Ze względu na konieczność równoległego przetwarzania modyfikacji wielu ścieżek w tym samym czasie nie można zapisać całego drzewa pod jednym kluczem w bazie danych. Spowodowało by to klasyczne zjawisko wyścigu - utratę danych, a w przypadku zastosowania mechanizmów blokady sekcji krytycznej zdecydowane obniżenie wydajności całej aplikacji. Synchronizacja dostępu do danych na poziomie procesu aplikacji uniemożliwiłaby wykorzystanie wielu serwerów, a tym samym ograniczyłaby na stałe możliwości skalowania całego projektu. W celu wyeliminowania konieczności ręcznej synchronizacji danych zostały wykorzystane atomowe operacje dostępne w bazie danych Redis. Do przechowywania informacji o projekcie aplikacja wykorzystuje wbudowane typy danych takie jak **String**, **List** oraz **Set**.

Lista najczęściej wykorzystywanych w aplikacji komend³¹:

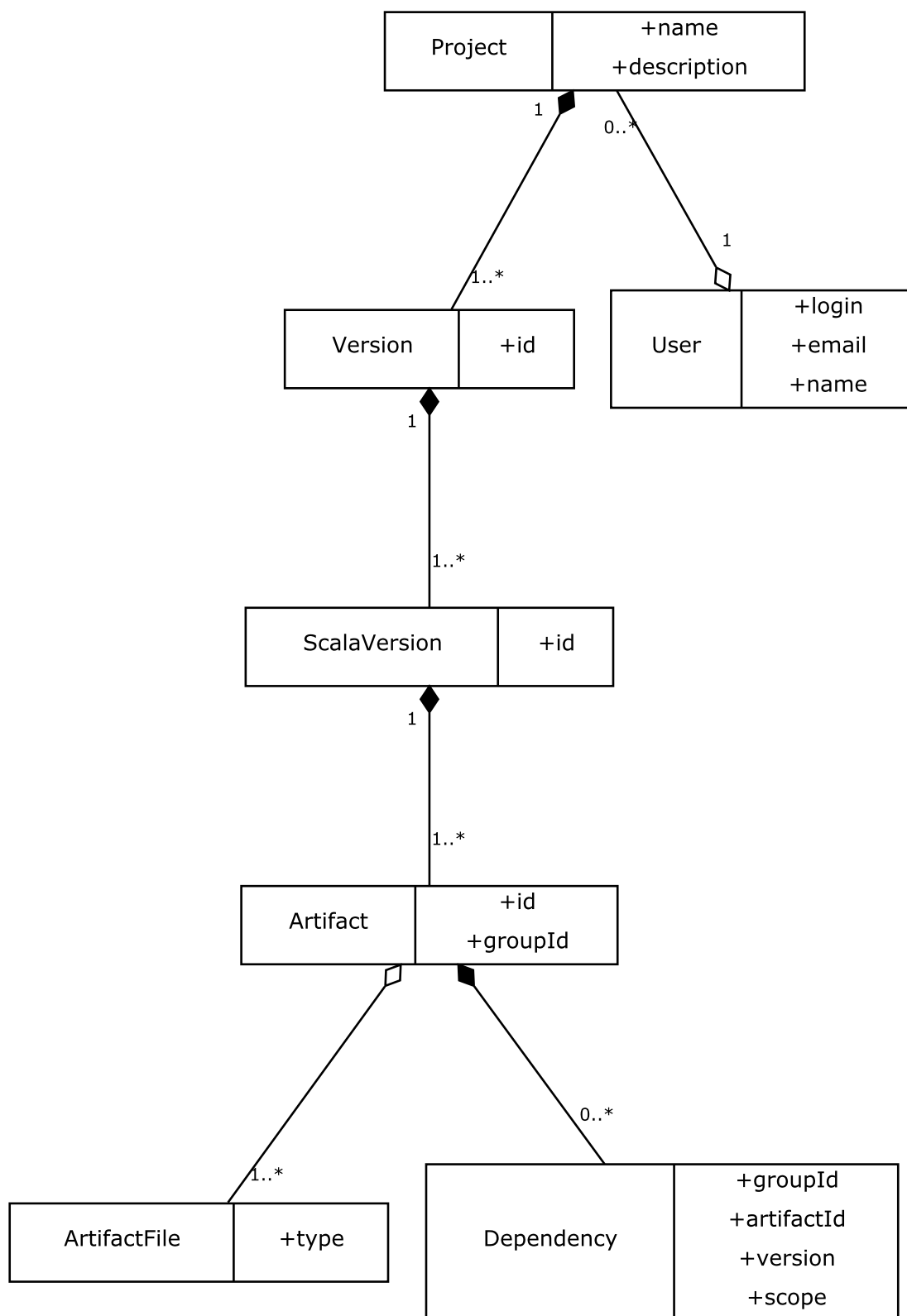
Komenda	Opis
SET key value	Ustawienie wartości value w polu o kluczu key
GET key	Pobranie wartości z pola o kluczu key
SADD key value	Dodanie wartości value do zbioru o kluczu key
SMEMBERS key	Pobranie wszystkich wartości zbioru o kluczu key
KEYS pattern	Pobranie listy kluczy pasujących do wzorca pattern
LPUSH key value	Dodanie wartości value na początek listy o kluczu key
LTRIM key offset len	Usunięcie len wartości zaczynając od indeksu offset z listy o kluczu key
LRANGE key offset len	Pobranie len wartości zaczynając od indeksu offset z listy o kluczu key

