

POLITECHNIKA LUBELSKA

Wydział Elektrotechniki i Informatyki

Kierunek Informatyka



PRACA MAGISTERSKA

**Programowanie i wdrażanie aplikacji sieciowych
w języku Python**

Dyplomant:
Marcin Rataj
Numer albumu:
56898D

Promotor:
dr Beata Pańczyk

Lublin, 2010

Niniejsza praca została opublikowana 23 czerwca 2010 na licencji:

Creative Commons

Uznanie autorstwa-Użycie niekomercyjne-Na tych samych warunkach

3.0 Polska

Wolno:

- kopiować, rozpowszechniać, odtwarzać i wykonywać utwór
- tworzyć utwory zależne

Na następujących warunkach:

- *Uznanie autorstwa*

Utwór należy oznaczyć w sposób określony przez Twórcę lub Licencjodawcę.

- *Użycie niekomercyjne*

Nie wolno używać tego utworu do celów komercyjnych.

- *Na tych samych warunkach*

Jeśli zmienia się lub przekształca niniejszy utwór, lub tworzy inny na jego podstawie, można rozpowszechniać powstały w ten sposób nowy utwór tylko na podstawie takiej samej licencji.

W celu ponownego użycia utworu lub rozpowszechniania utworu należy wyjaśnić innym warunki licencji, na której udostępnia się utwór. Każdy z tych warunków może zostać uchylony, jeśli uzyska się pisemne zezwolenie właściciela praw autorskich.

Treść licencji: <http://creativecommons.org/licenses/by-nc-sa/3.0/pl/>

Spis treści

1. Wstęp	5
2. Aplikacje sieciowe	6
2.1. Geneza aplikacji internetowych	6
2.2. Przegląd oraz rozwój popularnych technologii	7
2.2.1. Serwery HTTP	7
2.2.2. Języki i frameworki	8
2.2.3. Bazy danych: RDBMS	9
2.2.4. Bazy danych: NoSQL	11
2.3. Wykorzystane narzędzia	12
2.3.1. Język programowania Python	12
2.3.2. Framework Django	14
2.3.3. Baza CouchDB	15
2.3.4. Serwer Nginx	16
2.4. Środowisko programistyczne	16
3. Wykorzystane elementy Django oraz CouchDB	18
3.1. Metodyka Django	18
3.1.1. Podstawowe pojęcia	18
3.1.2. Budowa projektu Django	19
3.1.3. Nowa aplikacja	21
3.2. Model danych oparty o CouchDB	21
3.2.1. Dokumenty	21
3.2.2. Widoki	23
3.2.3. HTTP API	24
3.2.4. Zalety	25
3.2.5. Integracja z Django: <i>Couchdbkit</i>	25
3.2.6. Modele dokumentów	27
3.2.7. Widoki CouchDB w Django	29
3.3. System szablonów	34
3.3.1. Wstęp	34
3.3.2. Zmienne	34
3.3.3. Filtry	35
3.3.4. Tagi	36
3.3.5. Dziedziczenie szablonów	38
3.4. Obsługa zapytań HTTP	39
3.4.1. Projektowanie URL	39
3.4.2. Implementacja schematu URL	40
3.5. Formularze	42
3.5.1. Przykładowa operacja POST	43
4. Wybrane API oraz biblioteki	45
4.1. Wstęp	45
4.1.1. Przyczyny powstania API	45
4.1.2. API aplikacji internetowej	46
4.1.3. Biblioteki API	47
4.2. Projekt MusicBrainz	48
4.2.1. O projekcie	48
4.2.2. API XML Web Service	48
4.2.3. Biblioteka <i>python-musicbrainz2</i>	51
4.3. Last.fm	52
4.3.1. O serwisie	52

4.3.2.	Biblioteka <i>pylast</i>	53
4.4.	Flickr	55
4.4.1.	O serwisie	55
4.4.2.	API oraz biblioteka <i>flickrapi</i>	55
4.5.	YouTube	56
4.5.1.	O serwisie	56
4.5.2.	Gdata API	57
4.6.	DBpedia	58
4.6.1.	O projekcie	58
4.6.2.	Biblioteka <i>surf-sparql</i>	58
4.7.	reCAPTCHA	59
4.7.1.	O projekcie	59
4.7.2.	Biblioteka <i>recaptcha-client</i>	60
4.8.	BBC Music	61
4.8.1.	O serwisie	61
4.8.2.	Obsługa API bez dedykowanej biblioteki	61
4.9.	RSS oraz Atom	62
4.9.1.	Biblioteka <i>feedparser</i>	64
5.	Przykładowa aplikacja: MMDA	67
5.1.	Założenia projektu	67
5.2.	Status prawny przetwarzanych informacji	69
5.3.	Algorytmy gromadzenia i przetwarzania informacji	70
5.3.1.	Architektura cache	70
5.3.2.	Artysta	72
5.3.3.	Publikacja	72
5.3.4.	Multimedia	74
5.3.5.	Kanał wiadomości	75
5.3.6.	Wyszukiwanie	76
5.3.7.	Możliwości rozwoju	77
5.4.	Obsługa przez użytkownika	78
6.	Wdrożenie aplikacji	81
6.1.	Środowisko produkcyjne	81
6.2.	Konfiguracja serwera	83
6.3.	Bezpieczeństwo	84
6.4.	Optymalizacja wydajności	85
6.4.1.	Gunicorn	85
6.4.2.	Memcached	87
6.4.3.	Testy	90
7.	Podsumowanie	92
Literatura	93
	Strony internetowe	93
A. Załączniki	96
A.1.	Zrzuty ekranowe aplikacji	96
A.2.	Logi wykonanych testów	105
Spis rysunków	109
Spis listingów	110
Spis tabel	112

Rodzicom

1. Wstęp

Branża informatyczna przechodzi obecnie istotne zmiany dotyczące sposobu, w jaki pisane oraz rozwijane są aplikacje. Miejsce natywnych, uruchamianych przez system operacyjny narzędzi coraz częściej zajmują obecne w Internecie *aplikacje sieciowe*, czyli dynamicznie generowane strony WWW pełniące pożądaną funkcję.

Celem niniejszej pracy magisterskiej jest przedstawienie języka Python oraz wybranych, pokrewnych technologii jako wydajnego, darmowego i otwartego środowiska do tworzenia aplikacji sieciowych.

Wybór omawianych przez autora technologii wynika z doświadczenia zdobytego podczas wieloletniej pracy z aplikacjami internetowymi. Środowisko jest połączeniem uznanych, sprawdzonych narzędzi i standardów obecnych w sieci od początku jej istnienia oraz nowatorskich rozwiązań, jakie pojawiły się w ciągu ostatniej dekady.

Praca została podzielona na siedem rozdziałów – pierwszy stanowi niniejszy wstęp. Rozdział drugi przedstawia rys historyczny oraz przegląd popularnych rozwiązań. Przybliża ich rozwój i wyjaśnia specyfikę aplikacji internetowych. Opisana została również konfiguracja środowiska programistycznego.

Rozdział trzeci to opis wykorzystanych do tworzenia aplikacji elementów frameworka *Django* oraz bazy *CouchDB*. Zagadnienia takie jak: praca z bazą danych zorientowaną na dokumenty, projektowanie aplikacji w oparciu o zasoby URI, separacja warstw prezentacji, logiki oraz danych – wymagają przyswojenia pewnych podstaw teoretycznych. Stanowią one istotną część rozdziału.

Jedną z zalet języka Python jest olbrzymi wybór otwartych bibliotek. Rozdział piąty opisuje wykorzystanie wybranych w przykładowej aplikacji. Dodatkowo wymienione zostały problemy wraz z płynącymi ze specyfiki języka technikami programistycznymi wykorzystanymi do ich rozwiązania.

Najlepszym sposobem demonstracji omawianych zagadnień było tworzenie przykładowej aplikacji. Rozdział szósty omawia zasadę działania *MMDA* – prostego agregatora treści muzycznych. Obejmuje zarówno logikę biznesową odpowiedzialną za gromadzenie i przetwarzanie danych, jak i obsługę serwisu przez użytkownika. Zarówno aplikacja, jak i wykorzystane biblioteki napisane są w języku Python.

Kolejny rozdział omawia zagadnienia związane z *wdrożeniem* aplikacji: specyfikę środowiska produkcyjnego, bezpieczeństwo oraz dostępne techniki optymalizacji już napisanych aplikacji. Wybrane rozwiązania przetestowano na przykładzie *MMDA*.

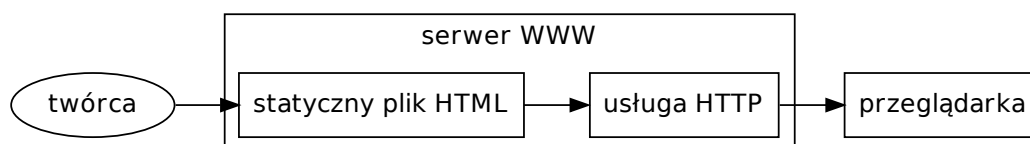
Ostatni rozdział to podsumowanie oraz wnioski płynące z pracy.

2. Aplikacje sieciowe

2.1. Geneza aplikacji internetowych

Sposób generowania i dostarczania treści w Internecie uległ istotnej transformacji na przestrzeni ostatnich kilkunastu lat. Głównym nośnikiem informacji był (i jest nadal) protokół *HTTP*¹, służący między innymi do przesyłania dokumentów w formacie *(X)HTML*². Tego typu dokument, po zinterpretowaniu przez *przeglądarkę*, w dalszej części pracy nazywany jest *stroną internetową*.

Początkowo każda strona internetowa była zwyczajnym plikiem tekstowym zawierającym składnię HTML. Plik tworzone ręcznie za pomocą edytora tekstowego. Następnie wysyłano na zdalny serwer, który zajmował się jego dystrybucją poprzez protokół HTTP. Przeglądarka otrzymywała dokładną kopię obecnego na serwerze dokumentu. Rola serwera WWW ograniczała się więc jedynie do roli pośrednika i dystrybutora statycznej treści (*Rys. 2.1*).



Rysunek 2.1. Cykl życia statycznej strony internetowej
Źródło: opracowanie własne

Technika ta pociągała za sobą liczne konsekwencje. Wprowadzanie zmian wiązało się z aktualizacją obecnych na serwerze plików. Gdy te same dane były obecne w kilku miejscach, konieczne było wykonywanie żmudnej i redundantnej edycji. Przy pewnym stopniu złożoności serwisu internetowego statyczne strony HTML przestały być wydajnym sposobem publikacji treści.

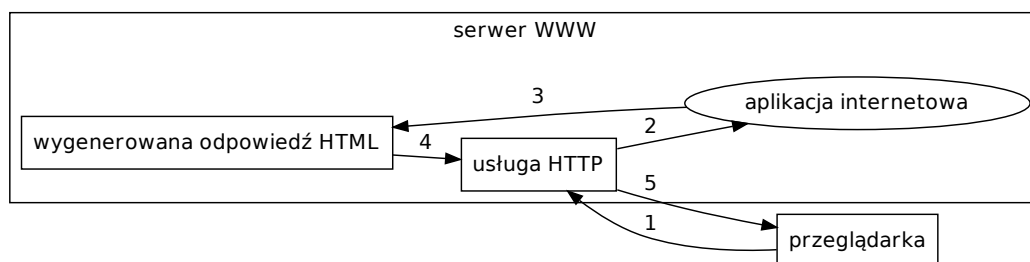
Używany już w 1993 roku, standard *CGI*³ jest jedną z pierwszych odpowiedzi na wyżej wymienione problemy [3]. Serwer HTTP za pośrednictwem CGI komunikuje się z aplikacją zainstalowaną na serwerze. Aplikacja ta może być napisana w dowolnym języku. Istotne jest, aby implementowała metody komunikacji narzucone przez CGI i generowała poprawną odpowiedź, na przykład w formacie HTML. Cykl pracy serwera wykorzystującego CGI ilustruje *rys. 2.2*.

Stronę internetową, której kod źródłowy generowany jest dynamicznie, nazywamy *aplikacją internetową*. W zależności od kontekstu, w dalszej części pracy aplikacją in-

¹ ang. *Hypertext Transfer Protocol*

² ang. *(Extensible) HyperText Markup Language*

³ ang. *Common Gateway Interface*



Rysunek 2.2. Cykl życia aplikacji internetowej
Źródło: opracowanie własne

ternetową nazywana jest również aplikacja generująca odpowiedź HTTP, zorientowana na interakcję z użytkownikiem przeglądarki internetowej [4].

2.2. Przegląd oraz rozwój popularnych technologii

2.2.1. Serwery HTTP

Wszystkie aplikacje internetowe we współczesnym Internecie opierają się na wprowadzonym przez CGI paradygmacie dynamicznie generowanych odpowiedzi. Różnice pojawiają się w łańcuchu przetwarzania żądań HTTP. Można wyróżnić następujące odmiany aplikacji sieciowej:

- aplikacja wymagająca zewnętrznego serwera HTTP
(komunikacja za pomocą CGI lub analogicznej technologii),
- aplikacja z wbudowaną funkcją serwera HTTP
(autonomiczna, kompletna – przykład: *Zope*).

Rozszerzona, otwarta implementacja standardu CGI nazwana *FastCGI* jest często wdrażana jako domyślna metoda komunikacji serwera HTTP z aplikacją internetową. Popularnym rozwiązaniem są również serwery implementujące metody komunikacji dedykowane dla danego środowiska programistycznego. Przykład: Java i środowiska uruchomieniowe *Tomcat*, *Glassfish*.

Istnieją rozwiązania, które łączą w sobie funkcjonalność serwera HTTP i aplikacji sieciowej. Niektóre serwery HTTP (na przykład Apache) posiadają moduły umożliwiające bezpośrednią kompilację lub wywołanie interpretera kodu aplikacji napisanych w wybranych językach. Najbardziej popularnym przykładem tego rozwiązania jest język PHP¹, który przez wiele lat – wraz z serwerem Apache – stanowił domyślne środowisko tworzenia aplikacji. Niestety z biegiem czasu Apache stał się ociężałym, pamięciożernym oprogramowaniem. Obecnie coraz częściej spotyka się tendencję migracji z Apache na lżejsze i bardziej wydajne serwery, takie jak *Lighttpd* [5], czy wykorzystany w niniejszej pracy *Nginx* [6].

¹ Skryptowy język programowania zaprojektowany do generowania stron internetowych.

2.2.2. Języki i frameworki

Przez długi czas królował standard CGI, który wciąż bywa bezpośrednio kojarzony ze środowiskiem aplikacji internetowych pisanych w Perlu¹. Z uwagi na rozbudowane możliwości manipulacji ciągami znaków oraz bogate repozytorium bibliotek i narzędzi *CPAN*² język ten był naturalnym wyborem wielu programistów. Należy zaznaczyć, iż w owym okresie PHP było wciąż prostym językiem skryptowym, który nie mógł się równać z wielo-paradygmatowym Perlem. Niestety, złożoność języka, odstraszająca krzywa uczenia się oraz rozwój innych języków i ich bibliotek sprawiły, iż Perl przestał być oczywistym wyborem.

Jedną z alternatyw stał się wciąż rozwijany język PHP. Wersja 5 wprowadziła konstrukty umożliwiające programowanie obiektowe. Wsparcie protokołu HTTP oraz formatu HTML przez wbudowane funkcje języka przyczyniły się do tego, iż PHP stał się popularnym środowiskiem tworzenia dynamicznych stron internetowych i na stałe zagościł w ofercie firm hostingowych.

Przykładem popularności tej technologii może być *Wikipedia, wolna encyklopedia*. Od strony technicznej wykorzystuje *MediaWiki*, która jest aplikacją internetową napisaną w PHP – umożliwia grupowe tworzenie, edycję i publikację treści. W 2008 roku serwis odnotowywał około 7 miliardów wyświetleń miesięcznie [7].

Język Java oraz związane z nią technologie internetowe takie jak *JSP*³ i *Servlet* stanowią kolejną alternatywę. Ich podstawową zaletą jest popularność języka oraz prostota wdrożenia aplikacji (serwery *Tomcat*, *Glassfish*). Standardy programistyczne języka Java są atutem przy grupowym tworzeniu aplikacji [8]. Minusem jest licencjonowanie niektórych bibliotek [9] oraz hosting wymagający wsparcia *Java EE*⁴. Dodatkowo język Java posiada statyczne typowanie i okazuje się mało elastyczny w niektórych zastosowaniach, dla których bywa porzucany na rzecz języków wyższego poziomu.

Wraz z rozwojem aplikacji internetowych stało się widoczne, iż pomiędzy implementacją aktualnych funkcji aplikacji, a językiem programowania brakuje pośredniczącej warstwy abstrakcji. Wiązało się to zarówno ze żmudnym powtarzaniem implementacji tych samych wzorców programistycznych w różnych aplikacjach, jak i chęcią uproszczenia całego procesu.

Początkowo powtarzające się wzorce programistyczne przyjmowały formę rozbudowanych bibliotek. Wkrótce jednak przekształciły się w pełnowartościowe środowiska tworzenia aplikacji, opierające się nie tylko na wykorzystaniu danego języka czy paradygmatu programowania, ale również na automatyzacji wielu popularnych czynności poprzez ustalone konwencje i dodatkowe narzędzia.

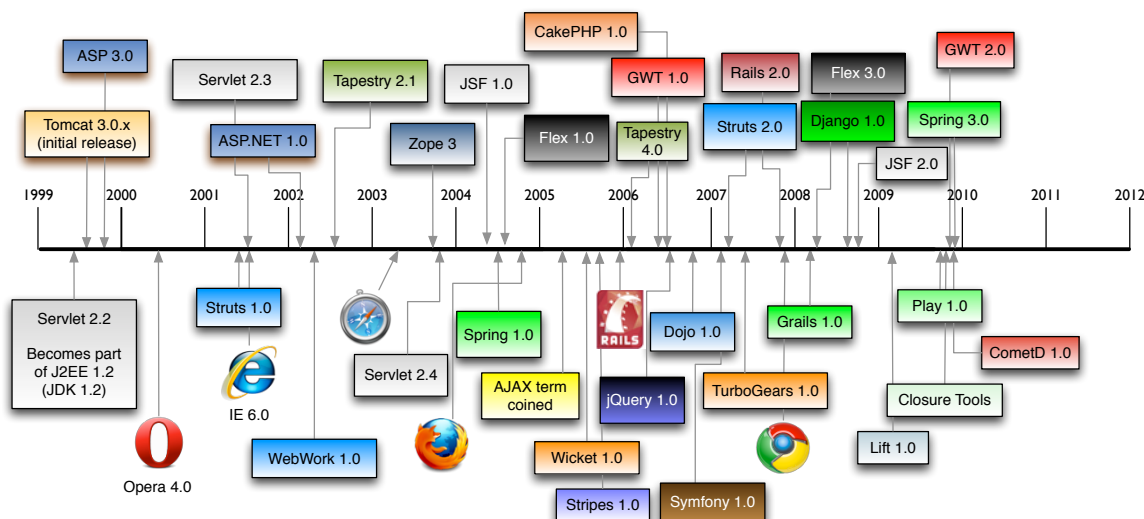
¹ Interpretowany język programowania ogólnego przeznaczenia, często wykorzystywany do pracy z danymi tekstowymi.

² ang. *Comprehensive Perl Archive Network*, globalne repozytorium zasobów dla języka Perl.

³ ang. *JavaServer Pages*

⁴ ang. *Java Platform, Enterprise Edition*

Powstały pierwsze frameworki w językach Java (1996: *WebObjects*) oraz Python (1998: *Zope 2*). Wkrótce pojawiły się odpowiedniki napisane w Perlu, PHP, czy też własnościowym ASP(.NET). Kolejne lata to wdrożenia różnych rozwiązań i kolejne wydania poprawiające błędy odnotowane w poprzednich, sprawdzonych w praktyce wersjach. Dalsza historia wybranych frameworków webowych została przedstawiona na rys. 2.3.



Rysunek 2.3. Historia rozwoju wybranych technologii i frameworków sieciowych

Źródło: <http://github.com/mraible/history-of-web-frameworks-timeline/>

Wydany w 2006 roku *Ruby On Rails* (RoR) stał się frameworkiem, który wygenerował duże poruszenie wśród programistów aplikacji sieciowych [10]. Jego pierwsza wersja charakteryzowała się użyciem dynamicznego języka wysokiego poziomu *Ruby* [11], paradygmatem MVC¹ oraz naciskiem na *konfigurację przez konwencję*. Wydanie RoR można potraktować jako początek nowego rozdziału w historii frameworków oraz metodologii tworzenia aplikacji sieciowych. Dzięki RoR programista mógł jeszcze bardziej skupić się na implementowaniu faktycznej funkcji aplikacji nie tracąc czasu na zajmowanie się niskopoziomą logiką biznesową związaną z protokołem HTTP i formatem HTML. Wprowadzono wiele rozwiązań, które stały się obowiązkową częścią składową współczesnych frameworków.

2.2.3. Bazy danych: RDBMS

Ostatnim elementem ewolucji aplikacji internetowych była ich „pamięć”, czyli bazy danych. Przez wiele lat królowały bazy relacyjne, które mimo swych ograniczeń bardzo dobrze spełniały swoją funkcję. Większość obecnie istniejących frameworków

¹ ang. *Model-View-Controller*, wzorzec projektowy polegający na podziale aplikacji na: model danych, warstwę prezentacji oraz logikę biznesową.

posiada bardzo dobre ORM¹, które umożliwiają wydajną i wygodną pracę z bazą bez konieczności wydania pojedynczej komendy SQL.

Niestety, wraz ze wzrostem popularności i rozwojem serwisów internetowych ograniczenia baz relacyjnych zaczęły stanowić poważny problem:

Schemat bazy danych, który należy zaprojektować przed utworzeniem aplikacji. Podczas dalszego rozwoju aplikacji, chcąc wprowadzić nową funkcjonalność, konieczna jest zmiana istniejącego schematu. Ponieważ pierwotny schemat bazy był zorientowany na konkretny zestaw operacji często zdarza się, że wprowadzana funkcja nie działa dostatecznie wydajnie na istniejącym schemacie. Konieczna jest jego gruntowna (zazwyczaj kosztowna i czasochłonna) przebudowa.

Dodatkowa warstwa abstrakcji, jaką tworzy środowisko protokołów, formatów i narzędzi wokół baz relacyjnych komplikuje budowę i tak już rozbudowanych aplikacji sieciowych. Każdy z popularnych obecnie RDBMS² wprowadza:

- Własną odmianę lub rozszerzenia języka SQL.
- Konieczność użycia pośrednika typu ODBC³ lub dedykowanego interpretera wewnętrznego protokołu komunikacji.
- Konieczność tłumaczenia specyfiki wybranej bazy na rozwiązania typu ORM, co często opóźnia dostępność nowych funkcji w API wyższego poziomu.
- Zestaw dedykowanych narzędzi zarządzających, z którymi należy się zapoznać, aby bezpiecznie i wydajnie użytkować wybraną bazę danych.

Większość z tych problemów jest niejako ukryta przed końcowym programistą. Korzystając z nowoczesnego frameworka używa on ORM. W większości przypadków nie odwołuje się ani razu do składni SQL: zapytania są generowane w tle, wiele z nich ORM optymalizuje automatycznie pod dany RDBMS. Niestety zdarza się, że rozwiązaniu temu brakuje odpowiedniej elastyczności lub wydajności i programista jest zmuszony – z uwagi na brak alternatyw – pisać fragmenty kodu operujące bezpośrednio na bazie za pomocą spreparowanych zapytań SQL.

Brak pełnej współbieżności, spowodowany występowaniem różnego rodzaju blokad nakładanych na elementy bazy danych. Blokady są narzucone ogólnie przez architekturę RDBMS. We współczesnym Internecie współbieżność stanowi istotne wyzwanie dla rozwiązań bazodanowych. Przykładem ilustrującym problem jest sytuacja, gdy wiele połączeń klienckich oczekuje na odczyt danej wartości, ponieważ pojedyncze ją zmienia. Operacja aktualizacji danych nakłada blokadę uniemożliwiającą operacje na tym samym zestawie danych. Blokada, która miała na celu zapewnić spójność informacji przechowywanych w bazie, przy odpowiedniej ilo-

¹ ang. *Object-relational mapping*, odwzorowanie oraz persystencja obiektów w strukturze bazy danych.

² ang. *Relational Database Management System*, środowisko złożone z relacyjnej bazy danych i powiązanych narzędzi.

³ ang. *Open DataBase Connectivity*, niezależne od języka programowania API komunikacji z bazą danych.

ści równoczesnych zapytań tworzy sieć znacznie bardziej poważnych problemów – spowalnia działanie całej aplikacji, wymuszając wdrożenie technik zmniejszających nacisk na bazę danych.

Skalowalność, która teoretycznie powinna być odpowiedzią na problemy wynikające z coraz większego obciążenia baz danych przez aplikacje internetowe. Niestety *skalowanie pionowe*, czyli zakup sprzętu o lepszych parametrach posiada zarówno technologiczne (ogranicza nas wydajność istniejących rozwiązań) jak i finansowe (ogranicza nas budżet) limity. Stąd *skalowanie poziome* (dodawanie większej liczby urządzeń) jest często preferowaną drogą rozwoju instalacji bazodanowych. Niestety zapewnienie synchronizacji pomiędzy serwerami jest zagadnieniem złożonym i karkołomnym. Założenia architektury RDBMS ponownie ograniczają programistów. Większość baz relacyjnych (np. MySQL) obsługuje replikację jednokierunkową, czyli architekturę *master-slave*, która z założenia wyklucza węzły typu slave z interaktywnego użycia. Inny przykład: blokady nakładane przez operacje zapisu podczas procesu replikacji wyłączają dany węzeł z praktycznego użycia. Niedogodności tego typu jest o wiele więcej, jednak nie są one przedmiotem tej pracy, a wymienione tutaj służą jedynie ilustracji jak złożone jest zagadnienie skalowania RDBMS.

Doświadczony administrator baz danych przy odpowiednim nakładzie pracy jest w stanie rozwiązać wszystkie z wyżej wymienionych niedogodności. Problemem zbiorczym pozostaje czas, poziom skomplikowania oraz koszt tego typu rozwiązań. Należy również mieć świadomość, iż system, który sprawdza się obecnie, w przyszłości może napotkać ograniczenia natury technicznej, których obejście nie będzie już opłacalne przy zastosowaniu RDBMS.

2.2.4. Bazy danych: NoSQL

Oczywiste jest, iż w większości przypadków bazy relacyjne są sprawdzonym, wystarczającym rozwiązaniem. Specyfika aplikacji internetowych kładzie jednak coraz częściej nacisk na współbieżność, skalowalność oraz dostępność. W pierwszej dekadzie dwudziestego pierwszego wieku sytuacja na rynku rozwiązań bazodanowych zaczęła ulegać zmianie. Największe serwisy internetowe przestały inwestować w infrastruktury RDBMS i rozpoczęły poszukiwania alternatywnych rozwiązań. Ruch ten, potocznie nazywany *NoSQL*¹, zaowocował powstaniem wielu nowych, wydajnych systemów przechowywania danych [12].

Cechy charakteryzujące większość rozwiązań ruchu NoSQL:

- wydajne i proste skalowanie poziome,
- brak sztywnego schematu bazy danych,
- dane przechowywane jako proste struktury (takie jak słowniki i listy),
- możliwość przechowywania dużych ilości danych,

¹ ang. *Not Only SQL*, synonim alternatyw dla relacyjnej bazy danych [12].

— brak zapytań SQL (zastąpione przez *widoki* lub analogiczne rozwiązania).

W roku 2010 wiele największych i najpopularniejszych serwisów internetowych nie używa już RDBMS. Wiele innych jest w trakcie migracji na rozwiązania NoSQL lub planuje ją w niedalekiej przyszłości. Proces ten ilustrują dwa przykłady.

Przykład 2.1: *Google BigTable*. Własnościowy projekt rozpoczęty w 2004 roku, obecnie jest podstawowym rozwiązaniem bazodanowych dla wielu produktów firmy Google: *Search*, *Reader*, *Maps*, *Book Search*, *Earth*, *Blogger.com*, oraz *Code* [13].

Zaprojektowany z myślą o ekstremalnych potrzebach współczesnych aplikacji internetowych jest w stanie operować na ilościach danych liczonych w petabajtach.

Przykład 2.2: *Cassandra*. Stworzony na potrzeby serwisu społecznościowego Facebook [14]. System został udostępniony jako wolne oprogramowanie i jest obecnie rozwijany przez Fundację Apache¹. Podobnie jak BigTable, oferuje magazyn danych w postaci *klucz-wartość*. Należy zaznaczyć, iż system w chwili pisania niniejszej pracy używany między innymi przez serwisy takie jak *Facebook*², *Digg*³, *Reddit*⁴ czy *Twitter*⁵.

2.3. Wykorzystane narzędzia

2.3.1. Język programowania Python

Python został stworzony przez holenderskiego programistę Guido van Rossuma w 1990 roku. Jest multi-paradygmatowym językiem ogólnego użytku stawiającym na oszczędną składnię i czytelny kod. Wspiera programowanie obiektowe, strukturalne i funkcyjne [2, 15].

Zalety

Będą to główne cechy języka Python:

- dojrzały i sprawdzony,
- prosta, przejrzysta i czytelna składnia,
- interpretowany, interaktywny,
- dynamiczny system typów,
- automatyczne zarządzanie pamięcią,
- bogaty wybór otwartych bibliotek.

Wady

Do często wymienianych słabych stron języka należą:

¹ ang. *Apache Software Foundation*, organizacja non-profit wspierająca projekty open-source

² Największy (w chwili pisania pracy – pod względem liczby użytkowników) serwis społecznościowy (<http://facebook.com>).

³ Największy społecznościowy agregator linków (<http://digg.com>).

⁴ Drugi co do wielkości społecznościowy agregator linków (<http://reddit.com>).

⁵ Największy serwis mikroblogowy (<http://twitter.com>).

- *brak pełnej enkapsulacji*

Zarzut wynika z braku znajomości założeń środowiska Python (jednym z nich jest zasada nieograniczania programisty przez architekturę języka).

- *prędkość*

Istnieje wiele rozwiązań problemu wydajności. Domyślnie kod jest kompilowany do dodatkowych plików zawierających *byte code* przyspieszający start aplikacji. Dla większości zastosowań domyślny interpreter jest wystarczająco szybkim rozwiązaniem. Partie kodu wymagające wyjątkowej wydajności można oddelegować do modułów pisanych w języku C lub specjalnej odmianie języka Python ze statycznym typowaniem: *Cython*[16].

- *problemy z wielowątkowością*

Rozwiązanie zależy od używanego interpretera języka. W większości przypadków problemem jest niewiedza programisty przyzwyczajonego do metodologii innego języka. Zagadnienie nie dotyczy aplikacji internetowych omawianych w niniejszej pracy.

Porównanie z wybranymi językami

Aplikacje internetowe tworzone są za pomocą wielu różnych języków. W większości przypadków są to języki ogólnego użytku i nie posiadają one wbudowanej obsługi protokołu HTTP, formatów HTML i pokrewnych technologii. Odpowiadają za nie dodatkowe biblioteki oraz frameworki. Razem, elementy te tworzą środowisko programistyczne aplikacji. Z powodu różnic w budowie i przeznaczeniu języków ich porównanie jest trudnym i często bezcelowym zadaniem.

Chcąc porównać języki należy określić dziedzinę oraz zakres interesujących nas zagadnień. Z uwagi na tematykę niniejszej pracy, *tabela 2.1* przedstawia zestawienie wybranych właściwości popularnych języków wykorzystywanych do tworzenia aplikacji internetowych. Opierając się na przedstawionym porównaniu, można stwierdzić, iż obecnie wybór języka często sprowadza się do indywidualnych preferencji programisty oraz specyfiki wykonywanego projektu.

Język Python sprzyja tworzeniu spójnego kodu (szczególnie istotne przy pracy grupowej), dając przy tym elastyczność, której brakuje niektórym językom. Dodatkowym atutem jest bogaty wybór otwartych bibliotek wysokiej jakości. Istnieje wiele frameworków wspomagających tworzenie aplikacji internetowych w języku Python. Do najbardziej popularnych należą: *Django*, *Pylons*, *TurboGears*, *CherryPy*, *Zope*, *WebPy* oraz *Grok*.

Dzięki filozofii *batteries included*¹, Python oraz oparte na nim rozwiązania są ciekawą alternatywą, która pozwala tworzyć rozbudowane aplikacje przy użyciu minimalnej ilości kodu i niskim nakładzie pracy.

¹ ang. „baterie w zestawie”

Tabela 2.1. Porównanie wybranych języków pod kątem tworzenia aplikacji sieciowych
(Źródło: opracowanie własne)

	Python	PHP	Java	Perl	Ruby
wsparcie paradygmatu OOP ^a	✓	✓ ^b	✓	✓	✓
obsługa HTTP w <i>bibliotece standardowej</i>	✓	✓	✓	✓	✓
dynamiczne typowanie	✓	✓	✗	✓	✓
przenośność	✓	✓	✓	✓	✓
bogaty wybór bibliotek	✓	✓	✓	✓	✓ ^c
wymagana maszyna wirtualna	✗ ^d	✗	✓	✗	✗ ^e
wolne oprogramowanie	✓	✓ ^f	✗ ^g	✓	✓

^a ang. *Object Oriented Programming*

^b Dostępne od wersji 5.

^c Biorąc pod uwagę fakt, że język jest stosunkowo młody.

^d Opcjonalna: alternatywne implementacje – *Jython* oraz *IronPython*.

^e Opcjonalna: alternatywna implementacja – *Jruby*.

^f Uwaga: używana licencja odrzuca zasadę *copyleft*.

^g Sun wydał Javę na licencji GPL, jednak część podstawowych modułów nadal jest własnościowa.

2.3.2. Framework Django

Projekt Django przeszedł interesującą ewolucję. Powstał jako własnościowe rozwiązanie amerykańskiej firmy programistycznej, w 2005 roku został wydany na licencji BSD. W 2008 roku powołano do życia *Django Software Foundation*, organizację, która czuwa nad dalszym rozwojem frameworka [17].

Django posiada kilka cech, które czynią go godnym uwagi.

Intuicyjny ORM

Jak większość elementów Django, wbudowany ORM jest intuicyjny i idiomatyczny. Znajomość podstawowych struktur obecnych w języku jest wystarczająca, aby rozpocząć implementację modelu danych.

Automatycznie generowany panel administracyjny

Czas tworzenia aplikacji internetowych może zostać znacznie skrócony dzięki funkcji generowania panelu administracyjnego. Utworzony panel implementuje komplet operacji CRUD na instancjach wybranych klas modelu danych. Programista oszczędza czas i może skupić się na dodaniu logiki biznesowej specyficznej dla danej aplikacji.

Własny język szablonów

System szablonów wykorzystany w Django (zgodnie z paradygmatem MVC) oddziela warstwę prezentacji danych od logiki biznesowej aplikacji. Konwencja *dzielenia szablonów* zmniejsza nakład pracy i zwiększa czytelność warstwy prezentacji. Nie ogranicza się jedynie do HTML, umożliwia pracę z dowolnym tekstowym formatem (np. *CSV*¹).

¹ ang. *Comma-Separated Values*, format przechowywania danych w plikach tekstowych.

Kluczowe zagadnienia: *dziedziczenie*, *tagi*, *zmienne* oraz *filtry* zostaną omówione w dalszej części pracy.

Elastyczny system cache

Programista może decydować, dla których elementów aplikacji cache będzie aktywne. Mogą to być zarówno jedynie kosztowne do wygenerowania fragmenty szablonów, jak i cały serwis. Zachowanie systemu cache może być uzależnione od czynników takich jak preferowany przez użytkownika język. Zagadnienie dotyczy obiektów, odpowiedzi bazy danych oraz nagłówków protokołu HTTP.

Wsparcie internacjonalizacji

Twórca aplikacji może określić elementy zależne od preferowanego języka użytkownika. Tłumaczenie odpowiednio przygotowanych aplikacji sprowadza się do dodania nowego pliku lokalizacyjnego.

Modularność, mini-aplikacje wielokrotnego użytku

Stworzona aplikacja Django może być w przyszłości wykorzystana jako moduł składowy innego projektu. Dzięki temu programista oszczędza czas wdrażając gotowe i sprawdzone rozwiązanie popularnego problemu.

Batteries included

Django zawiera wiele wbudowanych rozwiązań gotowych do użycia. Obsługa email, tworzenie kanałów RSS/Atom, system uwierzytelniania oraz autoryzacji to jedynie niektóre z nich. Programista otrzymuje narzędzia, dzięki którym może skupić się na implementacji specyficznych funkcji rozwijanego projektu.

Wybrane elementy Django wykorzystano w przykładowej aplikacji. Zostały one omówione w kolejnym rozdziale pracy.

2.3.3. Baza CouchDB

CouchDB jest współbieżną bazą danych zorientowaną na dokumenty. Jej autorem jest Damien Katz¹. Początkowo finansowany przez IBM, obecnie projekt jest rozwijany jako wolne oprogramowanie pod patronatem Fundacji Apache (licencja *Apache 2.0*).

Baza została zaprojektowana z myślą o technologiach obecnych w Internecie. Ma bardzo dobrą krzywą uczenia: wiedza, jaką programista aplikacji sieciowych już posiada (HTTP, JavaScript, JSON²) zostaje przeniesiona na zagadnienia związane z reprezentacją i przesyłaniem danych zawartych w bazie.

CouchDB charakteryzuje się:

- odpornością na awarie,
- pełną współbieżnością i brakiem blokad znanych z baz relacyjnych,
- dwukierunkową replikacją i wzorowym skalowaniem poziomym,
- wykorzystaniem *B-drzew*³ jako wewnętrznej struktury danych.

¹ Były pracownik IBM, założyciel Couch.io (<http://couch.io>).

² ang. *JavaScript Object Notation*, format tekstowy, będący podzbiorem języka JavaScript.

³ Drzewiasta struktura przechowująca pary *klucz-wartość* w uporządkowany liniowo sposób.

Pomimo młodego wieku, CouchDB jest używana między innymi przez:

- *Ubuntu One*, usługa dysku sieciowego obecna w systemie operacyjnym Ubuntu (od wersji 9.04).
- *Raindrop*¹, eksperymentalna platforma agregująca kanały komunikacji. Rozwijana przez Mozilla Foundation.
- *IBM*
- *BBC*²
- *Meebo*³

Szczegółowy opis wybranych zagadnień związanych z CouchDB znajduje się w rozdziale 3.2 (strona 21).

2.3.4. Serwer Nginx

Nginx jest wydajnym serwerem HTTP oraz proxy napisanym przez Igora Syssojewa, jako alternatywa dla ociężałych serwerów typu Apache. W przeciwieństwie do klasycznych serwerów HTTP nie polega na wątkach podczas obsługi zapytań. Używa opartej na zdarzeniach, asynchronicznej architektury, która gwarantuje niskie i łatwe do przewidzenia zużycie pamięci, nawet podczas dużych obciążeń [6].

Do wartych podkreślenia właściwości należą:

- budowa modułarna,
- rozbudowana funkcjonalność *reverse proxy*,
- wsparcie FastCGI,
- elastyczna konfiguracja,

Używany przez ponad 7% serwisów internetowych, jest trzecim pod względem popularności serwerem HTTP w Internecie. Od kilku lat Nginx odnotowuje konsekwentny wzrost liczby wdrożeń: o 337% w roku 2008 oraz 384% w 2009 [18].

Opis zaawansowanej konfiguracji serwera Nginx z przykładową aplikacją wykonaną w Django znajduje się w rozdziale 6 (strona 81).

2.4. Środowisko programistyczne

Aplikacja będąca przedmiotem niniejszej pracy została wykonana przy użyciu konkretnego zestawu narzędzi. W chwili rozpoczęcia prac składały się na nie:

Gentoo GNU/Linux

System operacyjny, metadystrybucja GNU/Linux o charakterze ciągłym.

Vim

Rozbudowany edytor tekstowy/IDE w wersji 7.2.

Python

Domyślny interpreter w wersji 2.6.4.

¹ <https://mozillalabs.com/raindrop>

² <http://twitter.com/endafarrell/status/9820160677>

³ <http://www.meebo.com/>

Django

Framework wspomagający tworzenie aplikacji internetowych, wersja 1.1.1.

CouchDB

Baza danych zorientowana na dokumenty, wersja 0.10.1.

Instalacja wyżej wymienionego oprogramowania pod systemem Gentoo GNU/Linux sprowadza się do wydania jednego polecenia:

```
emerge -av vim couchdb django couchdb-python python-musicbrainz
```

Wszelkie zależności i wymagane biblioteki zostaną automatycznie dodane i zainstalowane. Nowy projekt Django tworzy się (będąc w pożądanym katalogu) poleceniem:

```
django-admin.py startproject nazwa_projektu
```

Edytor Vim posiada wbudowaną obsługę języka Python, stąd żadna dodatkowa konfiguracja nie jest konieczna. Tabulacje oraz wcięcia są reprezentowane przez cztery spacje – ogólnie przyjętą konwencję programistyczną języka [19]. Podczas pisania aplikacji programista nie musi uruchamiać dedykowanego serwera HTTP. Django posiada wbudowaną funkcjonalność obsługi zapytań HTTP. Należy zaznaczyć, że domyślnie nie hostuje on plików statycznych, stąd konieczne jest dodanie do (znajdującego się w katalogu głównym projektu) pliku `urls.py` następującej deklaracji¹:

```
(r'^static/(?P<path>.*)$', 'django.views.static.serve',  
    {'document_root': settings.STATIC_DOC_ROOT, 'show_indexes': True})
```

Wyrażenie regularne przechwyci każde zapytanie skierowane do `/static/*`, następnie przekaże je do prostego pomocnika zwracającego pliki statyczne.

Deklarację lokalnej ścieżki do statycznych zasobów umieszcza się w pliku `settings.py`:

```
STATIC_DOC_ROOT = '/home/user/django_app/mmda/static'
```

Uwaga: ponieważ hostowaniem statycznych plików powinien zajmować się właściwy serwer HTTP (a nie aplikacja Django, która nie jest zoptymalizowana pod to zadanie), należy pamiętać o wyłączeniu wyżej wymienionego kodu po zakończeniu testów aplikacji.

Tak zmodyfikowane środowisko uruchamia się poleceniem:

```
python manage.py runserver
```

Uruchomiona aplikacja znajduje się pod adresem `http://127.0.0.1:8000/`.

Środowisko programistyczne jest gotowe do pracy.

¹ Wyjaśnienie mechanizmu *URLconf* znajduje się w rozdziale 3.4.1 (strona 39)

3. Wykorzystane elementy Django oraz CouchDB

3.1. Metodyka Django

3.1.1. Podstawowe pojęcia

Przed rozpoczęciem pracy z frameworkiem Django oraz innymi elementami warto zapoznać się z terminologią z nimi związaną. Ponieważ dokumentacje oraz omawiane projekty rozwijane są w języku angielskim, polskie odpowiedniki używane przez autora należy traktować jedynie jako wskazówkę zwiększającą czytelność dalszej części pracy.

Poniżej znajduje się wyjaśnienie wybranych pojęć mogących budzić wątpliwości u czytelnika.

Aplikacja

W zależności od kontekstu termin *aplikacja* może oznaczać:

- *Aplikację internetową*

W niniejszej pracy jest to projekt Django o nazwie `mmda` wraz z towarzyszącymi mu technologiami (jQuery, Nginx, CouchDB).

- *Aplikację Django*

Część projektu Django. Zawiera w sobie logikę biznesową realizującą konkretne zadanie. Pojedyncza aplikacja Django (o ile pozwala na to jej specyfika) może być wykorzystywana w wielu niezależnych projektach.

Projekt

Projektem nazywany jest zbiór wielu aplikacji Django wchodzących w skład jednej aplikacji internetowej. Posiadają wspólną konfigurację oraz wykorzystują wzajemnie swoje logiki biznesowe.

Widok

Przy omawianiu zagadnień związanych z CouchDB *widokiem* nazywany jest zestaw wyników wygenerowanych za pomocą funkcji Map/Reduce na podstawie dokumentów bazy danych.

Gdy przedmiotem opisu jest Django, *widokiem* nazywana jest funkcja znajdująca się w pliku `views.py`, wywoływana przez przypisany do niej w `urls.py` URL.

Dokument

Gdy pojęcie używane jest w kontekście bazy CouchDB, jego znaczenie nie wymaga wyjaśnień.

Dokumentem nazywany jest również obiekt będący instancją klasy zdefiniowanej w pliku `models.py`. Jest to oczywisty skrót myślowy: obiekt ten jest reprezentacją dokumentu CouchDB przez obiekt języka Python wraz z metodami dostarczonymi przez Couchdbkit, ułatwiającymi jego obsługę.

3.1.2. Budowa projektu Django

Każdy projekt Django jest zwykłą aplikacją napisaną w języku Python. Programista nie musi uczyć się nowej składni ani narzędzi do zarządzania projektem. Katalog główny projektu oraz każdej wchodzącej w jego skład aplikacji zawiera plik `__init__.py`. Oznacza on, iż dany katalog jest *pakiem* języka Python [2].

Listing 3.1 przedstawia uporządkowany przez autora rezultat wywołania polecenia `tree` na zawierającym projekt katalogu `mmda`. Zawartość katalogów `static` oraz `templates` została pominięta dla zwiększenia czytelności.

Listing 3.1. Uproszczona struktura
przykładowego projektu Django

```

1  mmda
2  |-- __init__.py
3  |-- artists/
4  |   |-- __init__.py
5  |   |-- _design/
6  |   |   |-- views/
7  |   |       |-- release_groups/
8  |   |           |-- map.js
9  |   |           |-- releases/
10 |   |               |-- map.js
11 |   |-- templatetags/
12 |   |   |-- __init__.py
13 |   |   |-- release_helpers.py
14 |   |-- models.py
15 |   |-- urls.py
16 |   |-- views.py
17 |-- news/
18 |   |-- __init__.py
19 |   |-- _design/
20 |   |   |-- views/
21 |   |       |-- artist_news/
22 |   |           |-- map.js
23 |   |-- models.py
24 |   |-- urls.py
25 |   |-- views.py
26 |-- pictures/
27 |   |-- __init__.py
28 |   |-- _design/
29 |   |   |-- views/
30 |   |       |-- best_pictures/
31 |   |           |-- map.js
32 |   |-- models.py
33 |   |-- urls.py
34 |   |-- views.py
35 |-- search/
36 |   |-- _design/
37 |   |-- __init__.py
38 |   |-- models.py
39 |   |-- urls.py
40 |   |-- views.py
41 |-- tags/
42 |   |-- __init__.py
43 |   |-- _design/
44 |   |-- models.py
45 |   |-- urls.py
46 |   |-- views.py
47 |-- videos/
48 |   |-- __init__.py
49 |   |-- _design/
50 |   |-- models.py
51 |   |-- urls.py
52 |   |-- views.py
53 |-- engine/
54 |   |-- __init__.py
55 |   |-- api/
56 |   |   |-- __init__.py
57 |   |   |-- flickr.py
58 |   |   |-- lastfm.py
59 |   |   |-- musicbrainz.py
60 |   |   |-- recaptcha.py
61 |   |   |-- youtube.py
62 |   |-- abstract.py
63 |   |-- artist.py
64 |   |-- cache.py
65 |   |-- feeds.py
66 |   |-- future.py
67 |   |-- news.py
68 |   |-- pictures.py
69 |   |-- release.py
70 |   |-- search.py
71 |   |-- tag.py
72 |   |-- utils.py
73 |   |-- videos.py
74 |-- static/
75 |   |-- (...)
76 |-- templates/
77 |   |-- (...)
78 |-- urls.py
79 |-- settings.py
80 |-- manage.py

```

Poniżej znajduje się jedynie krótka charakterystyka wybranych elementów. Zostaną omówione bardziej szczegółowo w kolejnych rozdziałach.

settings.py

Znajdujący się w głównym katalogu plik zawiera konfigurację projektu zapisaną za pomocą składni języka Python (wiersz 79).

manage.py

Skrypt automatyzujący wykonywanie popularnych czynności związanych z frameworkiem Django (tworzenie nowych aplikacji, synchronizowanie bazy danych, uruchamianie testowego serwera HTTP i tym podobne).

urls.py

Definicje obsługiwanych przez projekt adresów URL. Główny plik (wiersz 78) może odwoływać się do szczegółowych konfiguracji poszczególnych aplikacji wchodzących w skład projektu (wiersze 15,24,33,39,45,51).

artists/ news/ pictures/ search/ tags/ videos/

Aplikacje Django odpowiedzialne za wybrane funkcje projektu.

engine/

Pakiet niebędący aplikacją Django. Zawiera logikę biznesową specyficzną dla tworzonej aplikacji internetowej (agregacja oraz buforowanie treści z zewnętrznych API). Jest przykładem biblioteki niezwiązanej bezpośrednio z frameworkiem, ale wykorzystywanej przez należące do projektu aplikacje.

templatetags/

Pakiet zawiera definicje dodatkowych filtrów oraz tagów.

models.py

Znajdujący się w aplikacji Django plik zawiera model danych. W tworzonej aplikacji klasyczny ORM został zastąpiony przez Couchdbkit, stąd plik **models.py** zawiera modele dokumentów CouchDB.

views.py

Plik zawiera funkcje widoków obsługiwanych przez daną aplikację.

_design

Katalog przechowuje kopię dedykowanego danej aplikacji tzw. *design document* bazy CouchDB. Dokument tego typu może zawierać między innymi definicje funkcji Map/Reduce tworzących widok bazy danych (wiersze 8,10,22,31).

templates/

Katalog zawiera szablony wykorzystywane przez funkcje widoków Django do generowania odpowiedzi HTML.

static/

Katalog zawiera dane statyczne (grafikę, CSS, JS) wykorzystywane przez projekt.

3.1.3. Nowa aplikacja

Utworzenie nowej aplikacji sprowadza się do wydania¹ polecenia:

```
python manage.py startapp artists
```

Następnie należy włączyć aplikację do projektu. Konfiguracja projektu Django znajduje się w pliku `settings.py`:

```
INSTALLED_APPS = (  
    'couchdbkit.ext.django',  
    'mmda.artists',  
)
```

Aplikacja została włączona do projektu. Po przejściu do katalogu `artists` programista może rozpocząć pracę od edycji pliku `views.py` lub opcjonalnych `models.py` oraz `urls.py`. Konfiguracja aplikacji do pracy z CouchDB zostanie omówiona w dalszej części pracy.

3.2. Model danych oparty o CouchDB

Tak jak większość obecnie używanych frameworków, Django domyślnie używa wbudowanego rozwiązania typu ORM jako pośrednika pomiędzy obiektową warstwą abstrakcji aplikacji a relacyjną bazą danych. Jest to bardzo prosta i intuicyjna technika, która dla zachowania przejrzystości nie zostanie opisana w tej pracy.

Aby zademonstrować elastyczność, jaką daje pisanie aplikacji w języku Python, autor zrezygnował z użycia klasycznej, relacyjnej bazy danych. Tworzona aplikacja będzie wykorzystywać bazę zorientowaną na dokumenty – CouchDB. Z uwagi na zupełnie odmienne podejście do zagadnienia, opis integracji Django z CouchDB zostanie poprzedzony krótkim wstępem teoretycznym.

3.2.1. Dokumenty

Bazy danych zorientowane na dokumenty są jednym z rozwiązań proponowanych przez wspomniany w poprzednim rozdziale ruch NoSQL. Charakterystyczne właściwości baz tego typu zawierają się w kilku kluczowych zagadnieniach:

Brak schematu bazy

Dane są udostępniane dla aplikacji zewnętrznych jako płaski bank danych.

Dokument jako podstawowa jednostka danych

Dane są przechowywane w strukturach, które są dosyć wiernym odwzorowaniem dokumentów z rzeczywistego świata. Każdy przechowywany w bazie dokument zawiera cały kontekst konieczny do jego identyfikacji.

¹ W katalogu głównym projektu.

Dynamiczna struktura dokumentu

Dokument nie posiada narzuconej wewnętrznej struktury. Nie istnieje zestaw wymaganych pól, jakie powinien zawierać dokument danego 'typu'.

Struktury danych znane programistom

Dane przechowywane są w listach oraz listach asocjacyjnych. Sam dokument często bywa słownikiem, którego identyfikator jest jednym z kluczy.

JSON jest lekkim formatem używanym do serializacji i wymiany danych w Internecie. Umożliwia przesyłanie popularnych typów i struktur. Swoją popularność zdobył dzięki technologii AJAX¹, gdzie funkcjonuje jako alternatywa dla XML². JSON jest domyślnym formatem dokumentów w CouchDB, dzięki czemu programista nie jest zmuszony uczyć się nowej składni, ale pracuje ze znaną – czytelną dla człowieka – technologią. Kolejną zaletą tego rozwiązania jest fakt, iż JSON jako popularny format wśród aplikacji internetowych posiada bogate wsparcie ze strony języków programowania oraz narzędzi programistycznych. Dokumenty CouchDB mogą być parsowane i przetwarzane za pomocą już istniejących bibliotek i aplikacji.

Przedstawiony w *listingu 3.2* format dokumentu jest przejrzysty i czytelny dla człowieka.

Listing 3.2. Przykładowy dokument CouchDB w formacie JSON

```
1 {
2   "_id": "1f641f00-3802-47b9-8ba7-50f3349ebe77",
3   "_rev": "3-54a2434732a0a25d7808af8d74bb472d",
4   "artist_mbid": "e9dfc148-d5f6-425e-b80b-f99ed2bd7c09",
5   "title": "All Is Violent, All Is Bright",
6   "year": 2005,
7   "tags": [
8     "instrumental",
9     "post-rock"
10  ],
11  "_attachments": {
12    "cover.jpg": {
13      "stub": true,
14      "content_type": "image/jpeg",
15      "length": 20930,
16      "revpos": 2
17    }
18  }
19 }
```

¹ ang. *Asynchronous JavaScript and XML*

² ang. *Extensible Markup Language*, uniwersalny język reprezentacji danych w strukturalizowany sposób.

Zbiór par *klucz-wartość* jest podstawową strukturą zaimplementowaną w różnych językach jako słownik lub tabela asocjacyjna. Dzięki temu możliwe jest naturalne i bezpośrednie mapowanie dokumentów bazy na obiekty/konstrukty wybranych języków programowania.

W CouchDB każdy dokument jest identyfikowany przez wartość przechowywaną w polu `'_id'`. Może być nim dowolny unikalny ciąg znaków. Pole `'_rev'` zawiera identyfikator rewizji dokumentu. Każda zmiana przeprowadzona na dokumencie powoduje zmianę wartości tego pola. CouchDB posiada wbudowany system kontroli wersji nazwany MVCC¹, który umożliwia działanie bazy bez konieczności wprowadzania jakichkolwiek blokad. Wiele wersji tego samego dokumentu może istnieć równocześnie, jedynie najnowsza jest zwracana przy zapytaniu dotyczącym danego dokumentu [1].

Pozostałe pola dokumentu z *listingu 3.2* posiadają wartości będące ciągiem znaków (wiersz 5), liczbą (6), listą (7-10) lub słownikiem (11-18). Dokument może posiadać załączniki. Zmienna `stub` oznacza, iż w ramach optymalizacji baza zwróciła jedynie dotyczące ich metainformacje. Przesłanie faktycznych załączników następuje jedynie na wyraźne życzenie programisty.

3.2.2. Widoki

Odczyt całego dokumentu na podstawie jego identyfikatora jest mało elastycznym rozwiązaniem. Zdarza się, że programista chce filtrować, porównać lub zestawić dane należące do różnych dokumentów. Często interesuje go jedynie wybrany zestaw wartości lub dane posortowane według danego pola. Tego typu potrzeby zaspokajają *widoki*.

Rozróżniane są widoki *statyczne* i *dynamiczne*. Widok dynamiczny jest tymczasowym rozwiązaniem używanym w procesie tworzenia aplikacji. Po zakończeniu pracy należy go zamienić na drugi typ. Widoki statyczne definiowane są w specjalnych dokumentach (`_design/*`).

Widoki generowane są za pomocą techniki *Map/Reduce*, czyli duetu dwóch funkcji operujących na dokumentach. Technika Map/Reduce jest uniwersalna, niezależna od języka. Domyślna implementacja serwera widoków w CouchDB wykorzystuje silnik JavaScript *Mozilla Spidermonkey*². Jest to kolejny przykład na to, iż CouchDB stawia na wykorzystanie już istniejących formatów i technologii, dzięki czemu programista może szybko rozpocząć właściwą pracę – nie tracąc czasu na naukę nowych protokołów, formatów, narzędzi i ich składni.

Funkcja *Map* wykonywana jest na każdym dokumencie. Na wejściu otrzymuje dokument, na wyjściu generuje parę *klucz-wartość*. Z uwagi na użycie B-drzew jako wewnętrznej struktury przechowywania informacji w CouchDB, wygenerowana lista wyników jest automatycznie posortowana według kluczy.

¹ ang. *MultiVersion Concurrency Control*, system kontroli wersji i zapobiegania konfliktom w bazie danych.

² Implementacja JavaScript napisana w języku C, używana m.in. przez przeglądarkę *Firefox 3.6*.

Funkcja *Reduce* jest opcjonalna i powinna być używana z rozważą. Przetwarza elementy zwrócone przez funkcję *Map*. Jest wykorzystywana, gdy zachodzi potrzeba znacznej redukcji przetwarzanych danych, np. dużej ilości tekstu do liczbowych reprezentacji wielkości skalarnych.

Główną zaletą widoków jest to, iż w przeciwieństwie do zapytań SQL znanych z baz relacyjnych są definiowane (znane) po stronie bazy danych. Dzięki temu możliwe jest wprowadzenie wielu optymalizacji. Przetwarzanie wszystkich dokumentów przez funkcję *Map* ma miejsce jedynie za pierwszym razem. Tak utworzona lista wyników jest zapisywana w postaci B-drzewa. Kolejne wywołania widoku bazują na istniejącym już B-drzewie wyników i aktualizują je, jeżeli powiązane dokumenty uległy zmianie [1].

3.2.3. HTTP API

W CouchDB komunikacja z bazą odbywa się za pomocą protokołu HTTP. Interfejs HTTP implementuje założenia *REST*¹. Zasoby bazy są identyfikowane przez *URI*². Operacje CRUD³ realizowane są za pomocą czterech metod HTTP: *PUT* (tworzenie), *GET* (odczyt), *POST* (aktualizacja/edycja) oraz *DELETE* (usuwanie).

Przykład z *listingu 3.3* przedstawia odczyt dokumentu o identyfikatorze *docid* z bazy *dbname* z lokalnej instancji CouchDB za pomocą narzędzia *curl*⁴.

Listing 3.3. Odczyt dokumentu z bazy CouchDB

```

1 $ curl -v -X GET http://127.0.0.1:5984/dbname/docid
2 * About to connect() to 127.0.0.1 port 5984 (#0)
3 *   Trying 127.0.0.1... connected
4 * Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
5 > GET /dbname/docid HTTP/1.1
6 > User-Agent: curl/7.19.6 (x86_64-pc-linux-gnu) (...)
7 > Host: 127.0.0.1:5984
8 > Accept: */*
9 >
10 < HTTP/1.1 200 OK
11 < Server: CouchDB/0.10.1 (Erlang OTP/R13B)
12 < Etag: "1-55fa92c1a80a923f12ca22e79eefded1"
13 < Date: Tue, 09 Mar 2010 19:23:37 GMT
14 < Content-Type: text/plain; charset=utf-8
15 < Content-Length: 72
16 < Cache-Control: must-revalidate
17 <
18 {"_id":"docid","_rev":"1-55fa92c1a80a923f12ca22e79eefded1","test":true}
19 * Connection #0 to host 127.0.0.1 left intact
20 * Closing connection #0

```

¹ ang. *REpresentational State Transfer*

² ang. *Uniform Resource Identifier*, internetowy standard umożliwiający łatwą identyfikację zasobów.