Przykład różnicy w działaniu atomowych operacji na specjalnych typach danych w porównaniu z prostym zapisem wartości dla danego klucza.

Zwykły zapis wartości:

(**[]** - oznacza zbiór pusty, **[1]** - zbiór zawierający element 1)

³¹Redis - Lista dostępnych komend - <http://redis.io/commands>



Zapytanie A	x	Zapytanie B	y	Wartość w polu klucz bazy danych
				[]
x = GET klucz	[]		[]	[]
	[]	y = GET klucz	[]	[]
x << 1	[1]		[]	[]
	[1]	y << 2	[2]	[]
SET klucz x	[1]		[2]	[1]
	[1]	SET klucz y	[2]	[2] <- błąd, utrata danych

Nastąpiła tutaj utrata danych poprzez nadpisanie wartości. Rozwiązaniem jest wykorzystanie operacji atomowych na zbiorach.

Zapytanie A	Zapytanie B	Wartość w polu klucz bazy danych
		[]
SADD klucz 1		[1]
	SADD klucz 2	[1,2]

W tym przypadku nie ma zjawiska wyścigu - nie następuje utrata danych.

Informacje o projektach są zapisywane w bazie pod specjalnymi kluczami.

Oznaczenia:

- **P** - nazwa projektu
- **V** - wersja
- **S** - wersja języka Scala
- **A** - nazwa artefaktu

Klucz	Typ	Opis
<code>projects</code>	Set	Zbiór nazw projektów
<code>projects:P</code>	String	Podstawowe dane projektu
<code>projects:P:versions</code>	Set	Zbiór wersji projektu
<code>projects:P:versions:V:scalaVersions</code>	Set	Zbiór wersji języka Scala
<code>projects:P:versions:V:scalaVersions:S:artifacts</code>	Set	Zbiór artefaktów
<code>projects:P:versions:V:scalaVersions:S:artifacts:A:dependencies</code>	Set	Zbiór zależności artefaktu
<code>projects:P:versions:V:scalaVersions:S:artifacts:A:files</code>	String	Dostępne pliki dla artefaktu

Taka struktura umożliwia bezproblemową modyfikację wielu ścieżek w drzewie projektu bez ryzyka utraty danych.