³ ang. *Create, Read, Update, Delete*, cztery podstawowe operacje na bazie danych.

⁴ Program wykorzystujący bibliotekę do obsługi protokołu HTTP o tej samej nazwie.

Linie rozpoczynające się od znaku `'>'` oznaczają zapytanie wysłane do serwera. Odpowiedź bazy rozpoczyna się znakiem `'<'`. Linia 18 zawiera żądany dokument w formacie JSON.

3.2.4. Zalety

HTTP, JSON, JavaScript

Użycie HTTP jako protokołu komunikacji z bazą wiąże się z licznymi korzyściami. Wiele narzędzi związanych z tym protokołem może być wykorzystanych podczas pracy z CouchDB. Widoczne w dwunastej linii *listingu 3.3* pole **Etag** zawiera identyfikator rewizji¹ przesyłanego dokumentu. Na jego podstawie można opracować systemy rozłożenia obciążenia lub cache'owania danych. Kolejnym atutem może być fakt, iż do analizy wykorzystania bazy danych wystarczą logi pośredniczącego serwera HTTP.

Język JavaScript oraz format JSON są znane każdemu programiście aplikacji internetowych. Dzięki temu możliwe jest natychmiastowe rozpoczęcie prac nad tworzoną aplikacją, bez konieczności przyswajania nowej technologii i związanych z nią formatów.

Replikacja

CouchDB umożliwia dwukierunkową replikację w dowolnej konfiguracji instancji. Oznacza to, iż dowolna liczba serwerów może synchronizować się ze sobą, dążąc do osiągnięcia *ewentualnej spójności danych*. Użycie MVCC gwarantuje rozwiązywanie konfliktów pomiędzy węzłami w deterministyczny sposób, nawet w wypadku braku połączenia między nimi.

Przykładowym zastosowaniem replikacji może być utrzymywanie zapasowej bazy danych synchronizowanej w czasie rzeczywistym lub rozłożenie ruchu na kilka instancji operujących na tych samych danych.

Współbieżność

Baza danych została napisana w języku *Erlang*². Nie posiada żadnych blokad. Jest w stanie obsłużyć o wiele więcej równoczesnych zapytań, niż klasyczna baza relacyjna oparta o SQL.

3.2.5. Integracja z Django: *Couchdbkit*

Pomimo stosunkowo młodego wieku, CouchDB jest bardzo dobrze wspierany przez zewnętrzne rozwiązania. Istnieją biblioteki przeznaczone dla najpopularniejszych języków programowania oraz wybranych frameworków. Przykładem jest wykorzystany w niniejszej pracy *Couchdbkit* – napisany w języku Python framework dostępu do bazy CouchDB [20].

¹ Wykorzystanie tego typu nagłówek opisano szerzej w rozdziale 4.11, na stronie 65.

² Funkcyjny język programowania przystosowany do budowy oprogramowania rozproszonego i wielowątkowego.

Odpowiednio skonfigurowany Couchdbkit umożliwia pracę na bazie danych i jej dokumentach porównywalną z operacjami na zwykłym słowniku (`dict`) języka Python. Specyfika bazy danych zorientowanej na dokumenty umożliwia dwukierunkowe mapowanie słowników oraz dokumentów.

Dla zwiększenia czytelności *listingu 3.4* zawartość przykładowego słownika została sformatowana, jednak należy zauważyć, iż przedstawiony kod to właściwie jedynie sześć linii:

- 1 Utworzenie obiektu reprezentującego serwer znajdujący się pod wskazanym adresem (domyślna wartość parametru `uri` umieszczona na potrzeby przykładu)
- 2 Utworzenie nowej bazy danych na serwerze oraz reprezentującego ją obiektu
- 3 Zapisanie słownika w bazie danych.
- 8 Odczyt dokumentu z bazy, utworzenie reprezentującego go słownika.
- 9 Utworzenie nowego pola w istniejącym dokumencie.
- 10 Zapisanie zmodyfikowanego dokumentu w bazie.

Listing 3.4. Przykład użycia *Couchdbkit*

```
1 serwer = Server(uri='http://127.0.0.1:5984')
2 baza = serwer.create_db("test_database")
3 baza['identyfikator'] = {
4     'tekst': 'wartosc',
5     'lista': [1, 2, 3, 4],
6     'słownik': {'klucz': 'wartosc'}
7 }
8 dokument = baza['identyfikator']
9 dokument['nowe pole'] = 'tekst'
10 baza['identyfikator'] = dokument
```

Widoczna oszczędność składni jest charakterystyczna dla języka Python. Couchdbkit działa w tle jako dyskretny pośrednik pomiędzy aplikacją a bazą danych. Elastyczność duetu Python oraz CouchDB widać w wierszu 9, gdzie do istniejącego już dokumentu dodano nowe pole.

Nie jest to jednak koniec udogodnień, jakie niesie ze sobą Couchdbkit. Dzięki modułowi `couchdbkit.ext.django` programista może w wygodny sposób wymienić domyślny ORM Django na rozwiązanie oparte o Couchdbkit bez konieczności dopisywania brakującej logiki biznesowej wykorzystywanej przez framework (ang. *drop-in replacement*).

Konfiguracja projektu Django sprowadza się do przypisania bazy danych do wybranej aplikacji Django w pliku `settings.py`:

```
COUCHDB_DATABASES = (
    ('mmda.artists', 'http://127.0.0.1:5984/mmda-artists'),
)
```

oraz dodania `couchdbkit.ext.django` do listy aktywnych aplikacji (również `settings.py`):

```
INSTALLED_APPS = (  
    'couchdbkit.ext.django',  
    'mmda.artists',  
)
```

Powyższy przykład przypisuje `mmda-artists` jako domyślną bazę danych dla aplikacji `artists` projektu `mmda`.

Aby utworzyć zdefiniowane bazy danych należy wydać polecenie:

```
python manage.py syncdb
```

Należy przy tym upewnić się, iż katalog wybranej aplikacji Django zawiera podkatalog `_design`, który przechowuje tzw. *design documents*. Są to specjalne dokumenty bazy CouchDB zawierające między innymi definicje widoków statycznych. Zostaną omówione w dalszej części pracy.

3.2.6. Modele dokumentów

Metodyka pracy z bazą danych zorientowaną na dokumenty oraz architektura przykładowej aplikacji opierają się na dynamicznym zestawie pól. Dla wygody programista może jednak zdefiniować modele reprezentujące wybraną grupę dokumentów o wspólnych właściwościach.

Przykładem tego typu modelu dokumentu w niniejszej pracy jest:

CachedArtist

Reprezentuje pojedynczego *artystę*: osobę, zespół lub ich kolaborację. Identyfikatorem tego typu dokumentów jest MBID¹.

Każdy artysta może posiadać inny zestaw atrybutów, takich jak:

- komentarz, lista aliasów,
- lista podobnych wykonawców,
- przynależność do zespołu / lista członków,
- ..i tym podobnych. W związku z tym deklaracja dedykowanych pól jest zbędna – zostaną dodane w chwili, gdy będą dostępne.

Wspólny dla wszystkich dokumentów tego typu jest słownik `cache_state`, służący do przechowywania informacji o chwili pobrania wybranych informacji z zewnętrznych źródeł. Ponieważ każdy `CachedArtist` je posiada (tak jak i linki zewnętrzne `urls`) dodanie tego typu pól do modelu likwiduje konieczność każdorazowej, ręcznej inicjalizacji pustego słownika.

Widoczny na *listingu 3.5* fragment pliku `models.py` przedstawia definicję klasy `CachedArtist` dziedziczącej po klasie `Document`. Technika jest analogiczna do tej sto-

¹ Odmiana UUID (ang. *Universally Unique Identifier*) używana przez serwis MusicBrainz (omówiony w rozdziale 4.2, strona 48).

sowanej przez domyślny ORM Django, gdzie klasa definiująca model dziedziczy po `Model`.

Listing 3.5. Przykładowy model dokumentu Django/Couchdbkit

```

1 from couchdbkit.ext.django.schema import Document, DictProperty
2 from mmda.engine.cache import CachedDocument
3
4 class CachedArtist(Document, CachedDocument):
5     urls = DictProperty(default={})
6     cache_state = DictProperty(default={})

```

Dodatkowe dziedziczenie po `CachedDocument` nie jest związane z omawianym zagadnieniem: umożliwia nadanie obiektom reprezentującym dokumenty dodatkowych atrybutów specyficznych dla tworzonej aplikacji.

`Document` jest cieniem opakowaniem użytej w *listingu 3.4* klasy `couchdbkit.schema.base.Document`. Dodaje on funkcje związane z Django:

- pracę z domyślną bazą zdefiniowaną w `settings.py`,
- wsparcie dynamicznie generowanych formularzy¹,
- obowiązkowe pole `doc_type` zawierające nazwę klasy bazowej dokumentu.

Praca z modelem dokumentu likwiduje potrzebę każdorazowej definicji obiektu reprezentującego bazę danych (*Listing 3.6*). Metoda `get_or_create` zawsze zwraca gotowy do pracy dokument. Jeśli nie istniał – zostaje utworzony. Programista może również użyć metody `get`, która w wypadku braku dokumentu o wskazanym identyfikatorze zwróci wyjątek `ResourceNotFound`.

Listing 3.6. Praca z modelem dokumentu Django/Couchdbkit

```

1 artist = CachedArtist.get_or_create(mbid)
2 if 'name' not in artist:
3     artist.name = 'Tool'
4     artist['artist_type'] = 'Group'
5     artist.dates = {'from': '1999'}
6 ...
7     artist.save()

```

Oparty na modelu dokumentu obiekt udostępnia dwie metody operacji na dynamicznych atrybutach: za pomocą składni opartej o kropkę (wiersz 3) lub klasycznej metody dostępu do pól słownika (wiersz 4). Wybór jednej z nich zależy od indywidualnych preferencji programisty. Obiekt zachowuje się jak klasyczny słownik, więc obsługuje większość dotyczących ich operacji. Przykład: w wierszu 2 sprawdzono czy

¹ Zagadnienie pominięte w niniejszej pracy, gdyż procedura nie różni się od tej w domyślnym ORM Django.

pole o wskazanej nazwie znajduje się w obiekcie `artist`. W identyczny sposób programista może sprawdzić czy słownik zawiera wskazany klucz. Wiersz 7 zapisuje w bazie danych uaktualniony dokument reprezentowany przez obiekt `artist`.

Listing 3.7 prezentuje alternatywną metodę tworzenia dokumentów. Istniejący, surowy słownik języka Python (wiersz 1) zostaje opakowany przez model `CachedArtist` (wiersz 2). Należy zauważyć, iż tego typu obiekt w chwili utworzenia nie posiada swojej reprezentacji w bazie danych. W chwili zapisu do bazy zostanie mu przypisany losowy identyfikator. Aby ułatwić jego odnalezienie w przyszłości, przypisano mu identyfikator `mbid` (wiersz 3).

Listing 3.7. Konwersja słownika na dokument CouchDB

```
1 | słownik = {'name': 'Tool', 'artist_type': 'Group' }
2 | artist = CachedArtist.wrap(słownik)
3 | artist._id = mbid
4 | artist.save()
```

Obie przedstawione metody dają programiście stosunkowo dużą elastyczność i mogą być stosowane zamiennie. Należy pamiętać, iż identyfikator (ciąg znaków, w powyższych listingach reprezentowany przez zmienną `mbid`) musi być unikalny w przestrzeni identyfikatorów pojedynczej bazy danych. Ewentualny konflikt powoduje wywołanie wyjątku `ResourceConflict`. W tworzonej aplikacji `mbid` jest zawsze unikalnym (dla każdego artysty) ciągiem znaków. Szczegółowe informacje dotyczące używanych w niej identyfikatorów znajdują się w rozdziale 5.3.

3.2.7. Widoki CouchDB w Django

Operacje na dokumentach bazy danych zaspokajają większość potrzeb przeciętnej aplikacji internetowej. Odpowiednio zaprojektowany model danych umożliwia bezpośrednią pracę na obiektach reprezentujących wybrany typ dokumentu bez konieczności dopisywania dodatkowej logiki biznesowej.

Zdarza się jednak, iż specyfika wykonywanej aplikacji wymusza filtrowanie, agregację, zestawianie danych z większej liczby dokumentów. Do tego typu zastosowań służą omówione wcześniej *widoki* bazy CouchDB. Biblioteka `Couchdbkit` udostępnia wygodny mechanizm tworzenia, synchronizacji oraz wykorzystania widoków z poziomu aplikacji Django.

Skonfigurowana do pracy z `Couchdbkit` aplikacja zawiera katalog `_design`. Umieszczone w nim *design documents* są zapisywane w bazie poprzez wydanie polecenia:

```
python manage.py syncdb
```

Dzięki temu mechanizmowi programista może edytować pliki definiujące zachowanie widoków oraz innych aspektów bazy danych w ramach plików należących do projektu Django. Technika ta znacznie ułatwia aktualizację aplikacji oraz późniejsze wdrożenie.

W przykładowej aplikacji artysta posiada dedykowaną stronę zawierającą (jeśli dostępne): notatkę biograficzną, dyskografię, powiązane adresy internetowe. Informacje o publikacjach muzycznych znajdują się w tej samej bazie danych co dokumenty reprezentujące artystów. Są to dokumenty oparte o model `CachedReleaseGroup`. Wszystkie zawierają obowiązkowe pole `artist_mbid` określające identyfikator artysty, do którego należy dana publikacja, pole `primary` zawierające identyfikator podstawowej wersji oraz opcjonalną datę jej wydania.

Problem 1

W jaki sposób pobrać z bazy danych dokumenty `CachedReleaseGroups` związane z danym artystą?

Problem 2

Dokument tego typu może zawierać więcej niż jedną wersję danego (na przykład) albumu. Każda z wersji zawiera własną listę utworów oraz informacje o wydaniu, spis osób zaangażowanych i tym podobne.

Czy konieczne jest pobieranie całych dokumentów, skoro wykorzystane w widoku będą jedynie: identyfikator, tytuł oraz data pierwszego wydania?

Co w wypadku, gdy informacja o dacie wydania nie jest dostępna?

Rozwiązaniem obu problemów jest utworzenie widoku bazy danych, który zwróci uporządkowaną listę par klucz-wartość zawierających pożądane informacje.

W ramach przykładu utworzono plik `_design/views/release_groups/map.js` zawierający deklarację funkcji `Map` przedstawionej w *listingu 3.8*.

Listing 3.8. Funkcja `Map` definiująca prosty widok CouchDB

```
1 function(doc) {
2   if (doc.doc_type == "CachedReleaseGroup" && doc.primary) {
3     if (doc.primary[1] == null) {
4       emit(doc.artist_mbid, {"title": doc.title,
5                             "release_type": doc.release_type,
6                             "mbid": doc.primary[0], "year": null});
7     } else {
8       emit(doc.artist_mbid, {"title": doc.title,
9                             "release_type": doc.release_type,
10                            "mbid": doc.primary[0],
11                            "year": doc.primary[1].substr(0,4),
12                            "date": doc.primary[1]});
13     }
14   }
15 }
```

Architektura CouchDB umożliwia pisanie widoków w dowolnym języku, jednak dla zachowania przejrzystości pracy autor postanowił opisać współpracę z bazą danych

o domyślnych ustawieniach. W związku z tym definicje widoków są pisane w języku JavaScript.

Funkcja Map iteruje po wszystkich dokumentach w bazie. Jeżeli dokument jest oparty o model `CachedReleaseGroup` oraz posiada pole `primary` (wiersz 2) sprawdza czy pole nie zawiera daty wydania (wiersz 3). Jeżeli data nie jest dostępna, funkcja zwraca słownik zawierający pole `year` o wartości `null` (zostanie wykorzystany przy sortowaniu). Jeżeli data jest dostępna, zwracany słownik w polach `year` oraz `date` zawiera wartości pobrane z dokumentu. Duplikacja danych przez pole `year` jest celowa: każdorazowe pobieranie daty z bazy i obrabianie jej przy każdym wyświetleniu strony nie wiąże się z dużym opóźnieniem, jednak o wiele rozsądniej jest wyliczyć taką wartość raz i przechowywać ją w statycznym widoku.

Po wydaniu polecenia `manage.py syncdb` wynik działania widoku (*Listing 3.9*) można zaobserwować pod adresem:

`http://127.0.0.1:5984/mmda-artists/_design/artists/_view/release_groups`

Listing 3.9. Fragment widoku CouchDB w formacie JSON

```

1 {"total_rows":4795,"offset":0,"rows":[
2 {"id":"ecfd0a01-57c8-3949-990a-1a950942e0e0",
3  "key":"66fc5bf8-daa4-4241-b378-9bc9077939d2",
4  "value":{"title":"10,000 Days",
5           "release_type":"Album",
6           "mbid":"58a5c0c1-60aa-4e56-8fd5-4ee4777c9d47",
7           "year":"2006",
8           "date":"2006-04-28"}
9 },
10 {"id":"6891a28c-5865-36e2-9e5f-c9fac1d3595f",
11  "key":"87c5dedd-371d-4a53-9f7f-80522fb7f3cb",
12  "value":{"title":"Debut",
13           "release_type":"Album",
14           "mbid":"b8b4c8d8-9336-4df6-9a29-1684a94f99e4",
15           "year":"1993",
16           "date":"1993-07-12"}
17 },
18 {"id":"b04aaf5d-e155-3b60-a5fe-370fba5660c6",
19  "key":"87c5dedd-371d-4a53-9f7f-80522fb7f3cb",
20  "value":{"title":"Hunter EP",
21           "release_type":"Remix",
22           "mbid":"1b53595d-0ee9-48b2-8fd2-05e89e5f6b03",
23           "year":null}
24 },
25 ...
26 ]}]

```


Surowy widok w formacie JSON widoczny w *listingu 3.9* to prosta lista słowników. Pole `id` zawiera identyfikator dokumentu, z którego pochodzi wynik. Jest zawsze obecne w widoku, niezależnie od zdefiniowanej przez programistę funkcji mapującej.

Funkcja `emit` z *listingu 3.8* otrzymuje zawsze dwa argumenty. Przekładają się one na pola widoku: pierwszy z nich to `key`, w którym w omawianym przykładzie umieszczono identyfikator dokumentu reprezentującego artystę. Pole `value` zawiera słownik z pożądanymi danymi dotyczącymi publikacji muzycznej.

W tym miejscu należy zauważyć, iż utworzona przez widok lista wyników jest posortowana leksykograficznie według zawartości pola `key`. Ponieważ raz utworzony widok statyczny jest przechowywany i aktualizowany w bazie, możliwe jest szybkie filtrowanie wyników po wartości lub przedziale wartości tego pola. Należy uwzględnić tę własność widoków CouchDB przy doborze argumentów funkcji `emit`.

W omawianym przykładzie poszukiwane są informacje o publikacjach konkretnego artysty, więc wymagane są wyniki z konkretną wartością pola `key`. Filtrowanie sprowadza się do dodania parametru `?key=` na końcu adresu widoku, na przykład:

```
http://127.0.0.1:5984/mmda-artists/_design/artists/_view/release_groups?
key="66fc5bf8-daa4-4241-b378-9bc9077939d2"
```

Tak sformułowane zapytanie do widoku zwraca jedynie wyniki dotyczące publikacji wybranego autora. Utworzony widok rozwiązał wymienione problemy i spełnia postawione przed nim wymagania.

Każdy widok bazy obsługuje wiele innych, predefiniowanych parametrów. Do najczęściej używanych należą:

`startkey` / `endkey` – umożliwiają określenie leksykograficznego zakresu kluczy,

`descending` – odwrócenie kolejności zwracanych wyników,

`limit` – określenie maksymalnej ilości wyników,

`include_docs` – dołączenie dokumentów o identyfikatorach określonych przez pole `id` do wyników. Jest to wydajny sposób pobierania pełnych dokumentów z poziomu widoku bez konieczności kopiowania ich zawartości do pola `value`.

Listing 3.10 przedstawia kod, który pobiera elementy omawianego wcześniej widoku filtrowane po kluczu `key` równym `mbid` (wiersze 1-2). Następnie za pomocą tzw. *listy składanej* tworzona jest lista zawierająca pożądanane słowniki znajdujące się w polach `value`.

Listing 3.10. Dostęp do danych widoku za pomocą *Couchdbkit*

```
1 view = get_db('artists').view('artists/release_groups',
2                               key=mbid)
3 primary_releases = [match['value'] for match in view.all()]
```

Couchdbkit umożliwia obsługę widoków za pomocą intuicyjnego API w języku Python. W zależności od tego czy wynikiem działania widoku są pełne dokumenty, czy

też jedynie wybrane pola, programista może automatycznie opakować wyniki w obiekty modelu danych, bądź operować na zwykłych strukturach danych takich jak słowniki oraz listy.

Należy zauważyć, iż metoda `view` zawiera ścieżkę widoku pozbawioną członu `/view/`. Jest to zabieg czysto kosmetyczny zwiększający czytelność kodu (nazwa metody jednoznacznie już określa czym jest `release_groups`).

Alternatywnym scenariuszem pracy z widokiem jest pobranie pełnych dokumentów. Przykład: zamiast wybranych pól określonych jako drugi argument funkcji `emit`, znajdujących się w polu `value`, programista może wykorzystać parametr `include_docs`, aby dołączyć pełne dokumenty określone identyfikatorami `id` do listy wyników widoku.

Przyjętą konwencją w takim wypadku jest określenie drugiego parametru funkcji `emit` jako `null`, gdyż pole `value` nie będzie używane:

```
emit(doc.artist_mbid,null);
```

Uwaga: częstym błędem popełnianym przez programistów rozpoczynających pracę z CouchDB jest umieszczanie kopii dokumentu jako drugiego argumentu funkcji `emit`. Wynikiem takiego działania jest zbędne duplikowanie danych (widoki statyczne są przechowywane w B-drzewach tak jak zwykłe dokumenty) oraz ogólne spowalnianie pracy widoku.

Pobierając pełne dokumenty za pomocą metody `view` klasy `CachedReleaseGroup` programista ma pewność, iż każdy wynik jest automatycznie opakowywany jako instancja danego modelu (*Listing 3.11*).

Listing 3.11. Pobieranie pełnych dokumentów za pomocą *Couchdbkit*

```
1 | view = CachedReleaseGroup.view('artists/release_groups',  
2 |                               key=mbid, include_docs=True)  
3 | release_groups = view.all()
```

Podsumowując, widoki są wygodnym kanałem poboru danych z bazy. Charakteryzuje je duża elastyczność: mogą zwracać zarówno pełne dokumenty, jak i jedynie wybrane pola lub zestawy danych wygenerowane na podstawie pól dokumentów źródłowych (przykład: pole `year`). Nic nie stoi na przeszkodzie, aby funkcja mapująca zwracała wiele wyników na podstawie jednego dokumentu.

Moduł biblioteki *Couchdbkit* integrujący CouchDB z Django jest dojrzały. Pokrywa różnorakie scenariusze pracy z bazą danych, umożliwiając płynne przejście z domyślnego ORM na bardziej liberalną pracę z dokumentami.

3.3. System szablonów

3.3.1. Wstęp

Większość aplikacji internetowych oprócz logiki biznesowej posiada warstwę prezentacji danych. Aplikacja będąca przedmiotem tej pracy wykorzystuje wbudowany we framework Django system szablonów. Stanowi on wygodne narzędzie wspomagające tworzenie dynamicznych dokumentów HTML. Cechą charakterystyczną tego systemu jest – zgodnie z paradygmatem MVC – oddzielenie logiki biznesowej od warstwy prezentacji danych.

Konfiguracja projektu Django sprowadza się do określenia w pliku `settings.py` ścieżki do katalogu zawierającego wykorzystywane szablony. Na przykład:

```
TEMPLATE_DIRS = (  
    '/home/user/mmda/templates',  
)
```

Szablon jest zwykłym plikiem tekstowym zawierającym kod HTML oraz specjalne elementy: *zmienne* i *tagi*. Generowanie dokumentu na podstawie szablonu polega na podstawieniu wartości w miejsce nazw zmiennych szablonu oraz wykonaniu kodu związanego z działaniem tagów. Cała procedura jest wykonywana automatycznie przez framework. W zależności od potrzeb programista może ingerować w ten proces, modyfikować zmienne za pomocą *filtrów*, lub definiować własne filtry oraz tagi.

System szablonów Django nie obsługuje pełnej składni języka Python. Jest on o wiele bardziej restrykcyjny, niż (na przykład) system obecny we frameworku Ruby On Rails, który umożliwia zagnieżdżanie klasycznego kodu Ruby wewnątrz szablonów. Jest to celowe założenie projektowe Django, skłaniające programistę do oddzielenia logiki biznesowej od warstwy prezentacji.

Metodyka pracy z szablonami Django polega na dodawaniu brakujących funkcji za pomocą filtrów oraz tagów.

3.3.2. Zmienne

Bloki oznaczające zmienne oddzielane są za pomocą notacji:

```
{{ nazwa_zmiennej }}
```

Silnik renderujący umieszcza w dokumencie odpowiadającą zmiennej sekwencję znaków. Nazwa zmiennej może być przekazana z widoku Django (`views.py`) lub utworzona lokalnie (np. przy użyciu tagu `for`). W wypadku gdy zmienna o podanej nazwie nie została zainicjalizowana wstawiany jest ciąg `''`, czyli pusty string.

3.3.3. Filtry

Django posiada bogatą bibliotekę wbudowanych filtrów zaspokajających częste potrzeby programistów. Każda zmienna może zostać przetworzona przez dowolną ilość filtrów. Zmienna przetwarzana przez filtry oznaczana jest jako sekwencja filtrów oddzielonych znakiem kreski pionowej (ang. *pipe*):

```
{{ nazwa_zmiennej|filtr1|filtr2 }}
```

Przykład 3.1: Filtr `date` służący do formatowania zmiennych zawierających datę. Zakładając, iż `zmienna_z_data` jest obiektem typu `datetime` fragment kodu:

```
{{ zmienna_z_data|date }}
```

może wygenerować ciąg znaków: `'April 13, 2010'`.

Przykład 3.2: Filtr oprócz samej zmiennej może przyjmować zadeklarowane po dwukropku argumenty. Kod `{{ zmienna_z_data|date:"Y" }}` zwróci ciąg `'2010'`.

Otwarte API umożliwia pisanie własnych filtrów. Pliki z definicją filtrów są umieszczane w podkatalogu `templatetags` (przykład: `mmda/artists/templatetags/release_helpers.py`)

Przykład 3.3: Wbudowany filtr `dictsort` służy do sortowania listy słowników według wskazanego klucza. Implementacja filtru podczas tworzenia listy wyników pomija słowniki, które go nie posiadają. Pożądanym w tworzonej aplikacji zachowaniem jest umieszczenie tego typu elementów na końcu listy wyników. Rozwiązaniem problemu jest utworzenie nowego filtru realizującego zamierzony sposób sortowania.

Listing 3.12 zawiera zmodyfikowany filtr `dictsort`. Podczas tworzenia listy składowanej (wiersz 3) elementom nieposiadającym wskazanych przez zmienną `key` wartości przypisany jest pusty ciąg znaków (`''`). Dzięki temu tego typu obiekty znajdują się na końcu listy wyników. Dekorator `@register.filter` nadaje funkcji `inclusivedictsort` atrybut filtru.

Listing 3.12. Implementacja przykładowego filtru szablonu Django

```

1 @register.filter
2 def inclusivedictsort(list, key):
3     decorated = [(i[key] if key in i else '', i)
4                  for i in list]
5     decorated.sort()
6     decorated.reverse()
7     return [item[1] for item in decorated]
```

Aby móc skorzystać z utworzonych filtrów, konieczne jest zaimportowanie zawierającej je biblioteki. Służy do tego tag `load`:

```
{% load release_helpers %}
```

Użycie zaimportowanego filtru w szablonie jest analogiczne do filtrów wbudowanych:

```
lista_slownikow|inclusivedictsort:"klucz"
```

Filtry mogą operować na zmiennych zarówno w bloku `'{{ }}'` jak i wewnątrz tagów.

3.3.4. Tagi

Tagi są złożonym konstruktem systemu szablonów Django i mogą pełnić wiele funkcji:

- kontrola przepływ wykonania kodu szablonu,
- przetwarzanie zmiennych wykraczające poza możliwości filtrów,
- przyspieszenie pracy z szablonem poprzez realizowanie popularnych czynności.

Przykład 3.4: Jak posiadając listę zawierającą nazwy oraz identyfikatory artystów wygenerować linki do poświęconych nim podstron?

Listing 3.13 demonstruje działanie wybranych wbudowanych tagów. Wiersze 1 oraz 6 zawierają deklarację rozpoczęcia i zakończenia kontrolującego iterację bloku `for`. Tag `url` służy do generowania adresów URL na podstawie nazwy widoku (zdefiniowanej w pliku `urls.py` projektu lub aplikacji) oraz argumentów przekazywanych do funkcji obsługującej widok (definiowanej w `views.py`). W omawianym przykładzie przedostatni argument tagu `url` jest przetwarzany za pomocą filtru `slugify`, który konwertuje nazwę artysty do formatu zgodnego z RFC 2396 [21].

Listing 3.13. Przykład użycia wbudowanych tagów `url` oraz `for`

```
1 {% for artist in current_artist.similar %}
2 <a href="{% url show-artist artist.name|slugify artist.mbid %}">
3     {{ artist.name }}</a>
4 {% empty %}
5     brak
6 {% endfor %}
```

Wygenerowany przez szablon z *listingu 3.13* fragment kodu HTML dla przykładowego obiektu `current_artist` o identyfikatorze `66fc5bf8-daa4-4241-b378-9bc9077939d2` przedstawia *listing 3.14*.

Listing 3.14. Wygenerowany przez szablon przykładowy kod HTML

```
1 <a href="/artist/apc/078a9376-3c04-4280-b7d7-b20e158f345d/">
2     A Perfect Circle</a>
3 <a href="/artist/puscifer/6e4277d8-7b64-44a3-b823-13bd15114e22/">
4     Puscifer</a>
5 <a href="/artist/indukti/216de703-21b9-4396-a01f-7927a754c4a7/">
6     Indukti</a>
7 ...
```

Przykład 3.5: W jaki sposób wydzielić pewną część logiki biznesowej wykorzystywanej przy tworzeniu dokumentu HTML, aby można było się do niej wielokrotnie odwoływać? Programista może tworzyć własne tagi.

W przykładowej aplikacji zarówno artysta, jak i publikacja muzyczna mogą posiadać pole **abstract** zawierające krótką notkę/informację (ang. *abstract*). Wydzielenie fragmentu szablonu do oddzielnego pliku i zagnieżdżanie go przy użyciu tagu **include** nie jest wydajnym rozwiązaniem, gdyż wiązałoby się z każdorazowym sprawdzaniem, która nazwa zmiennej jest dostępna¹ lub obniżającą czytelność kodu unifikacją nazw w ramach wybranych szablonów.

Zgodnie z zasadą DRY² pożądanym rozwiązaniem jest utworzenie tagu (*Listing 3.15*), który wygeneruje kod HTML na podstawie obiektu przekazanego do niego jako parametr. Dzięki temu osiągnięty zostanie poziom izolacji umożliwiający wykorzystanie tego samego kodu zarówno na stronach artystów, jak i publikacji.

Listing 3.15. Tag `abstract_for` – implementacja

```
1 @register.inclusion_tag('artists/tags/abstract.html')
2 def abstract_for(entity):
3     return {'entity': entity}
```

Przykład użycia tagu z *listingu 3.15* w szablonie:

```
{% abstract_for artist %}
```

Mechanizm kompilacji Zgodnie z architekturą szablonów Django, tag `abstract_for` reprezentowany jest w drzewie dokumentu przez obiekt klasy dziedziczącej po `Node`. Po załadowaniu wszystkich zależności generowanego dokumentu rozpoczyna się proces kompilacji. Zostaje wywołana metoda `render` każdej istniejącej instancji `Node`. W wypadku omawianego tagu utworzonego przy użyciu dekoratora `inclusion_tag`, szablon `artists/tags/abstract.html` zostanie skompilowany z uwzględnieniem zawartości obiektu `entity` i włączony do kodu wygenerowanego dokumentu HTML.

Typy tagów Oprócz wymienionego już (ułatwiającego tworzenie izolowanych podszablonów) dekoratora `inclusion_tag`, dostępny jest `simple_tag` używany w wypadku, gdy użycie zewnętrznego pliku nie jest wymagane. Tagi tworzone za pomocą tych dekoratorów pokrywają większość potrzeb programisty. W pozostałych przypadkach otwarte API umożliwia tworzenie własnych implementacji klas dziedziczących po `Node`, dając nieograniczoną elastyczność operacji na kontekście/przestrzeni nazw szablonu oraz strukturze generowanego dokumentu.

¹ Szablony zagnieżdżone tagiem `include` mają dostęp do przestrzeni nazw przekazanej z `views.py` (w przeciwieństwie do np. Ruby On Rails, gdzie tzw. *partial* posiada własną przestrzeń nazw).

² ang. *Don't Repeat Yourself*

3.3.5. Dziedziczenie szablonów

Specyfika tworzenia warstwy prezentacji aplikacji internetowych wymaga wielokrotnego wykorzystywania tych samych fragmentów dokumentu HTML. Przykładem mogą być powtarzające się na każdej podstronie menu, elementy nawigacyjne, pola wyszukiwania, stopka. System szablonów Django udostępnia wygodny mechanizm dziedziczenia szablonów za pomocą tagów **block** oraz **extends**.

Widoczny na *listingu 3.16* szablon bazowy **base.html** posiada zdefiniowane bloki **title**, **meta**, **header**, **content** oraz **footer**. Każdy blok może być pusty lub zawierać w sobie dowolne elementy HTML oraz znaczniki szablonu (przykład: wiersz 13). Z punktu widzenia dokumentu bazowego, znaczniki określające bloki są ignorowane przy jego kompilacji. Są one metainformacją wykorzystywaną przez szablony potomne.

Listing 3.16. Dziedziczenie szablonów: szablon bazowy **base.html**

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>{% block title %}{% endblock %}</title>
5     <meta charset="utf-8" />
6     {% block meta %}{% endblock %}
7   </head>
8   <body>
9     <header>{% block header %}{% endblock %}</header>
10    <nav>{% block navigation %}{% endblock %}</nav>
11    <article>
12      {% block content %}
13      Nothing here ;-(
14      {% endblock %}
15    </article>
16    <footer>{% block footer %}{% endblock %}</footer>
17  </body>
18 </html>
```

Każdy dokument potomny (przykład – *Listing 3.17*) zawiera tag **extends** przyjmujący jako parametr ścieżkę do pliku szablonu bazowego oraz następujące po nim tagi **block**. Kompilacja szablonu potomnego polega na zastąpieniu zawartości bloków szablonu bazowego, przez ich odpowiedniki z szablonu potomnego. W wypadku gdy dany blok nie został zdefiniowany w szablonie potomnym, proces kompilacji wykorzysta wersję z dokumentu bazowego.

Mechanizm dziedziczenia szablonów umożliwia wygodną i precyzyjną modyfikację wybranych sekcji warstwy prezentacji aplikacji internetowej.

Listing 3.17. Dziedziczenie szablonów: szablon potomny

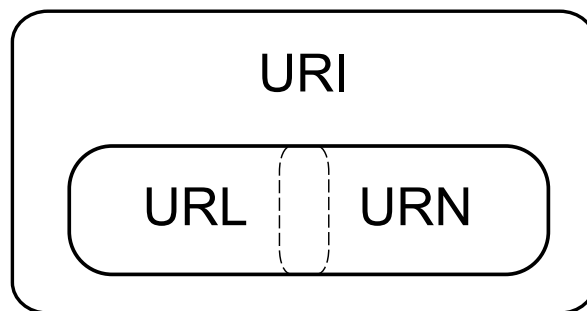
```
1 {% extends "base.html" %}
2 {% block title %}Child Page{% endblock %}
3 {% block header %}<h1>Page Header</h1>{% endblock %}
4 {% block content %}
5     Nobody Expects The Spanish Inquisition!
6 {% endblock %}
```

3.4. Obsługa zapytań HTTP

3.4.1. Projektowanie URL

Funkcjonowanie większości aplikacji opartych o protokół HTTP polega na dostępie do zasobów określonych przy pomocy URL¹.

URL jest odmianą URI². Jest identyfikatorem charakteryzującym dany zasób sieci Internet. W przeciwieństwie do URN³ zawiera kompletną informację o lokalizacji oraz protokole dostępu do zasobu [21]. Zależność między URI, URL oraz URN przedstawia rys. 3.1.



Rysunek 3.1. Relacje między URI, a URL i URN

Źródło: http://en.wikipedia.org/wiki/File:URI_Venn_Diagram.svg

W Django programista ma pełną kontrolę nad architekturą URL tworzonej aplikacji. Jednym z pierwszych etapów rozwoju nowego projektu jest określenie przestrzeni nazw adresów URL obsługiwanych przez implementowaną później logikę biznesową.

Przykład 3.6: Tworzona aplikacja powinna posiadać czytelne adresy URL jasno określające typ oraz identyfikator wywoływanego zasobu. Strony poświęcone danemu artyście powinny posiadać wspólny identyfikator oraz strukturę. (W poniższych przykładach dla zwiększenia czytelności pominięto początkową, niezmienną w ramach projektu część adresów URL.)

¹ ang. *Uniform Resource Locator*

² ang. *Uniform Resource Identifier*

³ ang. *Uniform Resource Name*

Pożądanym jest, aby strony poświęcone wykonawcy *Tool* o identyfikatorze *66fc5bf8-daa4-4241-b378-9bc9077939d2* posiadały następujące ścieżki zasobów serwera:

```
/artist/tool/66fc5bf8-daa4-4241-b378-9bc9077939d2/  
/artist/tool/pictures/66fc5bf8-daa4-4241-b378-9bc9077939d2/  
/artist/tool/videos/66fc5bf8-daa4-4241-b378-9bc9077939d2/  
/artist/tool/news/66fc5bf8-daa4-4241-b378-9bc9077939d2/  
/artist/tool/news/66fc5bf8-daa4-4241-b378-9bc9077939d2/feed/
```

Natomiast strona zawierająca informacje o publikacji muzycznej danego artysty:

```
/artist/tool/release/lateralus/579278d5-75dc-4d2f-a5f3-6cc86f6c510e/
```

Identyfikator wskazuje jednoznacznie którego artysty lub publikacji dotyczy zapytanie. Segmenty */tool/* oraz */lateralus/* pełnią funkcję dekoracyjną, niezwiązaną z logiką biznesową aplikacji. Nadają zasobom czytelną dla ludzkiego odbiorcy formę: URL zawiera w sobie informację, iż prowadzi do – na przykład – strony zawierającej informacje o publikacji *Lateralus* należącej do artysty o nazwie *Tool*.

Kolejną zaletą umieszczenia tych dodatkowych informacji w URL jest zwiększenie pozycji danej podstrony w wynikach wyszukiwarek internetowych poświęconych danym słowom kluczowym. Budowanie przyjaznych URL jest jedną z podstawowych technik SEO¹.

Dla zwiększenia złożoności projektu założono, iż zasób nieposiadający MBID powinien prowadzić do wyniku wyszukiwania. Innymi słowy, adres:

```
/artist/tool/
```

powinien prowadzić do zasobu:

```
/search/artist/tool/a5dcc2f26b8ae66a6865448dadbf8987b518a5fd/
```

Aby w sposób czytelny oddzielić mechanizmy wyszukiwania od stron artystów oraz umożliwić tworzenie permanentnych linków, adres wyników wyszukiwania ma być poprzedzony członem */search/*.

Identyfikator *'a5dcc2f26b8ae66a6865448dadbf8987b518a5fd'* będzie wynikiem działania funkcji skrótu SHA1 na ciągu znaków będącym skonkatenowanym typem oraz przedmiotem wyszukiwania. Należy zaznaczyć, że jest to identyfikator dokumentu **CachedSearchResult** zawierającego zbuforowany wynik wyszukiwania. Dzięki takiej architekturze linkowanie do wyniku wyszukiwania spowoduje wyświetlenie zawartości już istniejącego dokumentu, bez każdorazowego odpytywania zewnętrznych API.

3.4.2. Implementacja schematu URL

Konfiguracja URL każdego projektu Django znajduje się w pliku *urls.py*. Dodatkowo możliwe jest wydzielenie części konfiguracji do plików *urls.py* znajdujących się

¹ ang. *Search Engine Optimization*, szeroko pojęte działania promujące serwisy internetowe w wyszukiwarkach internetowych.

w podkatalogach aplikacji należących do projektu. Dla zachowania przejrzystości opisana niżej przykładowa implementacja zawiera jedynie wybrane definicje oraz zakłada umieszczenie jej w jednym pliku.

Umieszczony w *listingu 3.18* fragment pliku `urls.py` zawiera najczęściej spotykane metody budowania interfejsu URL. Wiersze 9-14 zawierają definicje zasobu strony artysty oraz publikacji muzycznej. Blok `(?P<uri_artist>\S+)` przechwytuje ciąg znaków odpowiadający wyrażeniu regularnemu `\S+` i zapisuje go w zmiennej `uri_artist`. Pozostałe segmenty URL są przetwarzane i zapamiętywane w zmiennych w analogiczny sposób. Przechwycone zmienne są przekazywane jako argumenty do wskazanej funkcji widoku (na przykład `mmda.artists.views.show_release`), która odpowiada za wygenerowanie odpowiedzi HTTP.

Listing 3.18. Konfiguracja URL w Django: `urls.py`

```

1 from django.conf.urls.defaults import *
2 from mmda.engine.feeds import ArtistNewsFeed
3 from django.conf import settings
4
5 feeds = {
6     'artist': ArtistNewsFeed,
7 }
8 urlpatterns = patterns('',
9     url(r'^artist/(?P<uri_artist>\S+)/release/(?P<uri_release>\S+)/
10         (?P<mbid>\w{8}-\w{4}-\w{4}-\w{4}-\w{12})/$',
11         'mmda.artists.views.show_release', name='show-release'),
12     url(r'^artist/(?P<uri_artist>\S+)/
13         (?P<mbid>\w{8}-\w{4}-\w{4}-\w{4}-\w{12})/$',
14         'mmda.artists.views.show_artist', name='show-artist'),
15     (r'^(?P<url>.+)/feed/$', 'django.contrib.syndication.views.feed',
16         {'feed_dict': feeds}),
17     (r'^artist/(?P<query_string>[/]+)$',
18         'mmda.search.views.create_search_result', {'query_type': 'artist'}),
19     (r'^static/(?P<path>.*$)',
20         'django.views.static.serve',
21         {'document_root': settings.STATIC_DOC_ROOT, 'show_indexes': True}),
22     url(r'^search/(?P<query_type>\S+)/(?P<query_string>[/]+)/
23         (?P<query_id>\w{40})/$',
24         'mmda.search.views.show_search_result', name='show-search-result'),
25     ...
26 )

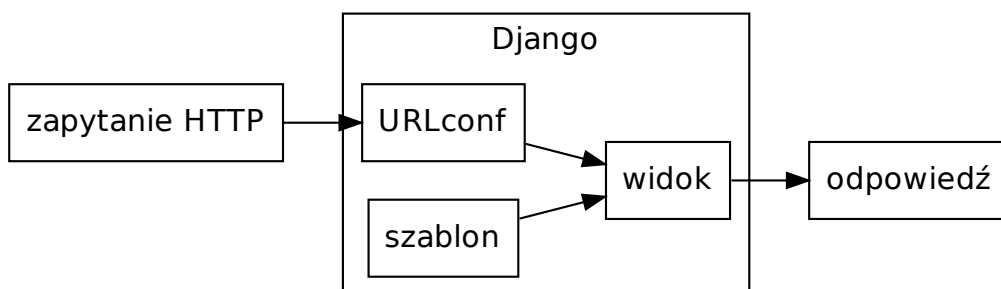
```

Definicja może mieć przypisaną nazwę (`name=`) która może być wykorzystana między innymi przez tag `url` podczas tworzenia szablonów aplikacji. W wypadku przypisania nazwy należy pamiętać, aby krótkę definiującą regułę URL (wiersz 15-21) zamienić na funkcję `url` (wiersz 9-14).

Niestandardową konstrukcją jest widoczna w wierszu 15 definicja URL kanału Atom. Wyrażenie regularne przechwytuje całą ścieżkę poprzedzającą segment `/feed/`, następnie przekazuje ją do wskazanego przez słownik `feeds` obiektu zawierającego logikę biznesową związaną z generowaniem dokumentu w formacie Atom. (Wersja rozwojowa Django (0.1.2) wprowadza składnię umożliwiającą unifikację oraz uproszczenie konstrukcji adresów kanałów Atom. Niniejsza praca oparta jest na stabilnej wersji 0.1.1, stąd nietypowa deklaracja URL.)

Wiersz 17-18 zawiera przykład przekazania statycznie wprowadzonej wartości zmiennej `query_type` do funkcji `create_search_result`. Dodatkowo pokazuje, iż URL posiadające ten sam początkowy segment `/artist/` mogą być obsługiwane przez dwie różne aplikacje projektu Django. Mechanizm tworzenia obsługi URL jest na tyle elastyczny, że obecna w wierszu 19-21 (dodana na czas rozwoju projektu) deklaracja adresu zasobów statycznych odwołuje się do (niebędącej częścią rozwijanego projektu) wbudowanej we framework funkcji `static.serve`.

Mechanizm przetwarzania wywołania danego adresu URL na odpowiedź aplikacji ilustruje *rys. 3.2*. Przez *URLconf* rozumiana jest konfiguracja zawarta w plikach `urls.py`.



Rysunek 3.2. Obsługa zapytań HTTP w Django

Źródło: Opracowanie własne

3.5. Formularze

Framework Django zawiera rozbudowane mechanizmy wspomagające pracę z formularzami. Klasyczny ORM umożliwia tworzenie formularzy HTML na podstawie modelu danych. Omawiany w niniejszej pracy Couchdbkit wspiera tę funkcję w identyczny sposób.

Z uwagi na specyfikę przykładowej aplikacji (agregacja treści z zewnętrznych serwisów) praca z formularzami sprowadza się do obsługi mechanizmu wyszukiwania oraz odświeżania danych artysty na żądanie. Omówiony zostanie pierwszy przypadek.

3.5.1. Przykładowa operacja POST

Wysłanie prostego formularza (*Listing 3.19*) metodą POST na adres `/search/` spowoduje wywołanie funkcji `mmda.search.views.create_search_result` z obiektem `request` jako pierwszym argumentem. Zawiera on komplet informacji dotyczących aktualnie przetwarzanego zapytania HTTP, w tym przesyłane przez klienta (użytkownika aplikacji) metodą POST dane.

Listing 3.19. Przykładowy formularz POST

```
1 <form action="/search/" method="post">
2     <label for="type">Search for</label>
3     <select name="type">
4         <option value="artist">Artist</option>
5         <option value="release">Release</option>
6         <option value="tag">Tag</option>
7     </select>
8     <label for="query">called</label>
9     <input type="text" name="query" value="">
10    <button type="submit">Search</button>
11 </form>
```

Przykładowa funkcja `create_search_result` (*Listing 3.20*) przedstawia metodę dostępu do pól przesłanych przy użyciu POST (wiersze 2-3). Atrybut `request.POST` jest słownikiem – zawierane przez niego pola mogą być odczytane za pomocą standardowej notacji.

Listing 3.20. Przykładowa funkcja przetwarzająca zapytanie POST

```
1 def create_search_result(request):
2     query_type = request.POST['type']
3     query_string = request.POST['query'].strip().lower()
4     query_id = get_basic_cached_search_result(query_type,
5                                               query_string)
6
7     return HttpResponseRedirect(reverse('show-search-result',
8     args=(slugify(query_type), slugify(query_string), query_id)))
```

Funkcja `get_basic_cached_search_result` jest odpowiedzialna za utworzenie (jeśli nie istnieje) wyniku wyszukiwania i zapisanie go w dokumencie o identyfikatorze przypisanym do zmiennej `query_id`. Logika biznesowa tej i innych funkcji zostanie omówiona w kolejnych rozdziałach.

Posiadając identyfikator wyniku wyszukiwania, funkcja `create_search_result` zwraca tymczasowe przekierowanie (HTTP 302) na adres wygenerowany przy pomocy funkcji `reverse` dla zdefiniowanego w `urls.py` schematu URL o nazwie `show-search-result`.

Zachowanie to zapobiega wielokrotnemu wysyłaniu formularza metodą POST w wypadku odświeżenia strony przez użytkownika, dodatkowo umożliwia tworzenie permanentnych linków dla każdego wyniku wyszukiwania.

Jest to ogólny schemat przetwarzania prostych zapytań POST w aplikacjach Django.

4. Wybrane API oraz biblioteki

4.1. Wstęp

4.1.1. Przyczyny powstania API

Aplikacje internetowe komunikują się użytkownikiem przy użyciu dynamicznie generowanych dokumentów HTML. Oprócz surowych informacji zawierają one również narzut w postaci elementów nawigacyjnych oraz kodu definiującego graficzny interfejs użytkownika.

Użytkownikiem serwisu internetowego mogą być również różnego rodzaju aplikacje. Ich interakcja ze stroną internetową wymaga zupełnie innego podejścia do zagadnień nawigacji, czy prezentacji danych. Technologie takie jak JavaScript oraz CSS służą oddzieleniu warstwy prezentacji od warstwy danych w dokumentach opartych o HTML.

Pomimo ich wykorzystania, aplikacje napotykają trudności podczas interpretacji stron przeznaczonych dla użytkowników. Są to:

- *utrudniona nawigacja*

Nawigacja większości stron zakłada obsługę graficznej reprezentacji dokumentu przy użyciu klawiatury oraz myszki. Często istotne elementy są wyróżnione za pomocą grafiki lub CSS. Spotykane są strony, które nawigację opierają w pełni o JavaScript lub technologię Adobe Flash, nie oferując alternatywy w „czystym” HTML.

- *brak metainformacji*

Oddzielenie materiałów reklamowych, bloku nawigacyjnego, stopki czy też nagłówka od głównej treści danej strony nie stanowi dla ludzkiego użytkownika problemu. Wystarczająca jest posiadana intuicja połączona z odpowiednio zaprojektowanym wyglądem strony.

Program komputerowy operuje na surowym HTML. Bez dodatkowych informacji nie jest w stanie oddzielić istotnych danych, od pozostałych elementów strony.

Semantycznie zbudowane dokumenty oparte o HTML5 oraz mikroformaty częściowo rozwiązują ten problem. Nie jest on jednak jedynym.

- *nadmiar zbędnych informacji*

Każda strona zawiera wcześniej wymieniony narzut w postaci nadmiarowego formatowania lub opisów przeznaczonych dla ludzkiego odbiorcy. Każdorazowe odwoływanie się do tego typu strony powoduje przesyłanie informacji, które są zupełnie zbędne.

- *niestabilna struktura dokumentu oraz URL*

Zdarza się, że wraz z nową wersją serwisu internetowego zmienia się wewnętrzna struktura dokumentów HTML. Często wprowadzona zostaje również nowa hierarchia URL. W takim wypadku wcześniej napisany parser HTML może przestać działać lub – co gorsza – zacząć zwracać błędne wyniki.

4.1.2. API aplikacji internetowej

Odpowiedzią na wymienione problemy jest utworzenie API, czyli spójnego, niezmiennego i udokumentowanego interfejsu komunikacji pomiędzy aplikacjami. W zależności od specyfiki serwisu API może mieć politykę otwartą – nieograniczony dostęp dla aplikacji zewnętrznych lub zamkniętą/mieszaną – dostęp do części/całości zasobów wymaga posiadania klucza, który może być wydawany nieodpłatnie lub sprzedawany wraz z licencją na dane pobrane przez API.

Wyróżnia się dwa najczęściej spotykane rodzaje API, określane przez pojęcia:

SOAP

Obecny w części systemów korporacyjnych¹ system komunikacji oparty o zapytania XML w formacie SOAP².

REST

API realizujące założenia REST stanowi alternatywę dla SOAP. Porzuca złożoną składnię na rzecz podstawowych operacji protokołu HTTP takich jak: POST, PUT, GET oraz DELETE na określonych przez URL zasobach [22].

Zalety API typu REST:

- przejrzysty i czytelny,
- łatwa dokumentacja,
- ułatwione tworzenie cache, buforowanie odpowiedzi (na poziomie protokołu HTTP),
- kompatybilny z SOAP (może być opakowany w zapytania SOAP).

Większość wykorzystanych w przykładowej aplikacji API opiera się o architekturę REST. Najczęściej spotykanymi formatami odpowiedzi API aplikacji sieciowych są XML (w tym jego pochodne) oraz JSON. Niektóre interfejsy umożliwiają zadeklarowanie preferowanego formatu odpowiedzi³.

XML

XML jest podzbiorem języka SGML⁴, używanym do reprezentacji danych w przejrzysty, semantycznie strukturalizowany sposób. Ponieważ jest uproszczoną wersją SGML, działanie parserów tego języka jest znacznie bardziej wydajne i spójne [23]. Prostota oraz otwartość standardu sprawiły, iż jest obecnie jednym z najpopularniejszych języków znacznikowych na świecie.

Przykładowe pochodne XML spotykane w internecie:

RDF – format opisu zasobów internetowych,

SOAP – wymieniony wcześniej format komunikacji spotykany w niektórych API,

RSS, Atom – formaty kanałów wiadomości.

¹ Między innymi *Java EE*.

² ang. *Simple Object Access Protocol*

³ Przykład: omówiony w dalszej części pracy serwis BBC (rozdział 4.8, strona 61).

⁴ ang. *Standard Generalized Markup Language*, uogólniony język znaczników.

JSON

Podzbiór języka JavaScript. Zyskał popularność dzięki powszechnemu wykorzystaniu jako format wymiany danych w aplikacjach opartych o technologię AJAX. Charakteryzuje się minimalnym narzutem składni w stosunku do przesyłanych danych, stąd często bywa używany jako lżejsza alternatywa dla XML [24].

4.1.3. Biblioteki API

Obsługa internetowego API przez aplikację wiąże się z koniecznością częstego wykonywania tych samych czynności. Dla prostego API typu REST będą to:

1. konstrukcja adresu URL zasobu,
2. ustalenie metody HTTP oraz nagłówków wywołania,
3. pobranie odpowiedzi API z wcześniej utworzonego URL,
4. parsowanie odpowiedzi,
5. utworzenie/wypełnienie obiektów języka reprezentujących odpowiedź API.

Naturalnym sposobem implementacji powtarzających się wielokrotnie czynności jest wydzielenie dotyczącej nich logiki biznesowej do zewnętrznej biblioteki. Biblioteka wspomagająca pracę z API może być rozwijana przez jego twórców lub osoby/zespoły niezwiązane bezpośrednio z danym serwisem internetowym (na podstawie dokumentacji lub inżynierii wstecznej).

Wiele bibliotek może implementować metody dostępu do jednego API. Mogą się różnić językiem lub metodyką wykorzystania. Większość API omówionych w dalszej części pracy posiada dedykowaną bibliotekę dla języka Python.

Przykładem jest wzorcowa biblioteka *python-musicbrainz2* implementująca metody API MusicBrainz – rozwijana przez osobę związaną z projektem. Biblioteki dedykowane innym językom programowania (przykład: *phpbrainz*) – dzięki szczegółowej dokumentacji i otwartej naturze serwisu – są rozwijane przez zewnętrznych programistów.

Do zalet wynikających z użycia biblioteki dedykowanej do obsługi danego API należą:

- brak styczności z surowym formatem, w którym przebiega komunikacja (XML, JSON),
- wykorzystanie konwencji oraz składni języka, w którym jest napisana biblioteka,
- uproszczona aktualizacja oraz dodawanie nowych funkcji,
- ogólne przyspieszenie i uproszczenie implementacji funkcji związanych z API.

4.2. Projekt MusicBrainz

4.2.1. O projekcie

Celem projektu MusicBrainz jest utworzenie oraz ciągła aktualizacja otwartej bazy metadanych muzycznych. Serwis powstał jako wolna alternatywa dla ograniczonego licencjami CDDB¹.

Rozwój bazy danych jest oparty o zmiany wykonywane przez wolontariuszy: automatyczne (zatwierdzane natychmiast) oraz zwykłe (wymagające weryfikacji przez innych użytkowników). Każda wprowadzana zmiana posiada odnośnik do zewnętrznego źródła potwierdzającego jej zasadność. Dodatkowo serwis posiada liczne wytyczne dotyczące sposobu formatowania oraz reprezentacji wybranych typów danych. Efektem jest wysoka wartość informacji zawartych w bazie.

Baza posiada pięć podstawowych encji:

artysta (ang. *artist*) – reprezentuje osobę lub grupę muzyczną,

publikacja (ang. *release*) – album, singiel lub dowolna inna publikacja muzyczna,

grupa publikacji (ang. *release group*) – grupa publikacji, np. różne edycje tego samego albumu,

utwór (ang. *track*) – pojedynczy utwór muzyczny,

wytwórnia (ang. *label*) – wytwórnia muzyczna lub dystrybutor.

Każda posiada unikalny identyfikator w formacie UUID², nazywany MBID. Umożliwia on między innymi współlistnienie oraz rozróżnianie wykonawców o tej samej nazwie.

Większość danych dostępna jest w *domenie publicznej*. Wybrane elementy, takie jak: statystyki, tagi, informacje o edycjach, obowiązuje licencja *Creative Commons Attribution-NonCommercial-ShareAlike 2.0* [25].

Będąc aktualnym oraz poprawnym źródłem metadanych muzycznych baza danych projektu MusicBrainz wykorzystywana jest przez wiele komercyjnych serwisów. Do najbardziej znanych należą: *BBC Music* oraz *Last.fm*.

4.2.2. API XML Web Service

API serwisu MusicBrainz umożliwia pobranie wybranych informacji z bazy w zautomatyzowany sposób. Pierwsza wersja oparta o RDF została zastąpiona przez XML REST API (*MusicBrainz Web Service 1.0*). Nowe API ułatwia tworzenie bazujących na nim rozwiązań w bardziej spójny i czytelny sposób.

Jeżeli tworzona aplikacja wykonuje więcej niż jedno żądanie HTTP na sekundę, zalecane jest uruchomienie własnego serwera lustrzanego. Mechanizm replikacji bazy

¹ ang. *Compact Disc Database*, własnościowa baza danych poświęcona publikacjom muzycznym w formie CD.

² ang. *Universally Unique Identifier*, określony przez RFC 4122 format uniwersalnych identyfikatorów.

PostgreSQL (*Live Data Feed*) jest dostępny nieodpłatnie dla zastosowań niekomercyjnych, możliwe jest również negocjowanie licencji komercyjnej.

Podstawowe operacje, takie jak pobranie informacji o artyście lub publikacji muzycznej, sprowadzają się do wysłania zapytania HTTP GET na adres URL danej metody API [26].

Przykład 4.1: Pobranie informacji o artyście.

Zapytanie o artystę określonego identyfikatorem MBID ma postać URL:

`http://musicbrainz.org/ws/1/artist/c0b2500e-0cef-4130-869d-732b23ed9df5?type=xml`

Domyślnie zwracane są jedynie podstawowe informacje, w wypadku artysty są to: nazwa, typ, oraz daty (*Listing 4.1*). Dołączenie dodatkowych typów danych polega na dodaniu do URL metody API parametru `inc`.

Na przykład: wywołanie adresu

`http://musicbrainz.org/ws/1/artist/c0b2500e-0cef-4130-869d-732b23ed9df5?type=xml&inc=url-rels`

spowoduje włączenie informacji o powiązanych adresach URL (*Listing 4.2*).

Listing 4.1. Podstawowa odpowiedź XML MusicBrainz API

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <metadata xmlns="http://musicbrainz.org/ns/mmd-1.0#">
3   <artist id="c0b2500e-0cef-4130-869d-732b23ed9df5" type="Person">
4     <name>Tori Amos</name>
5     <sort-name>Amos, Tori</sort-name>
6     <life-span begin="1963-08-22"/>
7   </artist>
8 </metadata>
```

Listing 4.2. Rozszerzona odpowiedź XML MusicBrainz API

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <metadata xmlns="http://musicbrainz.org/ns/mmd-1.0#">
3 <artist id="c0b2500e-0cef-4130-869d-732b23ed9df5" type="Person">
4   <name>Tori Amos</name>
5   <sort-name>Amos, Tori</sort-name>
6   <life-span begin="1963-08-22"/>
7   <relation-list target-type="Url">
8 <relation type="IMDb" target="http://www.imdb.com/name/nm0002169/" begin="" end=""/>
9 <relation type="Wikipedia" target="http://en.wikipedia.org/wiki/Tori_Amos" begin="" end=""/>
10 <relation type="Discogs" target="http://www.discogs.com/artist/Tori+Amos" begin="" end=""/>
11 <relation type="Fanpage" target="http://www.thedent.com/index.php" begin="" end=""/>
12 <relation type="OfficialHomepage" target="http://www.toriamos.com/" begin="" end=""/>
13 <relation type="Discography" target="http://www.yessaid.com/albums.html" begin="" end=""/>
14 <relation type="Discography" target="http://www.hereinmyhead.com/" begin="" end=""/>
15 <relation type="Myspace" target="http://www.myspace.com/toriamos" begin="" end=""/>
16   </relation-list>
17 </artist>
18 </metadata>
```

Lista wszystkich dostępnych argumentów parametru `inc` znajduje się w dokumentacji [26]. Możliwe jest ich łączenie za pomocą separatora „+”:

`inc=url-rels+artist-rels`

Przykład 4.2: Poszukiwanie artysty o podanej nazwie.

Drugim przypadkiem użycia API jest wyszukiwanie tekstowe po wartości wybranego pola, do którego MusicBrainz używa wydajnego silnika *Lucene*¹. Przykładowe zapytanie o artystę zawierającego ciąg *Tool* w polu `name` może mieć postać:

`http://musicbrainz.org/ws/1/artist/?type=xml&name=Tool`

Odpowiedzią API (*Listing 4.3*) jest zbiór artystów spełniających dane kryterium wyszukiwania wraz z wartością (`score`) określającą trafność wyniku.

Listing 4.3. Przykładowy wynik wyszukiwania za pomocą MusicBrainz API

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <metadata xmlns="http://musicbrainz.org/ns/mmd-1.0#"
3   xmlns:ext="http://musicbrainz.org/ns/ext-1.0#"><artist-list offset="0" count="16">
4   <artist type="Group" id="66fc5bf8-daa4-4241-b378-9bc9077939d2" ext:score="100">
5     <name>Tool</name>
6     <sort-name>Tool</sort-name>
7     <disambiguation>US progressive metal band</disambiguation>
8     <life-span begin="1990"/>
9   </artist>
10  <artist type="Person" id="bbfa1e12-8b61-4ed8-bc6e-52d2298f41a2" ext:score="100">
11    <name>Tool</name>
12    <sort-name>Tool</sort-name>
13    <disambiguation>90s trance DJ Thomas Buttner, aka DJ Tool</disambiguation>
14  </artist>
15  <artist type="Unknown" id="217f8cde-243c-4c0e-b4c5-175b5d55f2f2" ext:score="62">
16    <name>Power Tool</name>
17    <sort-name>Power Tool</sort-name>
18  </artist>
19  (...)
20 </metadata>

```

W analogiczny sposób można pobierać dane dotyczące pozostałych encji obecnych w bazie danych. Omawiana w dalszej części pracy aplikacja pobiera informacje dotyczące artysty, publikacji oraz grup publikacji. W chwili pisania niniejszej pracy trwają również prace nad *MusicBrainz XML Web Service 2.0*, będącym kompletnym API dla nowego schematu bazy MusicBrainz (ang. *Next Generation Schema*).