3.5.3 Przeglądanie i wyszukiwanie projektów przez interfejs webowy

Podstawowe wyszukiwanie projektów polega na wpisaniu pełnej lub części nazwy. Operacje wyszukiwania kluczy pasujących do wzorca można wykonać za pomocą komendy `KEYS pattern` gdzie `pattern` jest ciągiem znaków mogącym zawierać znaki specjalne takie jak `*` oznaczające zero lub więcej dowolnych znaków. Ze względu na strukturę kluczy wyszukiwanie ciągu `query` za pomocą komendy `KEYS projects:query*` zwróciłoby wiele rezultatów dla tego samego projektu.

Rozwiązaniem tego problemu jest stworzenie osobnego indeksu zawierającego tylko nazwy projektów. Każdy projekt zostaje dodatkowo zapisany pod kluczem `projects-index:P` z pustą wartością, gdzie `P` oznacza nazwę projektu. Pozwala to na wyszukiwanie ciągu `query` za pomocą komendy `KEYS projects-index:query*` i uzyskanie poprawnych wyników.

Pełne informacje dotyczące projektu są pobierane na podstawie jego nazwy P poprzez `GET projects:P`, a następnie wywołanie podobnych komend z odpowiednimi kluczami wgląd drzewa projektu.

3.5.4 Przeglądanie drzewa pakietów

Ważną funkcjonalnością obok przeglądania projektów po nazwie jest indeks artefaktów. Dla projektu scalaz indeks ma postać:

```
* org
  * scalaz
    * scalaz-core_2.8.0
      * 7.0.0
        * scalaz-core_2.8.0-7.0.0.pom
        * scalaz-core_2.8.0-7.0.0.jar
    * scalaz-core_2.9.1
      * 7.0.0
        * scalaz-core_2.9.1-7.0.0.pom
        * scalaz-core_2.9.1-7.0.0.jar
    * scalaz-typelevel_2.8.0
      * 7.0.0
        * scalaz-typelevel_2.8.0-7.0.0.pom
        * scalaz-typelevel_2.8.0-7.0.0.jar
```

- * scalaz-typelevel_2.9.1
 - * 7.0.0
 - * scalaz-typelevel_2.9.1-7.0.0.pom
 - * scalaz-typelevel_2.9.1-7.0.0.jar

Struktura ta odpowiada strukturze folderów i plików zapisanych na dysku. Najprostszym rozwiązaniem byłoby odwołanie się do systemu plików i wyświetlenie listy dostępnych artefaktów w danym folderze. Takie rozwiązanie jest jednak wysoce niewydatne, a także ogranicza możliwość zmiany miejsca i struktury przechowywania plików.

Zamiast odwoływać się do systemu plików w aplikacji został stworzony indeks w bazie danych. Ścieżką w systemie plików odpowiada klucz w indeksie, a wartość odpowiada zawartości danego katalogu. Jako wartość został ponownie wykorzystany typ danych **Set**.

Schemat budowy indeksu najlepiej zobrazuje przykład:

Klucz	Wartość
index:org	<pre> {"name": "scalaz", "_type": "package"} ----- {"name": "scalaz-core_2.8.0", "_type": "package"} {"name": "scalaz-core_2.9.1", "_type": "package"} {"name": "scalaz-typelevel_2.8.0", "_type": "package"} {"name": "scalaz-typelevel_2.9.1", "_type": "package"} ----- {"name": "7.0.0", "_type": "package"} ----- {"name": "scalaz-core_2.8.0:7.0.0", "_type": "file"} {"name": "scalaz-core_2.8.0-7.0.0.jar", "_type": "file"} {"name": "scalaz-core_2.8.0-7.0.0-javadoc.jar", "_type": "file"} {"name": "scalaz-core_2.8.0-7.0.0-sources.jar", "_type": "file"} ----- {"name": "7.0.0", "_type": "package"} ----- ... ----- {"name": "7.0.0", "_type": "package"} ----- index:org:scalaz:scalaz-core_2.8.0:7.0.0 ... ----- {"name": "7.0.0", "_type": "package"} ----- index:org:scalaz:scalaz-typelevel_2.8.0:7.0.0 ... ----- {"name": "7.0.0", "_type": "package"} ----- index:org:scalaz:scalaz-typelevel_2.8.0:7.0.0 ... ----- {"name": "7.0.0", "_type": "package"} ----- index:org:scalaz:scalaz-typelevel_2.8.0:7.0.0 ... ----- </pre>