Niskopoziomowy mechanizm działania REST API jest analogiczny w wypadku pozostałych serwisów, dlatego w kolejnych działach położono nacisk na obsługę omawianych API przy użyciu bibliotek języka Python.

¹ Otwarty silnik indeksowania oraz przeszukiwania tekstu.

4.2.3. Biblioteka *python-musicbrainz2*

Wzorcową implementacją *MusicBrainz XML Web Service 1.0* jest napisana w języku Python biblioteka *python-musicbrainz2*, dostępna na licencji BSD. Tworzona aplikacja wymaga wsparcia *release groups*, które zostało wprowadzone w wersji 0.7.1 biblioteki.

Przykład 4.3: Pobieranie informacji o publikacji muzycznej określonej identyfikatorem MBID. Pracując z biblioteką programista nie ma bezpośredniego kontaktu z odpowiedziami XML – interakcja z API odbywa się na wyższym poziomie abstrakcji poprzez obiekty języka Python. Zakres pobieranych informacji dotyczących publikacji muzycznej określa obiekt typu `ReleaseIncludes` (*Listing 4.4*).

Praca z API zawiera się w trzech krokach:

1. Zdefiniowanie adresu API (np. serwera lustrzanego) (wiersz 10),
2. Utworzenie obiektu typu `Query` operującego na tym API (wiersz 11),
3. Uzyskanie odpowiedzi `mb_release` poprzez wywołanie zapytania z opcjonalnym filtrem (wiersz 12).

Listing 4.4. Prosty przykład użycia *python-musicbrainz2*

```
1 import musicbrainz2.webservice as ws
2 MB_RELEASE_INCLUDES = ws.ReleaseIncludes(
3     tracks=True,
4     trackRelations=True,
5     artistRelations=True,
6     releaseRelations=True,
7     urlRelations=True,
8 )
9
10 mbid = 'b1b9bf98-86fa-4092-950f-1fb6744a25b4'
11 mb_webservice = ws.WebService(host=musicbrainz.org)
12 mb_query = ws.Query(mb_webservice)
13 mb_release = mb_query.getReleaseById(mbid, MB_RELEASE_INCLUDES)
```

Odpowiedź `mb_release` zawiera zwrócone w formacie XML informacje opakowane w obiekt języka Python. Posiada on metody umożliwiające dostęp do wybranych segmentów odpowiedzi w naturalny dla programisty sposób. Przykład: `mb_release.tracks` jest listą zawierającą informacje o utworach należących do danej publikacji.

Przykład 4.4: Pobieranie szczegółowych informacji o artyście określonym identyfikatorem MBID. Omawiana aplikacja wymaga pobrania jak największej ilości pożądaných informacji o wybranym artyście za pomocą pojedynczego wywołania API.

Dla *MusicBrainz XML Web Service 1.0* będą to:

- podstawowe informacje (oraz aliasy),
- relacje z innymi artystami,

- powiązane adresy url,
- podstawowe informacje o oficjalnych publikacjach (tytuł, liczba utworów, data wydania),
- grupy publikacji.

Niestety, obecny w bibliotece obiekt typu `ArtistIncludes` nie może zostać zainicjowany w sposób umożliwiający dodanie argumentu `release-events` do parametru `inc` zapytania¹. Jest on odpowiedzialny za włączenie informacji o datach wydania publikacji muzycznych do odpowiedzi API.

Język Python umożliwia wykorzystanie techniki programistycznej znanej jako *duck typing*². Dzięki przejrzystej, obiektowej architekturze biblioteki, dodanie brakującej funkcji sprowadza się do utworzenia zastępczej implementacji klasy `ArtistIncludes` o nazwie `ExtendedArtistIncludes` (Listing 4.5). Problem został rozwiązany bez ingerencji w wewnętrzny kod biblioteki. Dodatkowo aplikacja będzie działać poprawnie, nawet gdy w przyszłości biblioteka będzie sama oferować brakującą funkcję.

Listing 4.5. Dodanie brakującej funkcji *python-musicbrainz2*

```

1 import musicbrainz2.webservice as ws
2
3 class ExtendedArtistIncludes(ws.IIncludes):
4     def createIncludeTags(self):
5         return ['url-rels', 'sa-Official', 'artist-rels',
6               'release-groups', 'aliases', 'release-events', 'counts']
7
8 MB_ARTIST_INCLUDES = ExtendedArtistIncludes()
9 mbid = '66fc5bf8-daa4-4241-b378-9bc9077939d2'
10
11 mb_webservice = ws.WebService(host=musicbrainz.org)
12 mb_query      = ws.Query(mb_webservice)
13 mb_artist     = mb_query.getArtistById(mbid, MB_ARTIST_INCLUDES)
```

4.3. Last.fm

4.3.1. O serwisie

Last.fm powstał w 2005 jako połączenie serwisu agregującego statystyki dotyczące odsłuchanych utworów *Audioscrobber* z radiem internetowym *Last.fm*.

W obecnej formie serwis oferuje między innymi:

- rozbudowane statystyki oraz historię odsłuchań zarejestrowanych użytkowników,
- płatne, spersonalizowane radio,

¹ Funkcja zostanie najprawdopodobniej dodana w kolejnej wersji biblioteki.

² Technika polegająca na identyfikacji obiektu po posiadanych metodach, a nie zadeklarowanym typie.

- rozwijane przez społeczność galerie i artykuły dotyczące artystów oraz publikacji muzycznych.

REST API udostępniające wybrane zasoby serwisu w formie odpowiedzi XML wymaga autoryzacji kluczem, który każdy użytkownik może otrzymać bezpłatnie. Indywidualny klucz umożliwia blokowanie łamiących licencje aplikacji oraz tworzenie statystyk użycia. API daje dostęp zarówno do publicznych zasobów serwisu, jak i prywatnych, wymagających dodatkowego uwierzytelnienia loginem oraz hasłem użytkownika.

Ponieważ Last.fm aktywnie wspiera oraz wykorzystuje projekt MusicBrainz¹, do zgromadzonych zasobów można odwoływać się przy użyciu identyfikatora MBID. Artykuły dotyczące wykonawców oraz publikacji są objęte licencjami *Creative Commons Attribution/Share-Alike*, oraz *GNU Free Documentation License*. Pozostałe zasoby API są dostępne nieodpłatnie do zastosowań niekomercyjnych [27].

4.3.2. Biblioteka *pylast*

Istnieje kilka niezależnych bibliotek dla języka Python. Najbardziej kompletną implementacją Last.fm API jest *pylast*.

Przykład 4.5: Pobranie artykułu dotyczącego artysty o wskazanym identyfikatorze MBID realizują następujące kroki (*Listing 4.6*):

1. utworzenie interfejsu działającego z przydzielonym kluczem dostępu do API (wiersz 5),
2. wyszukanie oraz utworzenie reprezentacji artysty o wskazanym MBID (wiersz 6),
3. pobranie artykułu (wiersz 7).

Listing 4.6. Przykład użycia biblioteki *pylast*

```

1 import pylast
2 from django.conf import settings
3
4 mbid = '66fc5bf8-daa4-4241-b378-9bc9077939d2'
5 lastfm = pylast.get_lastfm_network(api_key = settings.LASTFM_API_KEY)
6 lastfm_artist = lastfm.get_artist_by_mbid(mbid)
7 lastfm_abstract = lastfm_artist.get_bio_summary()
```

Rozszerzanie biblioteki Pożądane jest, aby przykładowa aplikacja posiadała informacje o artystach wykonujących podobny rodzaj muzyki. Wymagane informacje to:

- nazwa,
- MBID,
- wartość liczbowa określająca stopień podobieństwa.

Last.fm API umożliwia pobranie tego typu informacji (są one obecne w odpowiedzi XML), jednak metoda `get_similar` biblioteki *pylast* zwraca listę krotek zawierających jedynie wartość liczbową oraz obiekt reprezentujący artystę. Wywołanie metody

¹ Last.fm udostępnia własny mirror MusicBrainz (<http://www.uk.musicbrainz.org>).

`artist.get_mbid` tworzy nowe zapytanie do API, co przy większej ilości przetwarzanych obiektów stanowi zbyt duży narzut.

Dzięki architekturze języka możliwe jest stworzenie alternatywnej implementacji metody `get_similar` poza biblioteką *pylast* (Listing 4.7). Programista nie jest ograniczany przez enkapsulację, która w języku Python jest jedynie konwencją, a nie narzuconym mechanizmem [2].

Listing 4.7. Przykład rozszerzenia biblioteki *pylast*

```
1 import pylast
2
3 def get_similar_extended(lastfm_artist, limit=10):
4     params = lastfm_artist._get_params()
5     params['limit'] = pylast._unicode(limit)
6     doc = lastfm_artist._request('artist.getSimilar', True, params)
7     names = pylast._extract_all(doc, "name")
8     mbids = pylast._extract_all(doc, "mbid")
9     matches = pylast._extract_all(doc, "match")
10    similar_artists = []
11    for i in range(0, len(names)):
12        if mbids[i]:
13            similar_artists.append({'name': names[i],
14                                   'score': matches[i]},
15                                   {'mbid': mbids[i]})
16    return similar_artists
```

Zewnętrzna funkcja `get_similar_extended` bazuje na oryginalnej metodzie `get_similar`. Odwołuje się do „prywatnych” metod biblioteki *pylast*: `_unicode`, `_extract_all`, dodając obecne w odpowiedzi XML identyfikatory MBID do listy wyników. Obecny w wierszu 12 warunek gwarantuje, iż zwrócona lista słowników będzie zawierać jedynie informacje dotyczące artystów posiadających MBID¹. Obiekt `lastfm_artist` jest reprezentacją artysty w serwisie Last.fm (Listing 4.6).

Jest to dobry przykład elastyczności oraz oszczędności, jaką charakteryzuje się język Python: programista może tworzyć rozszerzenia bazujące na zewnętrznej bibliotece, bez ingerencji w jej wewnętrzny kod lub dublowania metod na zewnątrz.

¹ Możliwy jest przypadek: artysta, którego odsłuchanie zostało przesłane do *Last.fm*, będzie tam obecny mimo, iż nie został (jeszcze) wprowadzony do bazy *MusicBrainz*.

4.4. Flickr

4.4.1. O serwisie

Flickr jest miejscem gdzie zarejestrowany użytkownik może przechowywać i udostępniać zdjęcia oraz krótkie materiały wideo. Serwis został wykorzystany w tworzonej aplikacji z kilku przyczyn:

Popularność Jest jednym z najbardziej dynamicznie rosnących repozytoriów zdjęć w Internecie. W 2009 roku liczba dostępnych zdjęć przekroczyła cztery miliardy.

Poziom zasobów Serwis skupia ludzi mających przynajmniej podstawową wiedzę o fotografii. Wielu użytkowników to profesjonaliści traktujący serwis *Flickr* jako rozbudowane portfolio. Przekłada się to na jakość zdjęć. Szczególnie interesujące są galerie zdjęć z koncertów muzycznych.

Mechanizmy przeszukiwania Oprócz obecnych w samych zdjęciach metadanych EXIF¹ dostępne są opisy oraz tagi przypisywane przez samych użytkowników. Dodatkowo możliwe jest zdefiniowanie zaawansowanych kryteriów określających między innymi:

- prawa autorskie, licencje,
- filtr treści niepożądanych.

4.4.2. API oraz biblioteka *flickrapi*

Zasady wykorzystania udostępnionego API przypominają te z Last.fm. Programista otrzymuje darmowy klucz służący do identyfikacji tworzonej aplikacji oraz autoryzacji dostępu do API. Obsługiwane są zarówno zapytania REST, jak i komunikacja w formacie SOAP. Dostęp jest bezpłatny dla projektów niekomercyjnych [28].

Biblioteką wspomagającą pracę z serwisem w języku Python jest *flickrapi* [29]. Pokrywa ona zarówno funkcje wyszukiwania istniejących zasobów, jak i wysyłanie nowych zdjęć do serwisu.

Przykład 4.6: Tworzona aplikacja wykorzystuje *flickrapi* do wyszukiwania zdjęć związanych z danym artystą. *Listing 4.8* zawiera uproszczony mechanizm wykorzystany w tworzonej aplikacji:

1. Obiekt `flickr` umożliwia obsługę API przy użyciu klucza dostępu (wiersz 7).
2. Wyszukiwanie odbywa się po tagach, gdyż dają bardziej dokładne wyniki. Tagiem staje się nazwa artysty (wiersz 9) oraz opcjonalne aliasy (wiersz 11).
3. Obiekt `includes` (wiersz 13) definiuje, jakie dodatkowe dane mają być zwrócone. W omawianym przykładzie są to: nazwa autora oraz URL zdjęcia i miniaturki.

¹ ang. *Exchangeable image file format*, format rozszerzający popularne pliki graficzne o metadane.

4. Obiekt `licenses` określa rodzaje licencji oraz warunków prawnych, jakie powinny spełniać zwrócone zdjęcia. Wiersz 14 zawiera wszystkie licencje dostępne dla aplikacji niekomercyjnej.
5. Obiekt `data_walker` jest utworzonym przez `flickr` iteratorem po liście wyników wyszukiwania (wiersz 16-12). Metoda `data_walker.next` zwraca obiekt reprezentujący pojedynczy wynik (zdjęcie). Liczba wyników jest ograniczona przez wartość `per_page`. Przekroczenie każdej krotności wartości `per_page` powoduje wykonanie w tle kolejnego zapytania do API. Jest to proces automatyczny.

Listing 4.8. Wyszukiwanie zdjęć za pomocą *flickrapi*

```
1 import flickrapi
2 from django.conf import settings
3
4 artist = CachedArtist.get('216de703-21b9-4396-a01f-7927a754c4a7')
5 flickr = flickrapi.FlickrAPI(settings.FLICKR_API_KEY, cache=True)
6
7 artist_tags = artist.name
8 if 'aliases' in artist:
9     artist_tags += ', ' + ', '.join(artist.aliases)
10
11 includes = 'owner_name, url_sq, url_o'
12 licenses = '1,2,3,4,5,6,7'
13
14 data_walker = flickr.walk(tag_mode='any',
15                           tags=artist_tags.lower(),
16                           media='photos',
17                           license=licenses,
18                           extras=includes,
19                           per_page=10)
20
21 first_photo = data_walker.next()
22 second_photo = data_walker.next()
```

Programista pracuje z obiektem `data_walker`, nie martwiąc się o mechanizm komunikacji z API. W tworzonej aplikacji wartość `per_page` została dobrana w sposób ograniczający liczbę odwołań do Flickr API do minimum.

4.5. YouTube

4.5.1. O serwisie

YouTube jest największym serwisem umożliwiającym umieszczanie oraz odtwarzanie krótkich filmów wideo. Powstał w 2005 roku. W 2007 został wykupiony przez firmę Google. Zawiera dużą ilość materiałów poświęconych muzyce, wideoklipów, fragmentów koncertów oraz wywiadów. Część wykonawców posiada własne *kanały*, w których zamieszcza tego typu treści w ramach autopromocji.

Serwis umożliwia osadzanie większości zasobów na zewnętrznych stronach oraz tworzenie aplikacji (w tym komercyjnych) w oparciu o rozbudowane API. Opisywana w następnym rozdziale aplikacja wykorzystuje YouTube jako źródło materiałów wideo.

4.5.2. Gdata API

API serwisu YouTube opiera się o *Google Data Protocol 2.0*. Jest to zgodny z *Atom-Pub*¹ protokół wykorzystywany w większości produktów firmy Google. Część zasobów wymaga uwierzytelniania loginem oraz hasłem serwisu Youtube lub kluczem deweloperskim.

Programista języka Python otrzymuje od Google bibliotekę *gdata*. Jest to zintegrowane środowisko umożliwiające interakcję tworzonej aplikacji z produktami firmy. Pakiet odpowiedzialny za komunikację z API serwisu YouTube to `gdata.youtube`.

Przykład 4.7: Tworzona aplikacja wykorzystuje YouTube w dwóch przypadkach:

- Gdy artysta posiada własny kanał: do pobrania jego zawartości (*Listing 4.9*). Jedyną wymaganą informacją jest nazwa profilu YouTube: `id` (wiersz 4).
- Gdy artysta nie posiada kanału: do wyszukania związanych z nim wideo (*Listing 4.10*). Wyszukiwanie odbywa się po ciągu znaków będących nazwą wykonawcy (wiersz 4) w ramach wybranej kategorii (wiersz 11). Z listy wyników usunięte są pozycje posiadające ograniczenia wynikające na przykład z praw autorskich lub lokalizacji geograficznej (wiersz 9).

Wynikiem każdego scenariusza będzie utworzenie obiektu `feed`.

Listing 4.9. Pobranie kanału wideo przy użyciu biblioteki *gdata*

```
1 import gdata.youtube as yt
2 import gdata.youtube.service as yts
3
4 id = 'dreamtheater'
5 url = "http://gdata.youtube.com/feeds/api/users/%s/uploads" % id
6
7 service = yts.YouTubeService()
8 feed = service.GetYouTubeVideoFeed(url)
```

Listing 4.10. Wyszukiwanie wideo przy użyciu biblioteki *gdata*

```
1 import gdata.youtube as yt
2 import gdata.youtube.service as yts
3
4 artist_name = 'Dream Theater'
5
6 service = yts.YouTubeService()
7 query = yts.YouTubeVideoQuery()
```

¹ Zdefiniowany w *RFC 5023* protokół publikacji oraz edycji zasobów sieci Internet przy użyciu HTTP oraz XML.

```

8 query.orderby = 'relevance'
9 query.racy = 'exclude'
10 query.max_results = 10
11 query.categories.append('Music')
12 query.vq = artist_name
13
14 feed = service.YouTubeQuery(query)

```

Obiekt `feed` zawiera metody umożliwiające dostęp do szczegółowych informacji na temat każdego wideo będącego na liście wyników `feed.entry` (*Listing 4.11*). W przykładowej aplikacji wykorzystywane są: tytuł, czas trwania, URL klipu w serwisie YouTube, URL odtwarzacza Flash oraz miniaturka.

Listing 4.11. Przetwarzanie obiektu `feed` biblioteki *gdata*

```

1 results = []
2 for entry in feed.entry:
3     video = {
4         'title':      entry.media.title.text,
5         'duration':   entry.media.duration.seconds,
6         'url':        entry.media.player.url,
7         'player':     entry.GetSwfUrl(),
8         'thumb':      entry.media.thumbnail[0].url
9     }
10    results.append(video)

```

4.6. DBpedia

4.6.1. O projekcie

Celem projektu DBpedia jest semantyczna klasyfikacja zawartości Wikipedii oraz jej udostępnienie w czytelnej dla aplikacji formie. Obecnie projekt łączy w sobie funkcje zaawansowanego API dla zasobów Wikipedii oraz węzła *Linked Data* [30]. Innymi słowy: udostępnia otwarte zasoby wiedzy w prosty do oprogramowania i przetwarzania sposób.

4.6.2. Biblioteka *surf-sparql*

Preferowaną metodą dostępu do zasobów DBpedii jest wykonanie zapytania SPARQL¹ określającego zakres pożądaných informacji. Ponieważ SPARQL operuje na danych reprezentowanych w formacie RDF/XML, do obsługi DBpedii w języku Python można wykorzystać dedykowaną tym rozwiązaniom bibliotekę *surf*.

Przykład 4.8: Tworzona aplikacja pobiera podstawowe informacje dotyczące artysty lub publikacji muzycznej z serwisu MusicBrainz. Należą do nich również adresy URL

¹ Rekomendowany przez organizację W3C język zapytań RDF.

powiązanych serwisów. Jeżeli jeden z URL wskazuje na artykuł Wikipedii, pożądane jest, aby aplikacja pobrała krótkie streszczenie tego artykułu (ang. *abstract*).

Do tego zadania idealnie nadaje się DBpedia oraz *surf* (Listing 4.12). Znając nazwę przykładowego zasobu (*Marek_Grechuta*) pobranie wartości *abstract* sprowadza się do:

1. określenia punktu dostępu SPARQL (wiersz 7),
2. utworzenia sesji do pracy z punktem (wiersz 8),
3. wykonania zapytania w ramach sesji (wiersz 15),
4. pobrania tekstu, jeżeli został zwrócony pozytywny wynik (wiersz 18).

Listing 4.12. Wykonanie zapytania SPARQL przy użyciu biblioteki *surf*

```

1 import surf
2
3 resource = 'Marek_Grechuta'
4 lang = 'en'
5
6 store = surf.Store(reader = "sparql_protocol",
7                     endpoint = "http://dbpedia.org/sparql")
8 session = surf.Session(store)
9 query = """SELECT ?abstract WHERE {{
10     <http://dbpedia.org/resource/%s>
11     <http://dbpedia.org/property/abstract>
12     ?abstract FILTER langMatches( lang(?abstract), '%s' )
13 }}""" % (resource, lang)
14
15 result = session.default_store.execute_sparql(query)
16
17 if result['results']['bindings']:
18     abstract = result['results']['bindings'][0]['abstract']

```

4.7. reCAPTCHA

4.7.1. O projekcie

Spam jest rozległym, internetowym zjawiskiem. Przyjmuje wiele różnych odmian: od niechcianej poczty e-mail, po natrętne roboty HTTP generujące niepożądane treści. Podatnymi na ten ostatni są wszystkie serwisy internetowe udostępniające formularze. Niezależnie od tego, czy jest to formularz kontaktowy, pole zawierające komentarz, czy też klasyczne forum dyskusyjne pożądane jest, aby wypełniał je człowiek.

Powszechnie stosowaną techniką weryfikacji jest CAPTCHA¹, czyli prosty test przyjmujący (na przykład) formę grafiki zawierającej zniekształcony tekst, który użytkownik powinien wprowadzić z klawiatury, aby udowodnić, iż jest człowiekiem.

Projekt reCAPTCHA udostępnia tego typu mechanizm do wykorzystania w rozwijanym projekcie [31]. Cechy charakterystyczne:

¹ ang. *Completely Automated Public Turing test to tell Computers and Humans Apart*

- otwarte, bezpłatne API (również dla zastosowań komercyjnych),
- alternatywna CAPTCHA dźwiękowa,
- wspiera proces digitalizacji książek.¹

4.7.2. Biblioteka *recaptcha-client*

Ze względów bezpieczeństwa każdy serwis otrzymuje dedykowany komplet kluczy (prywatny oraz publiczny) do pracy z API. Biblioteka *recaptcha-client* umożliwia łatwą integrację zabezpieczenia CAPTCHA z istniejącymi formularzami.

Przykład 4.9: Tworzona aplikacja posiada funkcję ręcznego odświeżenia informacji o artyście. Pożądane jest, aby czynność tę mógł wykonać jedynie człowiek.

Biblioteka *recaptcha-client* udostępnia metodę zwracającą gotowy kod HTML odpowiedzialny za osadzenie testu na stronie internetowej. Możliwe jest również jego spersonalizowanie poprzez napisanie własnej, alternatywnej funkcji (*Listing 4.13*). Osadzony w publicznie dostępnym dokumencie HTML kod zawiera publiczny klucz.

Listing 4.13. Kod osadzający test reCAPTCHA

```

1 from django.conf import settings
2
3 def get_captcha_html(api_server):
4     return """
5 <script>
6 var RecaptchaOptions={theme:'white', lang:'en',
7                         custom_theme_widget:'null'};
8 </script>
9 <script src="% (ApiServer)s/challenge?k=% (PublicKey)s"></script>
10
11 <noscript>
12   <div><object data="% (ApiServer)s/noscript?k=% (PublicKey)s"
13             type="text/html" height="300px" width="100%">
14   </object></div>
15   <p><textarea name="recaptcha_challenge_field" rows="3" cols="40">
16   </textarea></p>
17   <p><input type='hidden' name='recaptcha_response_field'
18         value='manual_challenge' /></p>
19 </noscript>
20 """ % {
21     'ApiServer' : api_server,
22     'PublicKey' : settings.RECAPTCHA_PUB_KEY
23 }
```

Klucz prywatny znany jest jedynie twórcy aplikacji i jest używany po stronie serwera, podczas walidacji formularza (*Listing 4.14*). Metoda `submit` przesyła wprowadzony

¹ Użytkownik otrzymuje grafikę zawierającą dwa wyrazy. Jednym z nich jest fragment tekstu, z którym nie poradziło sobie oprogramowanie OCR.

przez użytkownika tekst wraz z jego adresem IP, tokenem identyfikującym test oraz kluczem prywatnym aplikacji.

Odpowiedzią jest obiekt `captcha`, którego metoda `is_valid` określa czy użytkownik przeszedł test.

Listing 4.14. Kod walidujący test reCAPTCHA

```

1 import recaptcha.client.captcha as rc
2 from django.conf import settings
3
4 if request.POST:
5     ip      = request.META['REMOTE_ADDR']
6     token   = request.POST['recaptcha_challenge_field']
7     input   = request.POST['recaptcha_response_field']
8     captcha = rc.submit(token, input,
9                          settings.RECAPTCHA_PRIV_KEY, ip)
10    if captcha.is_valid:
11        # dalsze przetwarzanie

```

Przy użyciu opisanej techniki można dodać test reCAPTCHA do każdego, ogólnodostępnego formularza, nie ingerując w już istniejącą logikę biznesową.

4.8. BBC Music

4.8.1. O serwisie

BBC Music jest serwisem udostępniającym informacje na temat muzyki obecnej w produkcjach koncernu medialnego BBC. Korzysta z otwartych źródeł takich jak MusicBrainz oraz Wikipedia, uzupełniając pochodzące z nich informacje własnymi zasobami. Udostępnia proste RESTful API nieodpłatnie do zastosowań niekomercyjnych [33].

Dzięki wykorzystaniu MusicBrainz jako głównego źródła informacji, możliwe jest odwoływanie się do zasobów BBC Music przy użyciu identyfikatorów MBID.

4.8.2. Obsługa API bez dedykowanej biblioteki

BBC Music API nie posiada dedykowanej biblioteki. Umożliwia określenie preferowanego formatu odpowiedzi, mogą to być między innymi XML (*Listing 4.15*) lub JSON.

Listing 4.15. Przykładowa odpowiedź XML *BBC Music API*

```

1 <?xml version="1.0"?>
2 <artist artist_type="Group">
3     <name>Tool</name><sort_name>Tool</sort_name>
4     <gid>66fc5bf8-daa4-4241-b378-9bc9077939d2</gid>
5     <disambiguation>US progressive metal band</disambiguation>
6     <begin_date>1990-00-00</begin_date><end_date/>
7     <wikipedia_article>

```

```

8     <content>Tool is an American rock band (...)</content>
9     <url>http://en.wikipedia.org/wiki/Tool_%28band%29</url>
10    <title>Tool (band)</title><word_count>198</word_count>
11    <updated_at>2010-05-14T21:09:35+01:00</updated_at>
12  </wikipedia_article>
13 </artist>

```

Przykład 4.10: Zdarza się, że zasób DBpedii posiada inną formę, niż ostatni segment URL artykułu na Wikipedii¹. W efekcie pobranie artykułu wymaga dodatkowej weryfikacji nazwy zasobu lub narzutu kolejnych zapytań HTTP. Serwis BBC Music udostępnia kopię skrótu artykułu poprzez swoje API, dzięki czemu może być wykorzystany jako alternatywne dla DBpedii źródło.

Biblioteka *BeautifulSoup* umożliwia parsowanie HTML. Wykorzystany obiekt typu *BeautifulStoneSoup* służy do pracy z formatem XML.

Pobranie požądanej informacji z XML API (*Listing 4.16*) zawiera się w trzech krokach:

1. pobranie odpowiedzi XML REST API (wiersz 7),
2. utworzenie obiektu typu *BeautifulStoneSoup* (wiersz 8-9),
3. pobranie treści przy użyciu generowanych na podstawie XML metod (wiersz 11-12).

Listing 4.16. Parsowanie XML przy użyciu biblioteki *BeautifulSoup*

```

1 from urllib2 import urlopen
2 from BeautifulSoup import BeautifulSoup
3
4 mbid = '66fc5bf8-daa4-4241-b378-9bc9077939d2'
5 api_url = "http://www.bbc.co.uk/music/artists/%s/wikipedia.xml" % mbid
6
7 xml = urlopen(api_url).read()
8 xmlSoup = BeautifulSoup(xml,
9                          convertEntities=BeautifulSoup.HTML_ENTITIES)
10
11 abstract = {'content': xmlSoup.wikipedia_article.content.text,
12            'url':      xmlSoup.wikipedia_article.url.text }

```

Jest to dobry przykład pracy z API bez dedykowanej biblioteki. Dzięki *BeautifulSoup* programista może szybko rozpocząć pracę z przechowywanymi w formacie XML danymi lub stworzyć nową bibliotekę wspomagającą pracę z API bez konieczności implementacji parsera HTML/XML/SGML od podstaw.

4.9. RSS oraz Atom

Ciągła publikacja nowych treści w Internecie spowodowała powstanie technologii umożliwiających śledzenie tych zmian. Ostatnia dekada była okresem formowania się

¹ Przyczyną może być (na przykład) przekierowanie lub zmiana nazwy artykułu.

standardów, czego efektem jest obecność różnych formatów publikacji treści. Do najpopularniejszych rozwiązań należą formaty określone jako RSS oraz standard Atom [34, 35]. Są to dokumenty oparte o XML zawierające całość lub skrót wybranych zasobów dostępnych w serwisie wraz z odnośnikiem URL oryginalnej publikacji. Nazywane często *kanalami* umożliwiają subskrypcję dynamicznych treści w Internecie.

RSS 0.90

Określany również jako *RDF Site Summary*, format stworzony w 1999 przez firmę Netscape na podstawie szkicu standardu RDF. Niekompatybilny z finalnym standardem RDF.

RSS 1.0

Otwarty format stworzony przez *RSS-DEV Working Group*, również nazywany *RDF Site Summary*, kompatybilny z RDF.

RSS 0.91-0.94

Uproszczona wersja RSS 0.90 zaproponowana przez pracownika firmy UserLand. Nazwa została zmieniona na *Rich Site Summary*. Format przestał bazować na RDF.

RSS 2.0

Bazujący na RSS 0.9x *Really Simple Syndication* oferuje mechanizm rozszerzeń oparty o przestrzeń nazw XML.

Rodzina RSS to zbiór opartych o XML formatów, często ze sobą niekompatybilnych, rozwijanych w dwóch gałęziach przez różne organizacje (*Rys. 4.1*):

- oparte o RDF wersje 0.90 (Netscape) oraz 1.0 (RSS-DEV),
- alternatywna gałąź, do której należą 0.91-0.94 oraz 2.0 (UserLand, obecnie Harvard).

Brak wzajemnej kompatybilności, spójnej specyfikacji sprawił, iż w 2003 roku rozpoczęto dyskusję nad nowym formatem. Efektem tych prac jest Atom 1.0.

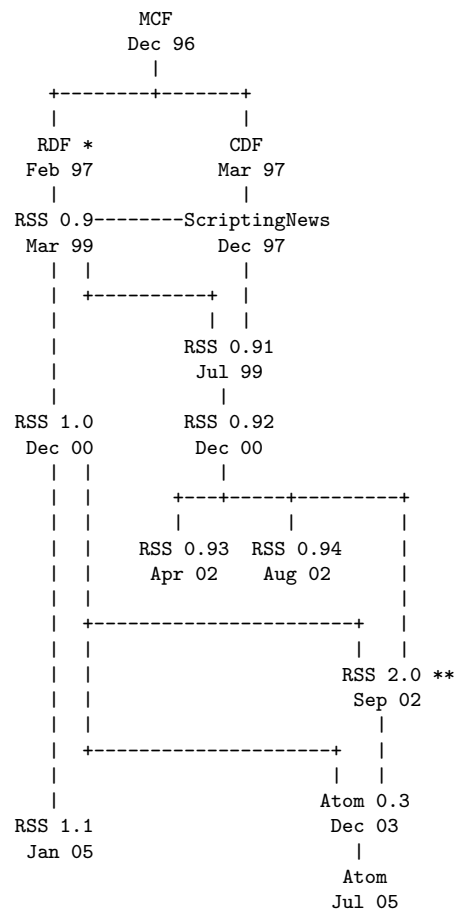
Atom 1.0

Oparty o XML format publikacji treści stanowiący alternatywę dla rodziny RSS. Charakteryzuje się spójną architekturą oraz rozwiązaniem problemów takich jak: określenie rodzaju przechowywanej treści, spójny format daty, internacjonalizacja elementów tekstowych, modularność.

Już wczesna wersja (0.3) została zaadoptowana przez firmę Google. Obecnie używana (1.0) jest propozycją IETF¹ jako domyślnego standardu publikacji tego typu informacji w sieci [35].

Niestety – wiele serwisów internetowych oferuje subskrypcję swoich treści w starszych formatach. Jedynie część umożliwia wybór pomiędzy kanałami z rodziny RSS

¹ ang. *Internet Engineering Task Force*, organizacja pracująca blisko z W3C oraz ciałami standaryzacyjnymi ISO/IEC nad rozwojem otwartych technologii używanych w Internecie.



* A data model rather than a syndication format
(serializations may be syndicated)
** Several sub-versions exist

Rysunek 4.1. Rozwój formatów publikacji kanałów wiadomości
Źródło: http://en.wikipedia.org/wiki/Syndication_format_family_tree

oraz Atom. Stawia to przed programistą skomplikowane zadanie identyfikacji formatu kanału oraz przygotowanie oddzielnego parsera dla każdego z nich.

4.9.1. Biblioteka *feedparser*

Programista języka Python ma do dyspozycji bibliotekę *feedparser*, której twórcą jest Mark Pilgrim. Jest odpowiedzią na obecną sytuację związaną z formatami subskrypcji treści. Umożliwia przetwarzanie kanałów aktywności serwisów internetowych w spójny sposób, bez konieczności bezpośredniej pracy z formatem dokumentu. Biblioteka jest dostępna na otwartej licencji MIT.

Przykład 4.11: Tworzona aplikacja przetwarza informacje dotyczące artysty. Wśród pobieranych z zewnętrznych serwisów informacji są adresy URL stron takich jak: oficjalny serwis, blog, strony prowadzone przez fanów. Każda z tych stron może zawierać dział z aktualnymi wiadomościami wraz z dedykowanym kanałem RSS/Atom. Pożądane jest, aby aplikacja utworzyła z dostępnych kanałów strumień wiadomości dotyczących danego artysty.

Dla uproszczenia przykładu (*Listing 4.17*) tworzona funkcja `update_stream` przyjmuje dwa argumenty: przechowujący pobrane wiadomości słownik `stream` oraz zawierający adres kanału string `url`. Obiekt reprezentujący strumień wiadomości tworzony jest przez wielokrotne wywołanie funkcji z tym samym słownikiem, ale różnymi adresami kanałów.

Listing 4.17. Parsowanie RSS/Atom przy użyciu biblioteki *feedparser*

```
1 import feedparser
2 from datetime import datetime
3
4 def update_stream(stream, url):
5     etag = None
6     lmod = None
7     if stream[url].has_key('etag'):
8         etag = stream[url]['etag']
9     if stream[url].has_key('lmod'):
10        lmod = stream[url]['lmod']
11
12    feed = feedparser.parse(url, etag=etag, modified=lmod)
13
14    if feed.status == 304:
15        print 'feed is up to date'
16    else:
17        stream[url]['entries'] = [{
18            'title': e.title if e.has_key('title') else None,
19            'text': e.summary if e.has_key('summary') else None,
20            'date': datetime(*e.updated_parsed[0:6]),
21            'url': e.link
22        } for e in feed.entries]
23
24        if feed.has_key('modified') and feed.modified:
25            stream[url]['lmod'] = datetime(*feed.modified[0:6])
26        if feed.has_key('etag') and feed.etag:
27            stream[url]['etag'] = feed.etag
28
29    return stream
```

Pobranie aktualnej wersji kanału wiąże się z odpytaniem zewnętrznego serwera HTTP. Twórca aplikacji powinien ograniczyć częstotliwość, z jaką sprawdzane są ewentualne aktualizacje kanałów. Niezależnie od ustawionego interwału zdarza się, że zdalny dokument nie uległ zmianie. W takim wypadku każdorazowe pobieranie posiadanej już wersji kanału stanowi niepotrzebny narzut. Metodą optymalizacji takich przypadków są nagłówki HTTP:

ETag – zawiera zazwyczaj sumę kontrolną generowaną na podstawie aktualnej zawartości dokumentu,

Last-Modified – nagłówek informujący o dacie ostatniej modyfikacji pobieranego dokumentu.

Mechanizm działania:

1. zapamiętanie wartości nagłówka przy pierwszym pobraniu dokumentu,
2. wysłanie zapamiętanej wartości przy kolejnym zapytaniu,
3. jeżeli dokument nie zmienił się, odpowiedzią będzie kod HTTP 304 (*not modified*), w przeciwnym wypadku zostanie zwrócona klasyczna odpowiedź 200 wraz z zawartością nowej wersji dokumentu, zapamiętanie nowej wartości nagłówka.

W zależności od serwisu wymienione nagłówki są stosowane równolegle lub zamiennie, dlatego dobrym rozwiązaniem jest wykorzystanie i obsługa każdego z nich niezależnie. Biblioteka *feedparser* wspiera zarówno *ETag*, jak i *Last-Modified* (*Listing 4.17*, wiersze 12, 24-27).

Obiekt `feed` zawiera sparsowany kanał wiadomości (wiersz 12). Umożliwia między innymi dostęp do elementów kanału (`feed.entries`), kodu odpowiedzi zdalnego serwera (`feed.status`) lub wymienionych nagłówków (`feed.etag`, `feed.modified`).

Omówione mechanizmy umożliwiają wysoki stopień optymalizacji: przesyłane oraz przetwarzane są jedynie kanały, które uległy zmianie.

5. Przykładowa aplikacja: MMDA

5.1. Założenia projektu

Tematem niniejszej pracy jest *programowanie i wdrażanie aplikacji sieciowych w języku Python*. Przykładowa aplikacja sieciowa nosi nazwę MMDA (ang. *Music MetaData Aggregator*). Aplikacja jest funkcjonującą weryfikacją wybranych koncepcji autora¹.

Zalety wykorzystania języka Python

Obecnie istnieje wiele dynamicznych języków wysokiego poziomu. Przykładowy projekt ilustruje dwie główne zalety języka Python:

- oszczędna i intuicyjna składnia, dzięki której tworzony jest czytelny kod,
- sprawdzone biblioteki, realizujące operacje niskiego poziomu w tle,

Czytelny kod jest podstawą, jeżeli projekt ma być rozwijany przez zespół programistów lub unowocześniany po długim okresie nieaktywności. Składnia oraz konwencje języka Python sprzyjają tworzeniu przejrzystego kodu, który często można zrozumieć bez odnoszenia się do dokumentacji.

Połączenie znanych rozwiązań z nową technologią

MMDA wykorzystuje CouchDB – bazę danych zorientowaną na dokumenty – jako alternatywę dla klasycznej, relacyjnej bazy danych. Aplikacja stanowi przykład wykorzystania biblioteki Couchdbkit, która zastępując klasyczny ORM Django umożliwia pracę z nowym typem bazy w naturalny i intuicyjny dla programisty sposób.

Otwarte standardy

Podstawą funkcjonowania Internetu jest dostępność zawartych w nim materiałów. Wykorzystanie otwartych standardów, technologii oraz formatów gwarantuje, iż z tworzonego serwisu będzie mógł skorzystać każdy, niezależnie od używanego systemu operacyjnego czy przeglądarki internetowej.

Wykorzystane programy oraz biblioteki posiadają otwarty kod źródłowy, dokumentację oraz są wydane na licencjach Wolnego Oprogramowania.

Otwarty jest więc zarówno sam kod aplikacji wraz z zależnościami, jak i generowane przez nią formaty dokumentów.

Dwa znaczenia *aplikacji sieciowej*

1) Aplikacja internetowa, czyli dostępna *w sieci* oraz 2) aplikacja wykorzystująca *sieć powiązań* pomiędzy zasobami różnych serwisów w Internecie, wchodząca z zewnętrznymi aplikacjami w interakcję.

Użyteczne i proste narzędzie

Aplikacja jest prostym agregatorem informacji dotyczących artystów oraz publikacji muzycznych. Umożliwia szybkie zestawienie obecnych w Internecie zasobów, dając ogólny obraz twórczości danej osoby lub grupy. Stanowi punkt startowy, zawierający

¹ ang. *proof of concept*

przykładowe materiały oraz odnośniki do bardziej szczegółowych lub pokrewnych zasobów.

Od strony praktycznej aplikacja powinna:

- udostępniać dynamicznie generowane strony dedykowane:
 - artystom (profil, zdjęcia, wideo, kanał wiadomości),
 - publikacjom (podstrony dedykowane każdej wersji),
 - tagom
- umożliwić łatwe wyszukiwanie tych informacji oraz linkowanie wyników,
- tworzyć czytelne, permanentnie URL,
- wykorzystywać zewnętrzne zasoby multimedialne (zdjęcia, wideo), przechowując jedynie wybrane metadane w bazie danych,
- stanowić minimalne obciążenie dla serwera,
- posiadać duże możliwości dalszego rozwoju (dzięki modularnej budowie).

Generowany przy użyciu Django serwis internetowy w warstwie prezentacji wykorzystuje następujące technologie:

HTML5

Rozwijany jako kontynuacja HTML4 oraz XHTML1 format stanowi odpowiedź na potrzeby współczesnego Internetu [36]. Wprowadza między innymi:

- dodatkowe elementy hierarchizujące zawartość dokumentu,
- natywne wsparcie dla multimediiów,
- nowe mechanizmy pracy offline oraz asynchronicznej komunikacji z serwerem,
- określone przez specyfikację mechanizmy obsługi błędów.

HTML5 jest kompatybilny wstecz, dzięki dodatkowym bibliotekom JavaScript może być obsługiwany nawet przez wiekowe przeglądarki internetowe.

Mimo wczesnego stadium rozwoju, roboczy szkic specyfikacji jest wspierany przez większość nowoczesnych przeglądarek. Praktyczne wykorzystanie formatu przez jedne z największych serwisów internetowych sugeruje, iż format ten stanie się w przyszłości dominującym. Dokumenty HTML5 posiadają uproszczony nagłówek `doctype` w postaci: `<!DOCTYPE html>`

CSS3

Przykładowa aplikacja korzysta z trzeciej iteracji *kaskadowych arkuszy stylów*¹. CSS jest językiem służącym do opisu warstwy prezentacji dokumentów HTML oraz XML. Wygląd aplikacji MMDA jest zdefiniowany przy użyciu CSS1, CSS2 oraz wybranych (obsługiwanych przez współczesne przeglądarki) elementów CSS3.

Arkusz stylów znajduje się w zewnętrznym pliku `/static/mmda.css`.

JavaScript

Przeżywający swój złoty wiek dynamiczny język skryptowy (właściwie: *ECMA-*

¹ ang. *Cascading Style Sheets, CSS*

Script). MMDA wykorzystuje napisaną w JavaScript bibliotekę jQuery¹ do generowania przyjaznego interfejsu użytkownika. Przykłady:

- dynamiczny interfejs galerii zdjęć oraz wideo,
- opóźnione ładowanie zewnętrznych grafik (*na żądanie*),
- interaktywna dyskografia artysty.

Aplikacja została napisana zgodnie z zasadą *graceful degradation*². Oznacza to, iż zachowa ona swoją podstawową funkcję nawet w przeglądarce nieobsługującej poprawnie elementów wprowadzonych w HTML5/CSS3 oraz z wyłączonym JavaScript.

5.2. Status prawny przetwarzanych informacji

Istotne jest, aby prezentowane na stronach aplikacji materiały nie łamały żadnych praw autorskich. Pożądane jest, aby zasoby te – tam gdzie jest to możliwe – były objęte otwartą licencją. Należy tu zaznaczyć, iż biblioteka wydana na otwartej licencji, może operować na mniej liberalnie licencjonowanych zasobach.

Specyfika aplikacji sprawia, iż pobierane oraz przetwarzane dane z zewnętrznych źródeł nie podlegają bezpośredniej moderacji. Innymi słowy: proces agregacji treści jest w pełni automatyczny, nie podlega żadnej ingerencji ze strony człowieka.

Poniższe zestawienie zawiera wykorzystane, zewnętrzne serwisy wraz z informacją o statusie prawnych udostępnianych przez nie zasobów.

MusicBrainz – brak ograniczeń licencyjnych, wykorzystane elementy bazy danych udostępniono w ramach domeny publicznej [25].

Last.fm – biografie artystów oraz notatki dotyczące publikacji muzycznych dostępne są w ramach podwójnej licencji:

- *Creative Commons Attribution/Share-Alike License* [37],
- *GNU Free Documentation License* [38].

Pozostałe zasoby (o ile nie zaznaczono³ inaczej) są własnością serwisu – możliwy jest bezpłatny dostęp dla projektów niekomercyjnych.

Flickr – MMDA pobiera jedynie fotografie objęte licencją umożliwiającą wykorzystanie w niekomercyjnej aplikacji [28].

YouTube – użytkownik publikujący wideo w tym serwisie zachowuje wszelkie prawa autorskie. Dodatkowo przyznaje serwisowi YouTube oraz wszystkim publikującym dany zasób serwisom ograniczoną licencję na jego publikację [39].

Wikipedia – treść artykułów objęta jest, wymienionymi już, licencjami CC-BY-SA [37] oraz GFDL [38].

¹ Lekka biblioteka ułatwiająca programowanie aplikacji po stronie przeglądarki internetowej.

² Architektura tego typu zakłada, iż w wypadku nieprzewidzianych okoliczności kluczowe elementy systemu przestają poprawnie funkcjonować jako ostatnie.

³ Przykład: zdjęcia posiadają informację o autorze oraz licencji (zgodnej z CC-BY-SA lub GFDL).

5.3. Algorytmy gromadzenia i przetwarzania informacji

5.3.1. Architektura cache

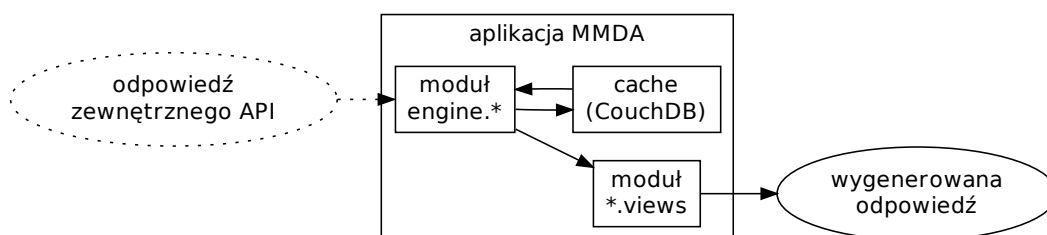
Główną funkcją aplikacji jest agregacja danych z zewnętrznych źródeł. Oczywiście optymalizacją jest przechowywanie pobranych zasobów (oraz metainformacji o zewnętrznych zasobach) w lokalnym buforze (ang. *cache*). Dzięki temu, kolejne wywołanie danej podstrony spowoduje pobranie danych z cache, odciążając zewnętrzne API i przyspieszając pracę samej aplikacji.

Biorąc pod uwagę logikę biznesową po stronie serwera, MMDA zakłada stosunkowo niską interakcję z użytkownikiem. W dużym uproszczeniu sprowadza się do realizacji następujących funkcji:

- wyszukanie artysty, publikacji lub tagu na podstawie wprowadzonego ciągu znaków,
- wyświetlenie zasobu,
- manualne odświeżenie wybranych danych.

Wyświetlenie zasobu, będąc prostą czynnością ze strony użytkownika, wiąże się z najbardziej złożoną pracą samej aplikacji. Uproszczony algorytm (*Rys. 5.1*) obsługi tego przypadku użycia, to:

1. Użytkownik wysyła żądanie HTTP zasobu określonego przez URL.
2. Aplikacja sprawdza czy wymagane informacje znajdują się w cache.
3. Jeżeli informacje nie znajdują się w cache następuje ich pobranie, przetworzenie, zapis do cache i powrót do punktu 2.
4. Użytkownik otrzymuje odpowiedź w postaci wygenerowanego dokumentu HTML.



Rysunek 5.1. Uproszczony proces generowania odpowiedzi dotyczącej zasobu przez MMDA

Źródło: opracowanie własne

Informacje dotyczące zasobów różnego typu są pobierane zawsze z cache. Architektura aplikacji zakłada automatyczne wywołanie modułów odpowiedzialnych za to, aby pożądaną informację znalazły się w buforze aplikacji. Funkcję cache pełnią dokumenty bazy CouchDB. Wysoka współbieżność, zaawansowane możliwości replikacji oraz

brak narzuconego schematu dokumentów sprawiają, iż baza danych zorientowana na dokumenty jest obiecującym medium dla tego typu zastosowań.

Aplikacja może pracować z nieograniczoną ilością baz danych. Możliwe jest wydzielanie wybranych typów dokumentów do oddzielnej bazy w celu zwiększenia czytelności, czy też delegowania ich na różne instancje CouchDB.

MMDA wykorzystuje sześć baz danych oraz wyróżnia siedem rodzajów¹ dokumentów:

CachedArtist – podstawowy dokument bazy `mmda-artists`, identyfikatorem jest MBID artysty, którego dotyczy. Zawiera informacje przybliżające sylwetkę danego wykonawcy. Stanowi najważniejszy element aplikacji: identyfikator artysty jest obecny we wszystkich typach dokumentów.

CachedReleaseGroup – pomocniczy dokument bazy `mmda-artists`, zawiera informacje dotyczące wszystkich wydań danej publikacji muzycznej. Identyfikatorem jest MBID grupy, dodatkowo MBID pojedynczych edycji używane są jako klucze w słowniku `releases`. Dokument zawiera również obowiązkowy MBID artysty w polu `artist_mbid`.

CachedArtistPictures – podstawowy dokument bazy `mmda-pictures`, identyfikatorem jest MBID artysty. Przechowuje metainformacje dotyczące zdjęć znajdujących się w zewnętrznych serwisach. Między innymi:

- URL miniaturki oraz oryginału,
- URL strony źródłowej,
- informacje o autorze.

CachedArtistVideos – podstawowy dokument bazy `mmda-videos`, identyfikatorem jest MBID artysty. Zawiera metainformacje o dotyczących danego artysty zasobach wideo:

- URL miniaturki oraz zasobu multimedialnego,
- URL źródła,
- czas trwania.

CachedArtistNews – podstawowy dokument bazy `mmda-news`, identyfikatorem jest MBID artysty. Zawiera bufor wiadomości pobranych z zewnętrznych kanałów RSS/Atom wraz z informacją o ostatnim czasie ich modyfikacji oraz sumą kontrolną zawartości.

CachedTag – podstawowy dokument bazy `mmda-tags`, identyfikatorem jest ciąg znaków składający się na nazwę tagu. Przechowuje informacje o artystach oznaczonych danym tagiem w różnych serwisach internetowych.

CachedSearchResult – przechowywany w bazie `mmda-seach` dokument zawiera wynik wyszukiwania. Identyfikatorem jest suma kontrolna znormalizowanego ciągu znaków będącego przedmiotem zapytania.

¹ Identyfikator typu dokumentu znajduje się w polu `doc_type`.

Szczegółowe mechanizmy agregacji najbardziej istotnych zasobów zostaną omówione w dalszej części pracy.

5.3.2. Artysta

Główną funkcją aplikacji jest przybliżenie sylwetki danego artysty przy użyciu dostępnych w Internecie zasobów. Ich agregacja, czyli pobranie z zewnętrznych źródeł, ma miejsce podczas pierwszego wywołania danego adresu URL. Zasada ta odnosi się do wszystkich dynamicznie generowanych treści omawianej aplikacji.

Punktem wyjścia jest strona dostępna według schematu URI: `/artist/:name/:mbid/`, gdzie `:name` to przyjazna reprezentacja nazwy artysty, a `:mbid` to jego identyfikator MBID. Zawiera ona *profil* danego wykonawcy lub zespołu, czyli informacje takie jak:

- krótką notatkę będącą skrótem biografii lub opisem wybranych prac (*abstract*),
- pełną dyskografię uporządkowaną według typu publikacji oraz daty wydania,
- powiązane URL serwisów zewnętrznych,
- relacje z innymi wykonawcami
 - lista członków w wypadku zespołów,
 - przynależność do zespołów w wypadku osób
- przypisane tagi,
- listę pokrewnych/podobnych artystów.

Profil artysty zawiera odnośniki do omówionych w kolejnych podrozdziałach stron publikacji, multimediiów oraz kanału wiadomości.

Podstawowym źródłem danych jest serwis MusicBrainz. Zakończona sukcesem agregacja informacji dotyczących danego artysty z tego serwisu (1) powoduje uruchomienie kolejnych zapytań do pozostałych API (2-4). Po odebraniu ostatniej odpowiedzi (lub przekroczeniu określonego limitu czasu) następuje przetwarzanie pobranych danych oraz ich zapis w postaci dokumentu **CachedArtist** (*Rys. 5.2*). Tam gdzie jest to logiczne, ma miejsce równoległe wywołanie kilku zapytań w celu przyspieszenia procesu.

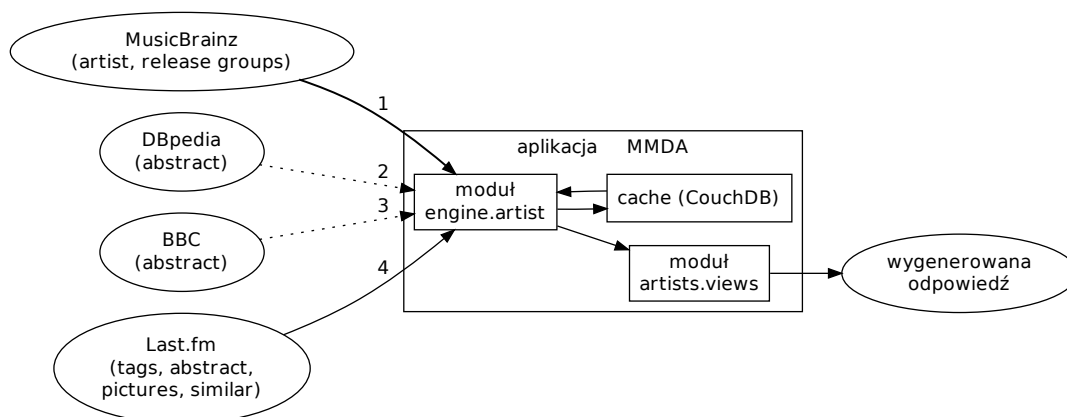
Z uwagi na relacje z pozostałymi elementami aplikacji, część agregowanych przy okazji wywołania profilu artysty treści jest wykorzystywana przy prezentacji innych zasobów:

- podstawowe informacje o publikacjach muzycznych (typ, tytuł, data wydania) są zapisywane jako podstawowe wersje **CachedReleaseGroup**,
- pobrane zdjęcia są przechowywane w dokumentach **CachedArtistPictures**.

5.3.3. Publikacja

URI `/artist/:name/release/:title/:mbid/` określa stronę publikacji muzycznej, gdzie `:name` jest reprezentacją nazwy autora publikacji, `:title` jej tytułu, natomiast `:mbid` identyfikatorem MBID danej wersji publikacji.