3.5.5 Użytkownicy i kontrola dostępu

Ze względu na wykorzystanie serwisu GitHub jako dostawcy autoryzacji użytkowników po stronie aplikacji wystarczy jedynie zapisać podstawowe dane użytkownika oraz wygenerowany dla niego unikalny token dostępowy do publikacji.

Użytkownicy są zapisywani jako obiekt pod kluczem `users:L`, gdzie `L` oznacza login użytkownika w serwisie GitHub. Obiekt zawiera pola takie jak email czy imię i nazwisko.

Klucz	Wartość
<code>users:teamon</code>	<code>{"login": "teamon", "email": "i@teamon.eu", "name": "Tymon Tobolski"}</code>

Token użytkownika jest wykorzystywany w dwóch przypadkach: przy wyświetleniu tokena użytkownikowi oraz przy kontroli dostępu sprawdzaniu podczas publikacji. Sprowadza się to do prostych operacji: pobranie tokenu znając nazwę użytkownika oraz pobranie nazwy użytkownika znając token. W tym celu w bazie są zapisywane dla pola: pierwsze o kluczu `users-to-tokens:L` i wartości `T`, a drugie o kluczu `tokens-to-users:T` i wartości `L`, gdzie `L` oznacza nazwę użytkownika, a `T` token.

Klucz	Wartość
<code>users-to-tokens:teamon</code>	<code>5nqrd6slef10414jiosh4rton6</code>
<code>tokens-to-users:5nqrd6slef10414jiosh4rton6</code>	<code>teamon</code>

Pozwala to na szybkie pobranie potrzebnych wartości bez potrzeby wykonywania skomplikowanych operacji.

3.6 Plugin sbt

Kluczowym elementem procesu publikacji jest poprawne skonfigurowanie narzędzi używanych podczas pracy na projekcie. Najbardziej popularnym tego typu narzędziem w środowisku języka Scala jest sbt³². Konfiguracja polega na ustawieniu repozytorium do którego mają zostać wysłane skompilowane artefakty³³.

Przykład konfiguracji (plik `build.sbt`):

³²sbt - Simple Build Tool - <http://scala-sbt.org/>

³³sbt Publishing - <http://scala-sbt.org/release/docs/Detailed-Topics/Publishing.html>


```
publishTo := Some("Nazwa" at "http://adres/repozytorium")
```

Ze względu na architekturę ScalaJars wymaga podania nazwy projektu oraz tokenu użytkownika w adresie repozytorium. Wpisanie tokenu w pliku `build.sbt` naraża użytkownika na nieautoryzowany dostęp osób trzecich do jego projektów w systemie ScalaJars, w szczególności jeśli projekt jest udostępniany na licencji OpenSource i każdy może zobaczyć jego kod źródłowy w tym także pliki konfiguracyjne `sbt`. Ignorowanie lub usuwanie tokenu z pliku `build.sbt` przy każdej aktualizacji kodu w systemie kontroli wersji jest uciążliwe i podatne na omyłkowe upublicznienie tokenu. Rozwiązaniem tego problemu jest umieszczenie tokenu użytkownika poza katalogiem projektu. Poza względami bezpieczeństwa ułatwia to też ewentualną zmianę tokenu bez konieczności edycji konfiguracji każdego projektu osobno. Aby maksymalnie uprościć cały ten proces został stworzony plugin do `sbt` - `sbt-scalajars`. Plugin ten generuje konfigurację publikacji projektu na podstawie ustalonej nazwy projektu oraz pliku `.scalajars` w katalogu domowym użytkownika. Instalacja oraz konfiguracja pluginu sprowadza się do dwóch prostych kroków:

1. Dodanie pluginu do pliku `plugins.sbt`

```
addSbtPlugin("org.scalajars" %% "sbt-scalajars" % "0.1.0")
```

2. Ustawienie nazwy projektu w pliku `build.sbt`

```
seq(scalajarsSettings:_*)  
  
scalajarsProjectName := "nazwa-projektu"
```

Podczas ewaluacji ustawień plugin nadpisuje parametr `publishTo` wartością:

```
publishTo := Some("ScalaJars" at "http://scalajars.org/publish/projekt/token")
```

gdzie `projekt` jest nazwą projektu pobieraną z parametru `scalajarsProjectName` a `token` z pliku `.scalajars`.

4 Wdrożenie

Aplikacja działa na Maszynie Wirtualnej Javy, więc jest teoretyczna możliwość uruchomienia jej na każdym systemie. Jednak autorzy bazy danych Redis oficjalnie wspierają tylko środowiska kompatybilne ze standardem POSIX³⁴. Z tego względu opisany sposób instalacji będzie oparty na tym właśnie standardzie.

Standardową metodą uruchamiania aplikacji opartych o framework Play! jest użycie tzw. reverse proxy - użycie serwera HTTP (np. nginx³⁵, Apache³⁶) który działa na porcie 80 i przekazuje połączenia do serwera aplikacji który działa na porcie z zakresu 1024-65535. Istnieje również możliwość uruchomienia aplikacji bezpośrednio na porcie 80 jednak wymaga to uruchomienia procesu aplikacji z prawami roota co w razie błędów w aplikacji może narazić serwer na uszkodzenie.

Opisany sposób instalacji dotyczy systemu Ubuntu³⁷ 12.04 LTS x86_64 oraz serwera HTTP nginx.

4.1 Konfiguracja środowiska

Wymagania systemowe:

- POSIX
- JRE + JDK (wersja 6 lub wyższa)
- nginx
- redis
- git³⁸
- sbt

W przypadku systemu Ubuntu powyższe aplikacje można zainstalować za pomocą menadżera pakietów `apt-get`.

³⁴POSIX - <http://pl.wikipedia.org/wiki/POSIX>

³⁵nginx - <http://nginx.org/>

³⁶Apache - <http://httpd.apache.org/>

³⁷Ubuntu - <http://ubuntu.com/>

³⁸Git - <http://git-scm.com/>

4.1.1 Konfiguracja serwera nginx

Plik `/etc/nginx/nginx.conf`

```
user www-data;
worker_processes 4;
pid /var/run/nginx.pid;

events {
    worker_connections 768;
}

http {
    server_names_hash_bucket_size 64;

    # Basic Settings
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;

    client_max_body_size 256M;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    server {
        listen 80;

        server_name *.scalajars.org scalajars.org;

        root /home/scalajars/public;
        index index.html index.htm;
```

```

location / {
    proxy_set_header    X-Real-IP    $remote_addr;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header    Host $http_host;
    proxy_redirect      off;

    if (-f $request_filename/index.html) {
        rewrite (.*) $1/index.html break;
    }

    if (-f $request_filename.html) {
        rewrite (.*) $1.html break;
    }

    if (!-f $request_filename) {
        proxy_pass http://localhost:60001;
        break;
    }

}

error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root    html;
}
}
}

```

Ważne elementy powyższej konfiguracji:

- `client_max_body_size 256M;`³⁹ - zwiększenie standardowego limitu na rozmiar pliku przesyłanego na serwer
- `server_name *.scalajars.org scalajars.org;`⁴⁰ - definicja domen które mają być przekierowane do aplikacji

³⁹nginx `client_max_body_size` = http://wiki.nginx.org/HttpCoreModule#client_max_body_size