Publikacja może być albumem, singlem, kompilacją. MMDA prezentuje jedynie oficjalne wydania, sygnowane przez artystę lub jego wytwórnię muzyczną. Proces genero-

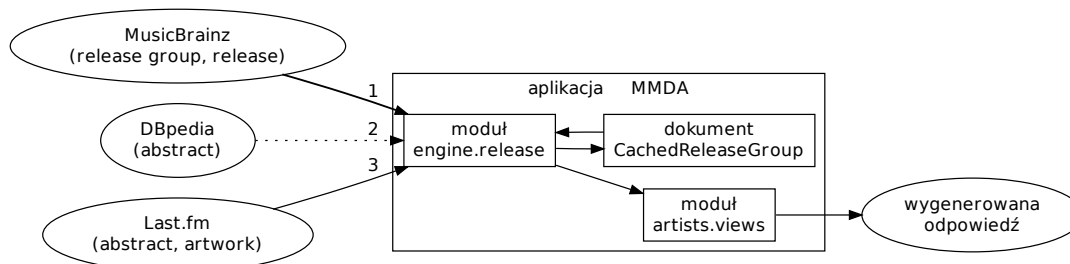


Rysunek 5.2. Schemat agregacji treści wykorzystywanych przez profil artysty

Źródło: opracowanie własne

wania odpowiedzi dotyczącej zasobu publikacji sprowadza się do weryfikacji obecności grupy publikacji zawierającej wersję określoną przez MBID. Podstawowe informacje o dyskografii pobierane są wraz z informacjami dotyczącymi danego artysty (*Rys. 5.2*) i zapisywane w postaci dokumentów `CachedReleaseGroup`. Umożliwia to prezentację dyskografii artysty na stronie jego profilu, bez obecności szczegółowych informacji o publikacjach w bazie. Jeżeli cache nie zawiera macierzystej grupy publikacji¹, następuje pobranie brakujących danych (również artysty, jeżeli nie jest obecny w cache).

W zależności od danych obecnych w `CachedReleaseGroup` aplikacja pobiera brakujące, wymagane przez stronę szczegółowe informacje dotyczące publikacji (*Rys. 5.3*).



Rysunek 5.3. Schemat agregacji treści wykorzystywanych przez stronę publikacji

Źródło: opracowanie własne

Między innymi:

- szczegółową listę utworów,
- krótką notatkę opisującą publikację,
- powiązane URL do serwisów zewnętrznych,

¹ Przykład: wywołanie strony publikacji bez uprzedniej wizyty na profilu artysty.

- relacje z artystami,
- relacje z innymi publikacjami
 - reedycje, wersje zremasterowane,
 - inne wersje tej samej publikacji,
 - kolejne/poprzednie płyty danego zestawu
- grafikę, zazwyczaj obecną na okładce płyty CD (ang. *artwork*).

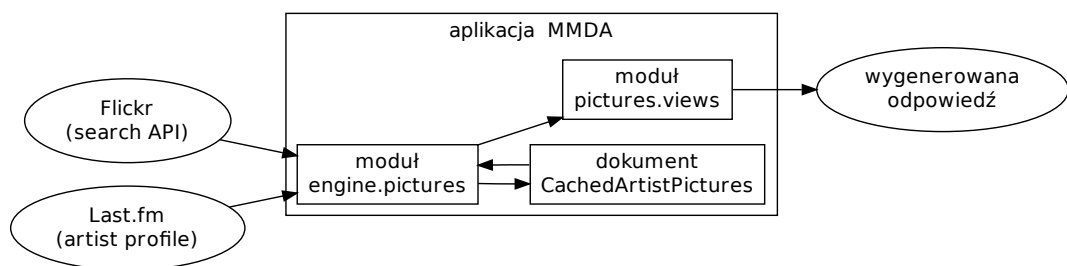
5.3.4. Multimedia

Zasoby multimedialne związane z danym artystą to zdjęcia oraz materiały wideo. Ich identyfikatory URI skonstruowane są według schematu:

```
/artist/:name/pictures/:mbid/  
/artist/:name/videos/:mbid/
```

gdzie `:mbid` jest identyfikatorem MBID artysty, którego dotyczą.

Informacje dotyczące grafik agregowane są równoległe z zewnętrznymi, wzajemnie uzupełniających się serwisów (*Rys. 5.4*). Serwis Last.fm zawiera zazwyczaj profesjonalne zdjęcia studyjne dodane przez użytkowników, samych artystów lub wydawnictwa muzyczne. Dane pobierane są przy użyciu identyfikatora MBID. Flickr jest miejscem, w którym publikowane są duże ilości zdjęć z koncertów. Niestety mechanizm wyszukiwania w serwisie nie wspiera MBID. Wykorzystana jest nazwa wykonawcy, stąd w niektórych przypadkach wyniki mogą być błędne.

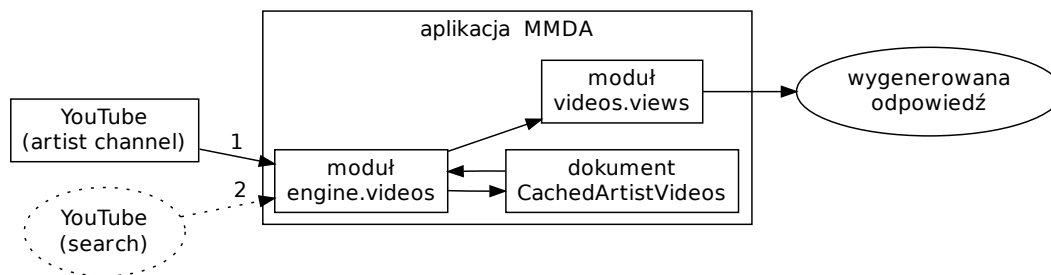


Rysunek 5.4. Schemat agregacji zdjęć w MMDA

Źródło: opracowanie własne

Wszystkie prezentowane przez aplikację zdjęcia zawierają odnośniki do stron źródłowych oraz informację o autorze. Wygenerowana galeria zdjęć umożliwia szybką nawigację przy użyciu interfejsu opartego o JavaScript.

Mechanizm tworzenia galerii klipów wideo (*Rys. 5.5*) jest uzależniony od tego, czy pośród obecnych w cache zasobów dotyczących artysty znajduje się informacja o *oficjalnym kanale YouTube*. Jeżeli aplikacja dysponuje URL kanału, pobrana oraz prezentowana jest jego zawartość (1). Brak tej informacji powoduje wykorzystanie najlepszych wyników wyszukiwania (2) jako zastępczego kanału wideo.



Rysunek 5.5. Schemat agregacji wideo w MMDA

Źródło: opracowanie własne

Należy zaznaczyć, iż zarówno zdjęcia, jak i materiały wideo są przechowywane przez zewnętrzne serwisy. Aplikacja udostępnia jedynie czytelny interfejs służący do wyszukiwania oraz przeglądaniach tych zasobów – nie kopiuje zdjęć czy też plików wideo na serwer, na którym jest uruchomiona. Zalety tego rozwiązania:

- wszelka moderacja niepożądanych treści odbywa się poza serwisem,
- niskie obciążenie serwera oraz łącza (przesyłanie jedynie lekkich dokumentów tekstowych),
- hostingiem multimediiów zajmują się serwisy posiadające przystosowaną do tego infrastrukturę.

5.3.5. Kanał wiadomości

Pośród pobranych z zewnętrznych serwisów informacji znajdują się adresy URL powiązanych z artystą stron internetowych. Niektóre z nich: blog, strona oficjalna, strona prowadzona przez fanów, generują przetwarzany przez MMDA strumień wiadomości.

Kanał wiadomości artysty określonego przez MBID o wartości `:mbid` dostępny jest w formacie strony HTML:

`/artist/:name/news/:mbid/`

oraz równoważnego kanału Atom:

`/artist/:name/news/:mbid/feed/`

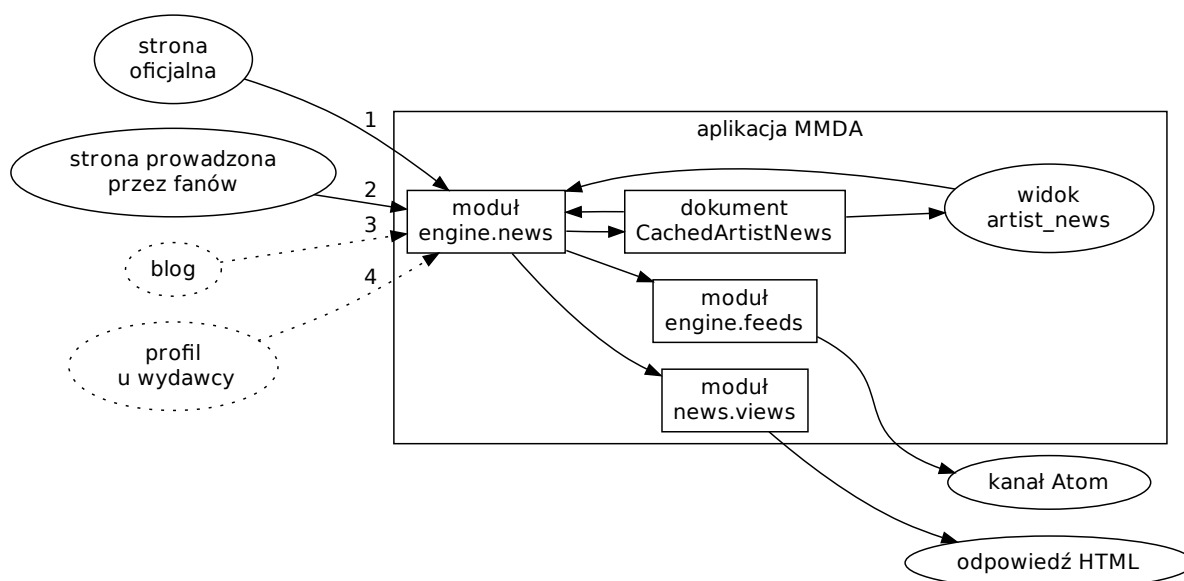
Najbardziej istotne cechy tworzonego przez MMDA kanału wiadomości:

- lista źródeł generowana na podstawie powiązanych z artystą URL,
- inteligentne cache wykorzystujące nagłówki HTTP do optymalizacji procesu aktualizacji kanałów,
- pełna bezobsługowość: aplikacja automatycznie wykrywa nowe źródła wiadomości i dodaje je do generowanego kanału.

Źródłem wiadomości jest kanał RSS/Atom określony przez obecny w dokumencie HTML tag `<link>`. MMDA pobiera i skanuje wymienione strony internetowe w poszu-

kiwaniu tego typu odnośników. Stosuje przy tym inteligentną technikę, która przerywa dalsze pobieranie dokumentu w chwili odnalezienia poszukiwanego wyrażeniem regularnym ciągu znaków – adresu URL kanału RSS/Atom. Znalezione kanały wiadomości są pobierane i parsowane przez opisaną w rozdziale 4.9.1 bibliotekę *feedparser*.

Adresy URL wykrytych kanałów (1-2) są przechowywane w dokumencie *CachedArtistNews* jako klucze słownika zawierającego treść pobranych z nich wiadomości (*Rys. 5.6*). Uporządkowany chronologicznie strumień wiadomości generowany jest przez widok bazy CouchDB o nazwie *artist_news*. Jeżeli w przyszłości pojawią się nowe źródła (3-4), aplikacja automatycznie (po wygaśnięciu okresu retencji cache) włączy je w proces aktualizacji *CachedArtistNews*.



Rysunek 5.6. Schemat tworzenia kanału wiadomości w MMDA

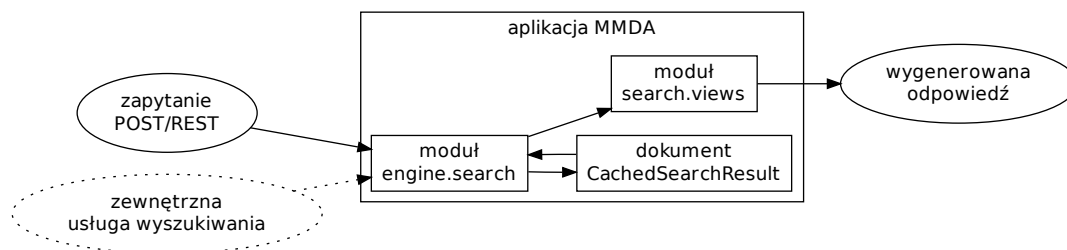
Źródło: opracowanie własne

5.3.6. Wyszukiwanie

MMDA udostępnia mechanizm wyszukiwania w postaci formularza wysyłanego metodą POST lub prostego interfejsu REST. Umożliwia wyszukiwanie:

- artystów (po nazwie lub jednym z aliasów),
- publikacji muzycznych (po tytule publikacji),
- tagów (zwracani są określani nimi artyści).

Wysoką elastyczność oraz jakość wyników gwarantuje wykorzystanie zewnętrznych mechanizmów wyszukiwania (*Rys. 5.7*). Artyści oraz publikacje muzyczne obecne w bazie MusicBrainz indeksowane są przez wydajny silnik *Lucene*, obsługujący wykorzystywane przez MMDA wyszukiwanie otwartotekstowe w ramach *MusicBrainz XML WebService 1.0*. Za wyszukiwanie tagów odpowiada API serwisu Last.fm.



Rysunek 5.7. Mechanizm wyszukiwania w MMDA

Źródło: opracowanie własne

Interfejs REST może być wykorzystany do wykonywania szybkich zapytań z poziomu przeglądarki internetowej. Zapytanie tego typu przyjmuje postać URI:

`/search/:query_type/:query/`

Segment `:query_type` przyjmuje wartość `artist`, `release` lub `tag`, natomiast `:query` jest ciągiem znaków będącym przedmiotem wyszukiwania.

Niezależnie czy zapytanie zostało wywołane przy użyciu POST czy REST, aplikacja zwraca przekierowanie na stronę wyniku:

`/search/:query_type/:query/:hash/`

Segment `:hash` jest identyfikatorem dokumentu typu `CachedArtistNews`, w którym MMDA przechowuje wyniki zawarte w odpowiedzi zwróconej przez zewnętrzne API.

Rozwiązanie to gwarantuje:

- permanentne linki dla strony wyników wyszukiwania,
- odciążenie zewnętrznych serwisów.

5.3.7. Możliwości rozwoju

Dla zachowania przejrzystości MMDA wykorzystuje ograniczoną ilość API. Aplikacja jest demonstracją pewnego zestawu koncepcji oraz technologii. Należy jednak zaznaczyć, iż została zaprojektowana w sposób umożliwiający rozszerzenie jej funkcji niskim nakładem prac. Obecnie wdrożona, może posłużyć jako proste, użyteczne narzędzie lub podstawa, na bazie której stworzona zostanie bardziej rozbudowana aplikacja. Przykładowymi ulepszeniami MMDA mogą być:

- alternatywne źródła tagów, zdjęć oraz wideo,
- dodatkowe źródła notatki *abstract* – w tym dla tagów,
- możliwość pobrania lub odtwarzania muzyki wydanej na licencjach *Creative Commons*,
- wykorzystanie tagów HTML `<video>` oraz `<audio>` (gdy HTML5 osiągnie większą popularność),
- utworzenie opartego o XML lub JSON API dla zasobów obecnych w cache.

5.4. Obsługa przez użytkownika

Na stronie dostępnej pod adresem `http://music.local` znajduje się formularz umożliwiający określenie kryteriów wyszukiwania artysty, publikacji lub tagu. Jest on zawsze obecny w nagłówku, niezależnie od przeglądanego przez użytkownika podstrony.

Wpisanie przykładowej frazy *graaf generator* i uruchomienie wyszukiwania spełniających to kryterium artystów, spowoduje wyświetlenie ekranu zawierającego nazwy wykonawców wraz z wartością określającą trafność wyniku (*Rys. A.1*, strona 96).

Kliknięcie na najlepszym z wyników przenosi użytkownika na stronę zawierającą *profil* artysty (*Rys. A.2*). Strona ta posiada kilka sekcji, porządkujących zebrane informacje:

Group/Person – informacje ułatwiające identyfikację grupy lub osoby:

- data urodzenia/założenia, śmierci/rozwiązania,
- alternatywna, często spotykana (np. błędna) pisownia nazwy artysty,
- krótka charakterystyka (zawsze obecna, gdy istnieje wielu artystów o tej samej nazwie).

Abstract – krótka notatka zawierająca podstawowe informacje o artyście, dokonania, wybrane wydarzenia.

Discography – sekcja zawiera dyskografię artysty (oficjalne, autoryzowane publikacje muzyczne). Lista jest uporządkowana chronologicznie. Możliwe jest filtrowanie widocznych publikacji według typu (albumy, single, kompilacje).

Artist/Group related websites – powiązane z artystą adresy URL zewnętrznych zasobów, dostępnych w Internecie.

Relationships – relacje artysty z innymi wykonawcami lub grupami muzycznymi. Jeżeli przeglądany artysta jest osobą, znajdują się tu informacje o członkostwie w zespołach. W przypadku grupy muzycznej widoczni są jej członkowie. Dodatkowo sekcja zawiera informacje o projektach będących efektem współpracy pomiędzy niezależnymi artystami.

Folksonomy – tak zwana *chmura tagów*, czyli graficzna reprezentacja słów, jakimi dany artysta został określony przez użytkowników różnych serwisów internetowych. Każdy tag jest linkiem do strony zawierającej listę najpopularniejszych wykonawców określonych danym słowem (*Rys. A.8*).

Similar artists – lista pokrewnych artystów. Mogą to być osoby związane z projektem (wymienione w *Relationships*) lub zupełnie niezależni artyści wykonujący muzykę tego samego gatunku.

W górnej części strony, poniżej nagłówka zawierającego nazwę artysty znajduje się menu akcji. Kliknięcie na każdej z pozycji powoduje przejście do podstrony zawierającej dodatkowe zasoby o aktualnym artyście. Oprócz omówionego *profilu*, dostępne są:

Pictures – grafiki przedstawiające osobę lub zespół, agregowane z serwisów Last.fm oraz Flickr (*Rys. A.3*). Kliknięcie na miniaturze powoduje uruchomienie przeglądar-

ki zdjęć opartej o jQuery. Umożliwia one płynną nawigację przy użyciu przycisków nawigacyjnych lub strzałek kierunkowych na klawiaturze oraz przejście do strony źródłowej.

Videos – galeria krótkich filmów wideo udostępnia najbardziej trafne pozycje opublikowane w serwisie YouTube (*Rys. A.4*). Jej obsługa jest analogiczna do *pictures*.

News – zawiera dynamicznie generowaną listę wiadomości pobieranych z dotyczących danego artysty serwisów. Nawet gdy jest on pusty (*Rys. A.5*), warto zasubskrybować dostępny kanał Atom. Gdy pojawią się nowe źródła, kanał zostanie automatycznie wypełniony (również wersja HTML) (*Rys. A.7*).

Refresh – jeżeli informacje dotyczące danego artysty są niekompletne, użytkownik może je uzupełnić. Na przykład dodać brakującą publikację muzyczną do serwisu MusicBrainz. Aby nowe informacje pojawiły się w MMDA należy odświeżyć oparte o CouchDB cache aplikacji, czemu służy niniejsza strona. Umożliwia określenie odświeżanych zasobów oraz podgląd aktualnego stanu cache, włącznie z datą ostatniej agregacji danych z zewnętrznych serwisów (*Rys. A.6*). Operacje ingerujące w bazę danych wymagają rozwiązania testu CAPTCHA.

Kliknięcie nazwy publikacji muzycznej powoduje przejście na poświęconą jej podstronę (*Rys. A.9*). Nagłówek w postaci tytułu oraz autora może posiadać okładkę (ang. *artwork*). Dalsza część strony zawiera następujące sekcje:

Track listing – uporządkowana lista utworów muzycznych wraz z czasem ich trwania,

Release events – informacje o dostępnych wydaniach danej publikacji: data, kraj, rodzaj nośnika, wydawnictwo,

Release related websites – powiązane z publikacją muzyczną adresy URL zewnętrznych, dostępnych w Internecie zasobów,

Credits – szczegółowe informacje o wkładzie wymienionych twórców w proces powstawania prezentowanej publikacji, wraz z odnośnikami do ich *profili*,

Alternative track listings – odnośniki do alternatywnych wydań danej publikacji muzycznej (np. różniących się listą utworów),

Relations with other releases – relacje z innymi wydaniem/publikacjami. Mogą to być kolejne/poprzednie nośniki z wielopłytkowego wydania, zremasterowana (odświeżona) wersja tej samej publikacji lub płyta zawierająca remiksy utworów.

Internacjonalizacja Należy zaznaczyć, iż aplikacja jest w stanie prezentować profile artystów (jak i publikacji) z całego świata, niezależnie od alfabetu w którym zapisana jest ich nazwa lub tytuł. Na szczególną uwagę zasługuje obsługa transkrypcji znaków

Unicode do ASCII przy użyciu biblioteki *unidecode* [32], wykorzystana przy tworzeniu czytelnych adresów URL (*Rys. A.10*).

Szybkie wyszukiwanie Użytecznym rozwiązaniem jest wywoływanie wyszukiwania bezpośrednio z wbudowanego w przeglądarkę formularza lub paska adresu¹.

Wystarczy skonfigurować przeglądarkę do pracy z URL:

```
http://music.local/search/artist/%s/
```

gdzie `%s` jest poszukiwanym ciągiem znaków a `artist` rodzajem poszukiwanego zasobu².

¹ Przykłady: *AwesomeBar* (Mozilla Firefox), *Omnibar* (Google Chrome/Chromium).

² Pozostałe dostępne typy zasobów to `release` (publikacja muzyczna) oraz `tag`.

6. Wdrożenie aplikacji

Niniejszy rozdział przedstawia wybrane techniki stosowane przy wdrożeniach aplikacji napisanych w języku Python. Część z nich jest uniwersalna (Nginx, Memcached) i może być wykorzystana w kontekście alternatywnych dla Pythona/Django rozwiązań.

Wdrożenie aplikacji nie ogranicza się jedynie do wyłączenia procedur odpowiedzialnych za wykorzystane podczas rozwoju aplikacji debugowanie.

Kwestie, które należy rozważyć przed oddaniem aplikacji do użytku:

- rodzaj serwera, na którym zainstalowane jest oprogramowanie,
- wymagania aplikacji co do sprzętu oraz łącza,
- ograniczenie dostępu do bazy danych oraz prawa dostępu samej aplikacji,
- mechanizm instalacji oraz aktualizacji wykorzystanego oprogramowania.
- sytuacje wyjątkowe: awaria sprzętu, duże obciążenie łącza, utrata danych.

Przykładowe wdrożenie opiera się o serwer dedykowany pracujący pod kontrolą systemu *Gentoo GNU/Linux*. W dalszej części rozdziału przyjęto następujące założenia:

Użytkownik systemu operacyjnego: `user`

Ścieżka projektu Django: `/home/user/mmda/`

URL bazy CouchDB: `http://127.0.0.1:5984`

URL wdrożonej aplikacji MMDA: `http://music.local`¹

URL instancji Memcached: `http://127.0.0.1:11211`

URL instancji Unicorn: `http://127.0.0.1:8000`

W poprzednich rozdziałach terminem *cache* – w kontekście tworzonej aplikacji – określone były dokumenty bazy CouchDB. W tym rozdziale słowo *cache* odnosi się do omówionego w dalszej części pracy rozwiązania opartego o usługę Memcached – chyba, że wyraźnie zaznaczono inaczej.

Wykonywane przez użytkownika `user` polecenia powłoki poprzedzane są znakiem `$`, natomiast wydane z konta `root` poprzedzone są `#`.

6.1. Środowisko produkcyjne

Środowisko produkcyjne różni się od programistycznego zarówno stopniem złożoności, jak i wymogami bezpieczeństwa. Niżej wymienione zagadnienia omawiają proces tworzenia środowiska produkcyjnego dla aplikacji MMDA.

Django Włączenie trybu produkcyjnego po stronie frameworka Django sprowadza się to ustawienia wartości `DEBUG=False` w pliku `settings.py` w katalogu projektu. Obecne w MMDA mechanizmy informujące o szczegółowej aktywności aplikacji są uzależnione od wartości tej zmiennej. Dodatkowo należy uaktualnić (w tym samym pliku) ścieżkę katalogu zawierającego szablony:

¹ Przykładowy host, na potrzeby testów wskazujący na `127.0.0.1` lub zdalny serwer.

```
TEMPLATE_DIRS = (  
    '/home/user/mmda/templates',  
)
```

Virtualenv Interpreter języka Python wykorzystuje zasoby dostępne w ścieżkach zdefiniowanych obiektem `sys.path`. Zazwyczaj wskazuje on na współdzielone przez wszystkich użytkowników systemu repozytorium pakietów języka Python, gdzie domyślnie instalowane są dodatkowe biblioteki. Nie zawsze jest to pożądane zachowanie. Lepszą polityką bywa utworzenie lokalnego środowiska, które zapewni wymagany przez daną aplikację poziom izolacji.

Do wymiernych korzyści tego rozwiązania należy zaliczyć:

elastyczność – możliwość uruchamiania kilku aplikacji korzystających z różnych wersji tych samych bibliotek (niezależnie od wersji zainstalowanej w systemie),

zwiększenie stabilności – aktualizacja systemu operacyjnego, bibliotek języka Python nie mają wpływu na działanie aplikacji,

zwiększenie bezpieczeństwa – instalacja bibliotek nie wymaga praw administratora, nie wpływa również na działanie innych aplikacji oraz systemu operacyjnego.

Narzędziem automatyzującym proces tworzenia wirtualnego i izolowanego środowiska dla interpretera języka Python jest *Virtualenv* [40]. Umożliwia kontrolę nad poziomem izolacji oraz wspiera instalację bibliotek przy użyciu pakietu *Setuptools*¹ (oraz *Distribute*²), dzięki czemu jest niezależny od menedżera pakietów systemu operacyjnego.

Utworzenie środowiska w katalogu ENV:

```
$ cd /home/user/  
$ virtualenv --distribute ENV
```

Nowe środowisko można aktywować dla wszystkich przyszłych wywołań interpretera w ramach danej powłoki poleceniem:

```
$ source ~/ENV/bin/activate
```

lub wywołując izolowany interpreter (lub zainstalowane później narzędzia) bezpośrednio:

```
$ ENV/bin/python  
$ ENV/bin/easy_install  
$ ENV/bin/gunicorn
```

Instalacja wykorzystanych w aplikacji MMDA zewnętrznych bibliotek³ oraz narzędzi sprowadza się do wydania serii poleceń `easy_install`. Zdarza się, iż aplikacja

¹ Niezależna od systemu operacyjnego biblioteka ułatwiająca instalację oraz zarządzanie pakietami języka Python.

² Rozwojowy fork *Setuptools* wspierający m.in. język Python 3.

³ W czasie wdrożenia aplikacji, biblioteka *python-musicbrainz2* w wymaganej wersji `>= 0.7.1` była już dostępna poprzez *Setuptools*.

działa poprawnie jedynie z wybranymi wersjami bibliotek. Domyślnie komendy instalują najnowsze wersje pakietów. Instalacja konkretnego wydania jest możliwa przy użyciu składni `pakiet==wersja`¹. Dobrą praktyką jest instalacja dokładnie tych wersji, z którymi aplikacja została przetestowana:

```
$ cd ~
$ ENV/bin/easy_install django==1.1.1
$ ENV/bin/easy_install python-memcached==1.45
$ ENV/bin/easy_install python-musicbrainz2==0.7.2
$ ENV/bin/easy_install couchdbkit==0.4.6
$ ENV/bin/easy_install beautifulsoup==3.0.8
$ ENV/bin/easy_install pylast==0.4.26
$ ENV/bin/easy_install surf.sparql_protocol==1.0.0
$ ENV/bin/easy_install python-cjson==1.0.5
$ ENV/bin/easy_install flickrapi==1.4.2
$ ENV/bin/easy_install gdata==2.0.9
$ ENV/bin/easy_install feedparser==4.1
$ ENV/bin/easy_install unicode==0.04.1
$ ENV/bin/easy_install recaptcha-client==1.0.5
$ ENV/bin/easy_install gunicorn==0.9.1
```

Każde wywołanie polecenia `easy_install` uruchamia proces instalacji wybranej biblioteki wraz z jej zależnościami – wszystko odbywa się wewnątrz izolowanego środowiska ENV. O pozytywnym zakończeniu instalacji informuje komunikat (przykład dla Gunicorn):

```
...
Installed /home/user/ENV/lib/python2.6/site-packages/gunicorn-0.9.1-py2.6.egg
Processing dependencies for gunicorn
Finished processing dependencies for gunicorn
```

6.2. Konfiguracja serwera

Przed rozpoczęciem instalacji należy określić, z jakimi modułami Nginx ma zostać skompilowany. W Gentoo wystarczy do pliku `/etc/make.conf` dodać linię:

```
NGINX_MODULES_HTTP="gzip perl fastcgi pcre rewrite access
                    auth_basic autoindex proxy memcached"
```

Instalację serwera Nginx, bazy CouchDB (wraz z dodaniem usług do skryptów startowych), (wykorzystanego w dalszej części pracy) Memcached oraz Virtualenv przy użyciu domyślnego managera dystrybucji realizują polecenia:

¹ Starsze wersje pakietów mogą wymagać podania URL źródeł przy użyciu parametru `-f`.

```
# emerge www-servers/nginx dev-db/couchdb dev-python/virtualenv \
    net-misc/memcached
...
# rc-update add memcached default
# rc-update add nginx default
# rc-update add couchdb default
```

Konfiguracja serwera Nginx polega na dodaniu do pliku `/etc/nginx/nginx.conf` dyrektywy `include`, importującej ustawienia omawianej aplikacji sieciowej (*Listing 6.2*) z wskazanego pliku :

```
http {
    ...
    include /etc/nginx/vhosts/music.local;
}
```

Uruchomienie zainstalowanych usług:

```
# /etc/init.d/memcached start
# /etc/init.d/nginx start
# /etc/init.d/couchdb start
```

Utworzenie wykorzystywanych przez aplikację baz danych:

```
$ cd ~/mmda
$ ENV/bin/python manage.py syncdb
```

6.3. Bezpieczeństwo

Przykładowe wdrożenie odbywa się na serwerze dedykowanym, do którego dostęp mają jedynie zaufane osoby przy użyciu protokołu SSH. Jedynym widocznym z zewnątrz portem związanym z funkcjonowaniem aplikacji jest port 80. Wszystkie wykorzystane usługi oparte o protokół HTTP nasłuchują na lokalnym adresie 127.0.0.1, dodatkowo serwer posiada firewall przepuszczający jedynie port SSH oraz HTTP (80, obsługiwany przez Nginx).

Każda z usług uruchomiona jest z prawami innego użytkownika, zapewniając dodatkowy poziom bezpieczeństwa (*Tabela 6.1*).