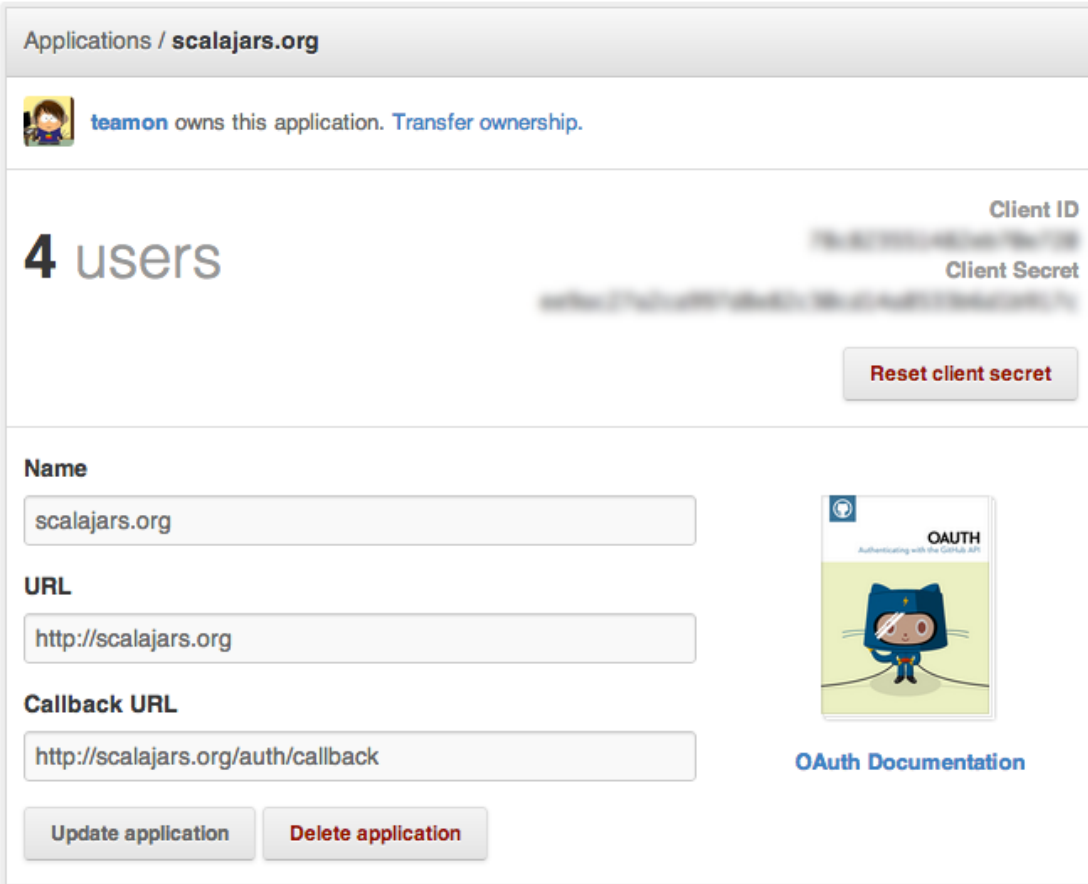
⁴⁰nginx-servername

- `proxy_pass http://localhost:60001;`⁴¹ - definicja portu na którym działa aplikacja

4.2 Konfiguracja aplikacji

4.2.1 Konfiguracja GitHub OAuth

Aby umożliwić użytkownikom logowanie do aplikacji ScalaJars kontem z serwisu GitHub należy w ustawieniach własnego konta w serwisie utworzyć nową aplikację OAuth w zakładce **Account Settings/Applications**. Przykład konfiguracji znajduje się na Rysunku 3. Po utworzeniu aplikacji OAuth należy skopiować wygenerowane **Client ID** oraz **Client Secret**, które będą potrzebne w kolejnym kroku konfiguracji.



The screenshot shows the GitHub 'Applications' page for the user 'teamon'. The application is named 'scalajars.org' and has 4 users. The 'Name' field contains 'scalajars.org', the 'URL' field contains 'http://scalajars.org', and the 'Callback URL' field contains 'http://scalajars.org/auth/callback'. The 'Client ID' and 'Client Secret' fields are visible but blurred. A 'Reset client secret' button is present. To the right, there is a 'OAuth' logo with a GitHub Octocat and a link to 'OAuth Documentation'. At the bottom, there are 'Update application' and 'Delete application' buttons.

Rysunek 3: Konfiguracja aplikacji OAuth w serwisie GitHub

⁴¹nginx `proxy_pass` - http://wiki.nginx.org/HttpProxyModule#proxy_pass

4.2.2 Pobranie kodu aplikacji

Kod źródłowy aplikacji ScalaJars jest przechowywany za pomocą systemu kontroli wersji Git, opublikowany w serwisie GitHub w repozytorium publicznym `teamon/scalajars.org`⁴². Pobranie aplikacji sprowadza się do wykonania komendy:

```
$ git clone git@github.com:teamon/scalajars.org.git
```

Serwis GitHub umożliwia także pobranie archiwum z ostatnią wersją projektu z adresu `https://github.com/teamon/scalajars.org/archive/master.tar.gz`.

4.2.3 Konfiguracja parametrów aplikacji

Konfiguracja aplikacji znajduje się w pliku `conf/application.conf`. Jest to plik w formacie HOCON⁴³. W repozytorium plik ten jest przygotowany tak, aby pobierał parametry ze zmiennych środowiskowych, jednak można wpisać ustawienia bezpośrednio do pliku.

Ważne parametry aplikacji:

Klucz	Opis
<code>upload.dir</code>	Katalog, w którym przechowywane będą opublikowane artefakty
<code>oauth.github.clientId</code>	Client ID pobrane z serwisu GitHub
<code>oauth.github.clientSecret</code>	Client Secret pobrane z serwisu GitHub
<code>redis.host</code>	Host bazy danych Redis
<code>redis.port</code>	Port bazy danych Redis

4.2.4 Uruchomienie aplikacji

Po wykonaniu powyższych kroków należy skompilować kod aplikacji do wykonywalnego archiwum JAR za pomocą komendy

```
$ sbt stage
```

Utworzy to katalog `target/staged` zawierający skompilowaną aplikację oraz wszystkie zależności, a także plik `target/start` służący do uruchomienia aplikacji.

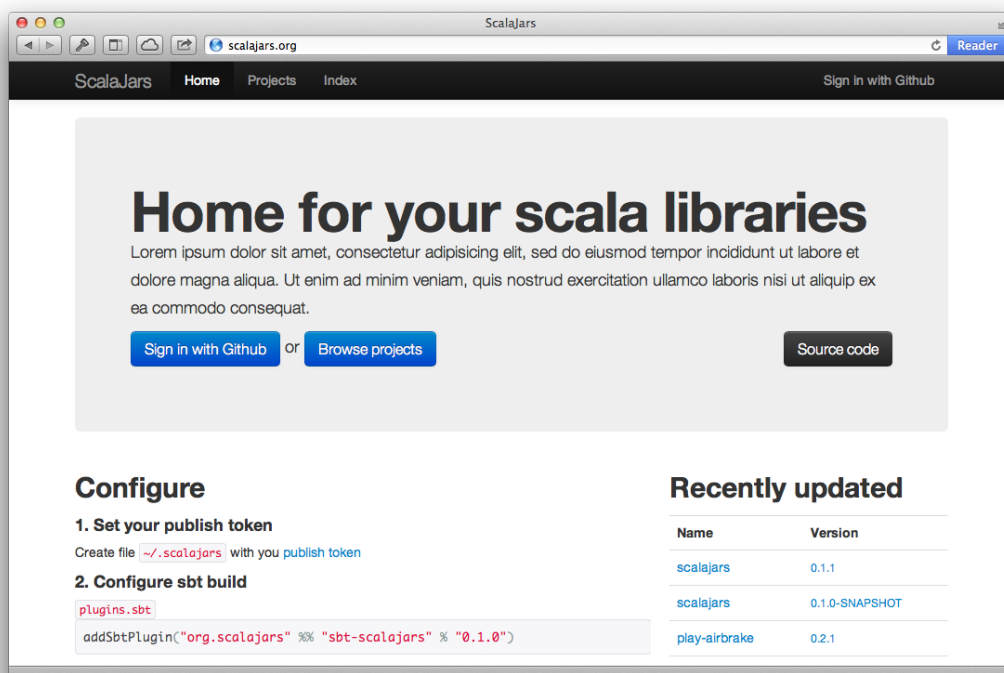
⁴²Repozitorium `teamon/scalajars.org` - <https://github.com/teamon/scalajars.org>

⁴³HOCON - <https://github.com/typesafehub/config>

Ostatnim parametrem który należy podać jest port na którym będzie nasłuchiwał serwer aplikacji - w tym przypadku jest to port 60001.

```
$ target/start -Dhttp.port=60001
```

Po wykonaniu powyższej komendy aplikacja zostanie uruchomiona i będzie dostępna pod adresem `http://scalajars.org`.



Rysunek 4: Strona główna aplikacji ScalaJars

5 Podsumowanie

5.1 Napotkane problemy

Największym wyzwaniem podczas budowy aplikacji ScalaJars była architektura bazy danych. Ze względu wykorzystanie bazy danych Redis nie można było zastosować powszechnego modelowania tak jak w przypadku baz relacyjnych.

5.2 Kierunki i możliwości rozwoju aplikacji

Głównym ograniczeniem obecnej wersji aplikacji jest przechowywanie plików na dysku lokalnym. W przypadku konieczności skalowania na wiele serwerów należałoby skorzystać z rozproszonej struktury przechowywania plików takiej jak np. Amazon S3⁴⁴. Pozwoliłoby to na użycie dowolnej ilości serwerów aplikacyjnych.

Istniejące funkcjonalności katalogu projektów można by poszerzyć o bardziej rozbudowaną strukturę, np. dodanie słów kluczowych do projektów co pozwoliłoby na bardziej sprawne wyszukiwanie projektów o danym przeznaczeniu. Ponadto dla samego projektu warto by było dodać możliwość podania odnośników do kodu źródłowego czy strony internetowej autora. Interesująca funkcjonalnością byłaby także możliwość przeglądania dokumentacji projektu bez konieczności pobierania plików JavaDoc.

Kolejnym etapem rozwoju aplikacji mogłoby być dodanie obsługi kolaborantów - umożliwienie kilku użytkownikom publikowanie danego projektu.

Ważną funkcjonalnością dla firm tworzących oprogramowanie w języku Scala byłaby możliwość publikowania projektów prywatnych, do których dostęp mieliby tylko uprawnieni użytkownicy.

Dużo trudniejszą i wymagającą więcej zasobów funkcjonalnością byłby system automatycznej kompilacji i publikowania artefaktów w momencie opublikowania nowej wersji kodu źródłowego w serwisie GitHub. Pozwoliłoby to użytkownikom danego projektu bezproblemowy dostęp do najnowszych wersji oraz zwalniałoby autora projektu z konieczności ręcznej publikacji przy wydawaniu nowych wersji projektu.

⁴⁴Amazon S3 - <http://aws.amazon.com/s3/>

6 Bibliografia

1. “Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition”, Martin Odersky, Lex Spoon, Bill Venners, 2008, ISBN: 9780981531649
2. “Functional Programming in Scala”, Pual Chiusano, Rúnar Bjarnason, 2012, ISBN: 9781617290657
3. “Play for Scala”, Peter Hilton, Erik Bakker, Francisco Canedo, 2012, ISBN: 9781617290794
4. “Maven: The Complete Reference”, <http://sonatype.com/books/mvnref-book/reference/>
5. “Maven: The Definitive Guide”, Sonatype Company, ISBN: 9780596517335
6. “Getting Started with OAuth 2.0”, Ryan Boyd, 2012, ISBN: 978-1449311605
7. “Version Control with Git: Powerful tools and techniques for collaborative software development”, Jon Loeliger, Matthew McCullough, 2009, ISBN: 9781449316389