Tabela 6.1. Izolacja procesów wykorzystywanych w aplikacji
(Źródło: opracowanie własne)

usługa	CouchDB	Nginx	Gunicorn	Memcached
użytkownik	couchdb	nginx	user	memcached

CouchDB Instancja bazy CouchDB nie wymaga konfiguracji – znajduje się na tej samej maszynie co uruchomiona aplikacja. Z uwagi na ograniczony dostęp dodatkowe

mechanizmy uwierzytelniania nie są wymagane. Jeżeli w przyszłości pojawi się potrzeba bezpośredniego dostępu z zewnątrz, serwer Nginx może służyć za pośrednika oferującego metody autoryzacji.

Memcached Po instalacji usługa domyślnie nasłuchuje na wszystkich aktywnych interfejsach sieciowych (0.0.0.0). Mimo, iż niechciany ruch jest filtrowany przez firewall, warto ograniczyć zasięg usługi definiując lokalny adres w `/etc/conf.d/memcached`:

```
LISTENON="127.0.0.1"
```

6.4. Optymalizacja wydajności

Przykładowa aplikacja jest stosunkowo wydajna, jednak zawsze istnieje możliwość dalszej optymalizacji. Należy również brać pod uwagę możliwość nagłego wzrostu obciążenia serwisu – większości problemów najłatwiej jest zapobiec na etapie wdrożenia.

6.4.1. Gunicorn

Domyślny interfejs FastCGI zazwyczaj spełnia swoje zadanie, jednak bardziej wydajną (dla napisanych w języku Python aplikacji) alternatywą jest *WSGI*¹. Gunicorn jest serwerem WSGI oraz load-balancerem, wykorzystanym w przykładowej aplikacji z uwagi na bardzo dobre wsparcie dla aplikacji napisanych przy użyciu frameworka Django oraz współpracę z serwerem Nginx [41].

Do jego funkcji należą:

- bardzo dobra integracja z różnymi frameworkami (*Django*, *Pylons*, *TurboGears2*),
- automatyczne forkowanie procesów (ang. *workers*) w zależności od potrzeb,
- wykorzystanie Nginx jako bufora obsługującego powolnych klientów,
- zwiększenie współbieżności aplikacji niskim kosztem.

Za automatyczne uruchomienie Gunicorn przy starcie systemu odpowiada skrypt startowy² `/etc/init.d/gunicorn.mmda` (*Listing 6.1*).

Konfiguracja skryptu zawiera szczegóły dotyczące obsługiwanej aplikacji oraz użytkownika, z którego prawami instancje serwera mają być uruchamiane. Gunicorn domyślnie działa na 8000 porcie lokalnego hosta (*Rys. 6.1*). Możliwe jest również wykorzystanie gniazd systemów UNIX.

Listing 6.1. Skrypt startowy serwera Gunicorn dla dystrybucji Gentoo

```
1 #!/sbin/runscript
2 EXEC="/home/user/ENV/bin/gunicorn_django"
3 APP="/home/user/mmda"
4 NAME="mmda"
```

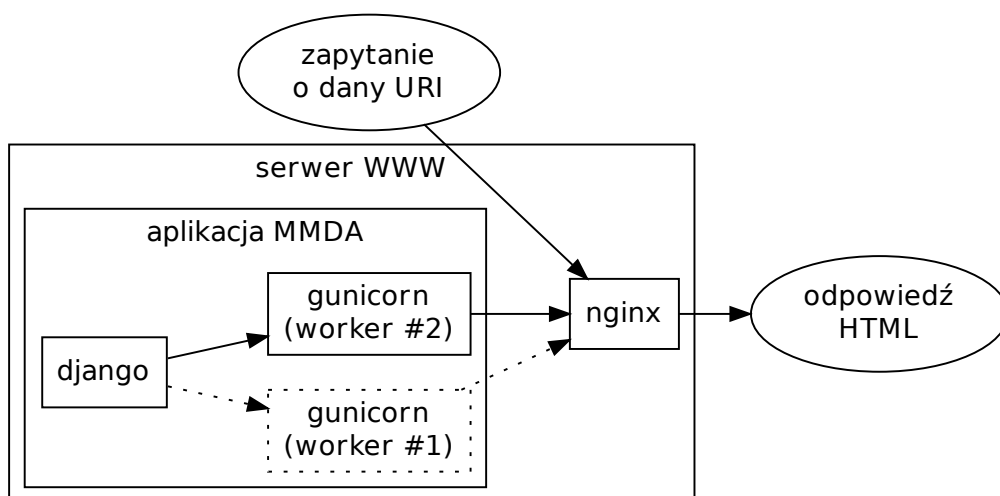
¹ ang. *Web Server Gateway Interface*, uniwersalny interfejs komunikacji pomiędzy serwerami HTTP a aplikacjami napisanymi w języku Python.

² Zgodnie z konwencją dystrybucji Gentoo, konfiguracja skryptów startowych umieszcza się w katalogu `/etc/conf.d`. Dla zwiększenia czytelności przykładowy skrypt zawiera konfigurację w sobie.

```

5 USER="user"
6 PID_FILE="/tmp/gunicorn_${NAME}.pid"
7
8 depend() {
9     need net nginx memcached couchdb
10 }
11 start() {
12     ebegin "Starting Gunicorn for '${NAME}' app"
13     start-stop-daemon --start --exec ${EXEC} --chdir ${APP} \
14         --chuid ${USER} -- --daemon --user=${USER} --pid=${PID_FILE} \
15         --workers=3
16     eend $? "Failed to start Gunicorn.${NAME}"
17 }
18 stop() {
19     ebegin "Stopping Gunicorn for '${NAME}' app"
20     kill -QUIT `cat ${PID_FILE}`
21     eend $? "Failed to stop Gunicorn.${NAME}"
22 }

```



Rysunek 6.1. Gunicorn jako serwer WSGI dla aplikacji Django
Źródło: opracowanie własne

Skrypt jest wykonywany oraz dodawany do domyślnego poziomu uruchomieniowego systemu w standardowy sposób:

```

# chmod +x /etc/init.d/gunicorn.mmda
# /etc/init.d/gunicorn.mmda start
# rc-update add gunicorn.mmda default

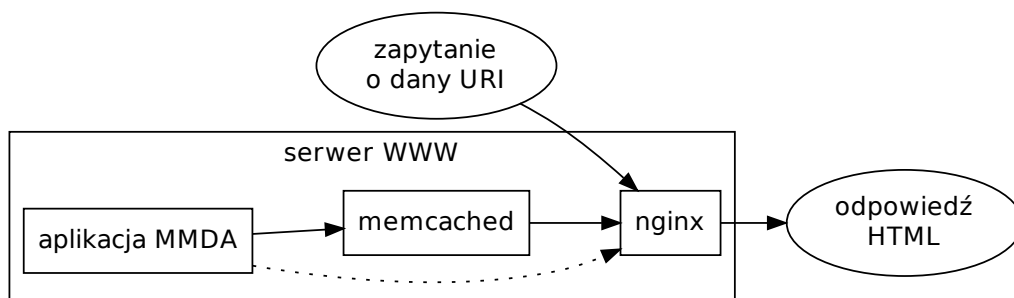
```

6.4.2. Memcached

Dynamicznie generowane przez MMDA strony za każdym razem pobierają i przetwarzają dane z dokumentów lub widoków bazy CouchDB. Wykorzystując *Memcached* jako silnik cache Django wraz z modułem dla serwera Nginx, możliwe jest znaczne odciążenie aplikacji poprzez przechowywanie wygenerowanych dokumentów w pamięci. Rozwiązanie jest wykorzystywane głównie przez serwisy oparte o generowanie i przetwarzanie dynamicznie zmienianych treści¹. Umożliwia zagospodarowanie nieużywanej pamięci operacyjnej, dając duży wzrost wydajności obsługiwanych serwisów internetowych [42].

Mechanizm działania dodatkowej warstwy cache (*Rys. 6.2*) sprowadza się do:

1. odebrania przez Nginx zapytania zasobu określonego danym URI,
2. próby pobrania odpowiedzi HTML z Memcached przy użyciu klucza `mmda:URI`, gdzie URI jest wywoływającym adresem,
3. jeżeli instancja Memcached nie zawiera wartości pod wskazanym kluczem, Nginx przekazuje zapytanie o dany URI do aplikacji Django,
4. Django generuje odpowiedź i przed przekazaniem do Nginx zapisuje ją w Memcached.



Rysunek 6.2. Memcached jako cache dla odpowiedzi HTML

Źródło: opracowanie własne

Za komunikację serwera Nginx z Memcached odpowiada dedykowany moduł wykorzystany przez konfigurację wdrażanej aplikacji (*Listing 6.2*, wiersz 21).

Listing 6.2. Konfiguracja Nginx do pracy z MMDA

```

1 upstream gunicorn {
2     server 127.0.0.1:8000;
3 }
4 server {
5     listen          music.local:80;
```

¹ Do serwisów wykorzystujących *Memcached* należą m.in. *Digg*, *Wordpress*, *YouTube*, *Flickr*, *Twitter*.


```
6      server_name      music.local;
7      access_log       /home/user/log/music.local.access.log main;
8      error_log        /home/user/log/music.local.error.log info;
9
10     location /static/ {
11         root /home/user/mmda;
12         break;
13     }
14     location / {
15         if ($request_method = POST) {
16             proxy_pass http://gunicorn;
17             break;
18         }
19         default_type    "text/html; charset=utf-8";
20         set $memcached_key "mmda:$uri";
21         memcached_pass   127.0.0.1:11211;
22         error_page 404 502 = @cache_miss;
23     }
24     location @cache_miss {
25         proxy_set_header    X-Real-IP    $remote_addr;
26         proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
27         proxy_set_header    Host $http_host;
28         proxy_redirect      off;
29         proxy_pass           http://gunicorn;
30     }
31 }
```

Konfiguracja Nginx realizuje kilka istotnych optymalizacji:

1. Pośrednikiem optymalizującym komunikację z aplikacją Django jest serwer określony nazwą **gunicorn** (wiersz 1-3). W rzeczywistości są to trzy instancje serwera Gunicorn.
2. Pliki statyczne (zasoby o URI rozpoczynających się segmentem `/static/`) zwracane są bezpośrednio przez Nginx (wiersz 10-13).
3. Zapytania POST są przekazywane bezpośrednio do serwera **gunicorn** (wiersz 15-18).
4. Zasoby pobrane z Memcached nie posiadają informacji rodzaju dokumentu oraz kodowaniu. Dla tego typu odpowiedzi zadeklarowano domyślne wartości (wiersz 22).
5. Klucz oraz instancja Memcached są takie jak te wykorzystywane przez projekt Django (wiersz 20-21).
6. Zapytanie jest przekierowane do **gunicorn** (wiersz 22) jeżeli odpowiedź nie jest obecna w cache (błąd 404) lub gdy usługa Memcached jest wyłączona (błąd 502).
7. Wybrane informacje dotyczące zapytania HTTP przekazywane są do **gunicorn** w postaci dodatkowych nagłówków HTTP (wiersz 25-27).
8. Nginx pełni funkcję proxy buforującego odpowiedzi aplikacji (w scenariuszu pierwszego wywołania danego zasobu, czyli bez udziału Memcached). MMDA zwraca odpowiedź przy użyciu jednej z instancji Gunicorn, który błyskawicznie przekazuje

ją do Nginx, zwalnia zajęte zasoby i oczekuje na nowe zadania. Użytkownik posiadający wolne łącze internetowe obsługiwany jest przez zoptymalizowany serwer Nginx, przez co aplikacja jest w stanie obsłużyć większą ilość równoczesnych użytkowników przy mniejszym zużyciu pamięci.

Za realizację optymalizacji związanych z Memcached po stronie aplikacji Django odpowiada prosta klasa typu *middleware*: `NginxMemcachedMiddleware` (*Listing 6.3*), zapisująca wygenerowaną odpowiedź w cache. Programista nie musi martwić się o szczegóły komunikacji pomiędzy Django a Memcached – wszystkim zajmuje się biblioteka *python-memcached*. Należy jednak uwzględnić mechanizmy aktualizacji oraz retencji zawartych w CouchDB danych (*Listing 6.4*).

Z uwagi na specyfikę aplikacji MMDA, dla wybranych typów zapytań (np. POST), odpowiedzi (np. komunikaty błędów) oraz zasobów (kanały Atom oraz podstrona służąca do podglądu zawartości bazy) bufor Memcached został wyłączony (*Listing 6.3*, wiersz 9-12).

Listing 6.3. Django middleware – klasa współpracująca z *Memcached*

```

1 from django.conf import settings
2 from django.core.cache import cache
3
4 class NginxMemcachedMiddleware:
5     def process_response(self, request, response):
6         if not settings.DEBUG:
7             path = request.get_full_path()
8
9             if request.method != "GET" \
10                or path.endswith('/feed/') \
11                or path.endswith('/refresh/') \
12                or response.status_code != 200:
13                 return response
14
15             key = "%s:%s" % (settings.NGINX_CACHE_PREFIX, path)
16             cache.set(key, response.content)
17
18         return response

```

Listing 6.4. Usuwanie nieaktualnych danych z *Memcached* w Django

```

1 from django.conf import settings
2 from django.core.cache import cache
3 from django.template.defaultfilters import slugify
4
5 def delete_memcached_keys(artist):
6     if not settings.DEBUG:

```

```

7         uri = (slugify(artist.name), artist.get_id)
8
9         keys = ["/artist/%s/%s/" % uri,
10                "/artist/%s/pictures/%s/" % uri,
11                "/artist/%s/videos/%s/" % uri,
12                "/artist/%s/news/%s/" % uri,
13                ]
14
15         ...
16         for key in keys:
17             cache.delete("%s:%s" % (settings.NGINX_CACHE_PREFIX, key))

```

Ostatnim krokiem jest konfiguracja projektu Django do pracy z Memcached jako backendem wbudowanego we framework systemu cache (poniżej istotny fragment pliku `settings.py`):

```

...
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'mmda.engine.cache.NginxMemcachedMiddleware',
)
NGINX_CACHE_PREFIX = 'mmda'
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'

```

6.4.3. Testy

Wybór opisanych narzędzi i specyficznej konfiguracji poparty jest wynikami testów wydajności. Zmierzona została zdolność wdrożonej aplikacji do obsługi dużej liczby równoczesnych żądań pojedynczego zasobu. Jest to jeden z istotnych wskaźników odporności serwisu internetowego na nagły wzrost ruchu¹.

Do testu wykorzystano narzędzie `ab`², uruchomione bezpośrednio na serwerze³:

```

# emerge app-admin/apache-tools
# ab -n1000 -c100 \
    "http://music.local/artist/nine-inch-nails/b7ffd2af-418f-4be2-bdd1-22f8b48613da/"

```

Polecenie wykonuje tysiąc żądań (dotyczących obecnego już w CouchDB) zasobu `/artist/nine-inch-nails/b7ffd2af-418f-4be2-bdd1-22f8b48613da/` utrzymując pulę stu równoczesnych połączeń HTTP.

Wyniki testów (*Tabela 6.2*) jednoznacznie wskazują, iż połączenie Memcached oraz kilku instancji Unicorn daje najlepsze wyniki – zarówno dla treści generowanych po

¹ ang. *Slashdot effect* – nagły wzrost zainteresowania adresem URL (zazwyczaj spowodowany promocją w popularnym serwisie), przekładający się na ekstremalne obciążenie serwera dużą liczbą równoczesnych zapytań dotyczących jednego zasobu.

² Pochodzące z projektu Apache narzędzie do testowania wydajności serwera HTTP.

³ Dzięki temu możliwe było osiągnięcie ruchu generowanego w normalnych warunkach przez dużą ilość rozproszonych klientów (często przekraczającego możliwości łącza).

raz pierwszy, jak i pobieranych z Memcached. Należy również zaznaczyć, iż *test 4* charakteryzuje się brakiem dodatkowego obciążenia procesora, podczas gdy *testy 1-2* to pełne obciążenie CPU przez cały czas trwania testu (związane z wielokrotnym generowaniem odpowiedzi przez framework). Wartości *testu 3* wynikają z faktu, iż w chwili rozpoczęcia testu żądany zasób nie znajdował się w buforze Memcached i dla części żądań musiał być wygenerowany przez framework.

Tabela 6.2. Wyniki testów obsługi dużej liczby równoczesnych żądań HTTP
(Źródło: opracowanie własne)

	obsłużonych żądań [req/s]	średni czas oczekiwania [ms]	całkowity czas testu [s]	średnia prędkość [Kbytes/s]
test 1 ^a	2,55	391,00	391,843	67,97
test 2 ^b	3,33	300,00	300,738	88,56
test 3 ^c	29,17	34,287	34,287	776,79
test 4 ^d	709,11	1,41	1,400	18886,00

^a 1x Gunicorn (*Listing A.1*, strona 105)

^b 3x Gunicorn (*Listing A.2*)

^c 3x Gunicorn + Memcached, odpowiedź pierwotnie nieobecna w cache (*Listing A.3*)

^d (3x Gunicorn) + Memcached, odpowiedź obecna w cache (*Listing A.4*)

Podsumowując – wdrożona aplikacja wykorzystuje dwa niezależne mechanizmy optymalizacji. *Gunicorn* obsługuje żądania wykonywane po raz pierwszy, zwiększając nieznacznie zdolność aplikacji Django do obsłużenia dużej liczby równoczesnych zapytań. *Memcached* jest innym rodzajem optymalizacji: przechowuje wygenerowane odpowiedzi, dzięki czemu kolejne wywołanie danego zasobu jest równoznaczne z pobraniem statycznego pliku z pamięci operacyjnej.

Dzięki wykorzystanym technikom optymalizacji aplikacja charakteryzuje się bardzo dobrą wydajnością. Dodatkowo jest odporna na zmiany obciążenia – nagły wzrost zainteresowania serwisem nie zakłóci pracy serwera. Istotnym atutem jest architektura konfiguracji serwera HTTP oraz samej aplikacji, która umożliwia przeniesienie wybranych usług na oddzielne maszyny, jeżeli w przyszłości pojawi się taka potrzeba.

7. Podsumowanie

Udział aplikacji internetowych (sieciowych) w rynku oprogramowania stale rośnie. Wraz z nim – zapotrzebowanie na programistów oraz architektów oprogramowania tego typu.

Celem niniejszej pracy było zapoznanie czytelnika z wiedzą teoretyczną oraz wybranymi przykładami praktycznymi, dotyczącymi tworzenia aplikacji w języku Python przy użyciu wybranych, otwartych rozwiązań. Autor uważa, iż cel został zrealizowany.

W kontekście pracy słowo *sieciowy* nabiera dodatkowego znaczenia. Reprezentuje serwis wykorzystujący zewnętrzne usługi oraz *sieć powiązań* pomiędzy danymi dostępnymi w Internecie – kierunek, w którym podąża wiele współczesnych aplikacji internetowych. Przestają być zamkniętymi systemami duplikującymi te same wewnętrzne funkcje. Wykorzystują wzajemnie swoje zasoby, umożliwiając szybkie tworzenie złożonych rozwiązań niskim nakładem pracy.

Aplikacja MMDA stanowi funkcjonujący przykład omawianych w pracy zagadnień. Przy użyciu języka Python oraz napisanych w nim narzędzi rozwiązano stosunkowo złożony problem agregacji treści z wielu różnych, niekompatybilnych ze sobą zewnętrznych serwisów.

Python posiada łagodną krzywą uczenia się. Programista rozpoczynający pracę z tym językiem z pewnością doceni szybkość przejścia między pomysłem a implementacją oraz przejrzystość wynikowego kodu. W zastosowaniach komercyjnych atutami może być: składnia niejako wymuszająca powstawanie łatwego w zrozumieniu i zarządzaniu kodu (co ma szczególne znaczenie przy pracy grupowej) oraz wymieniona szybkość implementacji logiki biznesowej (jedna z głównych zalet wykorzystanego frameworka Django).

Opisana w pracy przykładowa architektura wdrożenia opartego o lekki serwer Nginx wraz z buforem Memcached udowadnia, iż wydajność aplikacji zależy nie tylko od optymalnie napisanej logiki biznesowej.

Podsumowując, język Python oraz omówione w pracy otwarte rozwiązania i narzędzia stanowią łatwe, szybkie, spójne i czytelne środowisko do tworzenia zaawansowanych i wydajnych aplikacji sieciowych.

Literatura

- [1] J. Ch. Anderson, J. Lehnardt & N. Slater, *CouchDB: The Definitive Guide*, O'Reilly Media 2010
- [2] Mark Pilgrim, *Dive Into Python*, Apress 2004

Strony internetowe

- [3] David Robinson, Ken Coar *The WWW Common Gateway Interface 1.1*, luty 1996,
<http://datatracker.ietf.org/doc/draft-robinson-www-interface/>
- [4] Wikipedia – Wolna encyklopedia, *Aplikacja webowa*, marzec 2009,
http://pl.wikipedia.org/wiki/Aplikacja_webowa
- [5] lighttpd, *project page*, marzec 2010,
<http://lighttpd.net>
- [6] nginx, *english wiki*, marzec 2010,
<http://wiki.nginx.org>
- [7] *Scaling Wikipedia with LAMP: 7 billion page views per month*, wrzesień 2008,
http://blogs.sun.com/WebScale/entry/scaling_wikipedia_with_lamp_7
- [8] *Overview of Java Web Technologies, Part 1*, marzec 2004,
<http://www.devshed.com/c/a/Java/Overview-of-Java-Web-Technologies-1/>
- [9] Wikipedia, the free encyclopedia, *Java Class Library: Licensing* marzec 2010,
http://en.wikipedia.org/wiki/Java_Class_Library#Licensing
- [10] *Ruby On Rails* marzec 2010,
<http://rubyonrails.org>
- [11] *Język programowania Ruby* marzec 2010,
<http://ruby-lang.org/pl>
- [12] *NoSQL databases* marzec 2010,
<http://nosql-database.org>
- [13] Wikipedia, the free encyclopedia, *Google BigTable* marzec 2010,
<http://en.wikipedia.org/wiki/BigTable>
- [14] Wikipedia, the free encyclopedia, *Cassandra* marzec 2010,
http://en.wikipedia.org/wiki/Cassandra_%28database%29
- [15] Python Programming Language, *Official Website* marzec 2010,
<http://python.org>
- [16] Cython, *C-Extensions for Python* marzec 2010,
<http://cython.org>
- [17] Django, *The Web framework for perfectionists with deadlines* marzec 2010,
<http://djangoproject.com>
- [18] *Nginx, the little Russian web server taking on the giants*, luty 2010,
<http://royal.pingdom.com/2010/02/23/nginx>
- [19] PEP-0008: *Style Guide for Python Code*, lipiec 2001,
<http://www.python.org/dev/peps/pep-0008>
- [20] Couchdbkit, *CouchDB framework in Python* marzec 2010,
<http://couchdbkit.org/>

- [21] RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax* sierpień 1998,
<http://www.ietf.org/rfc/rfc2396.txt>
- [22] Wikipedia, the free encyclopedia, *Web service* marzec 2010,
http://en.wikipedia.org/wiki/Web_service
- [23] W3C, *Extensible Markup Language (XML) 1.0*, listopad 2008,
<http://www.w3.org/TR/2008/REC-xml-20081126/>
- [24] JSON, *Introducing JSON*, marzec 2010,
<http://json.org/>
- [25] MusicBrainz, *License*, marzec 2010,
http://musicbrainz.org/doc/MusicBrainz_License
- [26] MusicBrainz, *XML Web Service*, marzec 2010,
http://musicbrainz.org/doc/XML_Web_Service
- [27] Last.fm, *Web Services*, marzec 2010,
<http://www.last.fm/api>
- [28] Flickr, *Services*, marzec 2010,
<http://www.flickr.com/services/api/>
- [29] *Python Flickr API kit*, marzec 2010,
<http://stuvel.eu/projects/flickrapi>
- [30] The DBpedia Knowledge Base *About*, marzec 2010,
<http://wiki.dbpedia.org/About>
- [31] The reCAPTCHA Project, marzec 2010,
<http://recaptcha.net/>
- [32] Unidecode 0.04.1, marzec 2010,
<http://pypi.python.org/pypi/Unidecode/0.04.1>
- [33] BBC Music, *Developers*, marzec 2010,
<http://www.bbc.co.uk/music/developers>
- [34] Wikipedia, the free encyclopedia, *RSS*, marzec 2010,
<http://en.wikipedia.org/wiki/Rss>
- [35] RFC 4287, *The Atom Syndication Format*, grudzień 2005,
<http://tools.ietf.org/html/rfc4287>
- [36] W3C, *HTML5: W3C Working Draft*, marzec 2010,
<http://www.w3.org/TR/2010/WD-html5-20100304/>
- [37] Creative Commons, *Attribution-ShareAlike 3.0 Unported*, marzec 2010,
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>
- [38] GNU, *Free Documentation License*, marzec 2010,
<http://www.gnu.org/licenses/fdl.html>
- [39] YouTube, *Terms of Service*, marzec 2010,
<http://www.youtube.com/t/terms>
- [40] virtualenv, *Virtual Python Environment builder*, marzec 2010,
<http://pypi.python.org/pypi/virtualenv>
- [41] Unicorn, *WSGI HTTP Server for UNIX*, marzec 2010,
<http://gunicorn.org>

- [42] Memcached, *High-performance, distributed memory object caching system*, marzec 2010, <http://memcached.org/>

A. Załączniki

A.1. Zrzuty ekranowe aplikacji

The screenshot shows the MMDA (Music Metadata Discovery) application interface. At the top, there is a search bar with a dropdown menu set to 'Artist' and a 'Search' button. Below the search bar, the title 'Artist search results' is displayed, followed by the query 'for query "graaf generator"'. The results are presented in a two-column table, where each row lists an artist name, a similarity percentage, and a link to the artist's page. The artists are sorted by similarity, with 'Van der Graaf Generator' at the top (~100%) and 'Standard Generator' at the bottom (~29%).

Artist	Similarity	Link
Van der Graaf Generator	~100%	D-Generator
Graaf	~32%	Love Generator
Generator	~47%	Operator Generator
Grabarna Graaf	~32%	Hydro Generator
Graaf Tel	~32%	Phase Generator
Hannah Graaf	~32%	Bass Generator
Magdalena Graaf	~32%	Generator M
The Generator	~29%	Noize Generator
Mondo Generator	~29%	Void Generator
Wave Generator	~29%	Rave Generator
2nd Generator	~29%	Dream Generator
Random Generator	~29%	Standard Generator

MMDA 1.0 powered by MusicBrainz, Wikipedia, Last.fm, Flickr and Google APIs.

Rysunek A.1. MMDA: strona wyszukiwania artystów

Źródło: <http://music.local/search/artist/graaf-generator/0e06fadaae8bf6bb08f7e80ce7ed6d93221c1e5c/>

Search for

Artist

called

Search

Van der Graaf Generator

ARTIST PROFILE

[profile](#)
[pictures](#)
[videos](#)
[news](#)

Abstract

Van der Graaf Generator, sometimes known by the shorter Van der Graaf, are an English progressive rock band. They were the first act signed to Charisma Records. The band achieved considerable success in Italy during the 1970s. In 2005 they embarked on a reunion, which continues to the present day. The signature Van der Graaf Generator sound in the 1970s was a combination of Peter Hammill's distinctive and dynamic voice and David Jackson's electronically-treated saxophones, generally playing over thick chordal keyboard parts (such as Hammond organ and/or clavinet). The band explored the complete range of phonaesthetics from euphony to cacophony, often within the same song. Van der Graaf Generator albums tended to be darker in atmosphere than many of their prog-rock peers (a trait they shared with King Crimson, whose guitarist Robert Fripp guested on two of their albums), and guitar solos were the exception rather than the rule. Hammill is the primary songwriter for the band, and the line between music written for his solo career and for the band is sometimes blurred. In interviews, Hammill stated that even though he wrote the majority of VdGG music, its arrangements were always collective, while in the case of his solo recordings, he wrote and arranged all the compositions.[citation needed]

read more at Wikipedia

Discography

Album

Compilation

Live

Single

2008	Trisector
2005	Present
1982	Time Vaults
1976	World Record
	Still Life
1975	Godbluff
1971	Pawn Hearts
1970	H to He, Who Am the Only One
	The Least We Can Do Is Wave to Each Other
1969	The Aerosol Grey Machine

Similar artists

Peter Hammill	~100%	King Crimson	~36%	Yezda Urfa	~31%
Gentle Giant	~54%	Änglagård	~34%	Emerson, Lake & Palmer	~31%
Caravan	~38%	Camel	~31%	Gryphon	~29%

Group

also known as: *Van der Graaf*, *VdGG*
active from 1967

Artist related websites

BBC Music Page
 Wikipedia
 Fanpage
 Last.fm
 Discogs
 MusicBrainz

Relationships

Members:

- Guy Evans from 2004
- Peter Hammill from 2004
- Hugh Banton from 2004
- Guy Evans from 1968 to 1972-08-09
- Guy Evans from 1975-01 to 1976-12
- Peter Hammill from 1967 to 1972-08-09
- Peter Hammill from 1975-01 to 1976-12
- David Jackson from 1969-09 to 1972-08-09
- David Jackson from 1975-01 to 1976-12
- David Jackson from 2004 to 2005-11
- Hugh Banton from 1968 to 1972-08-09
- Hugh Banton from 1975-01 to 1976-12
- Keith Ian Ellis from 1968 to 1969-08
- Nic Potter from 1969-09 to 1970-08
- Nick Pearne from 1967 to 1968
- Judge Smith from 1967 to 1968

Folksonomy

experimental progrock british psychedelic classic rock
progressive rock rock art rock progressive 70s

MMDA 1.0 powered by MusicBrainz, Wikipedia, Last.fm, Flickr and Google APIs.

Rysunek A.2. MMDA: profil artysty

Źródło: <http://music.local/artist/van-der-graaf-generator/b892f72d-05e2-4ff7-b863-3d5dec6331fd/>

97



Rysunek A.3. MMDA: zdjęcia artysty

Źródło: maj 2010,

<http://music.local/artist/van-der-graaf-generator/pictures/b892f72d-05e2-4ff7-b863-3d5dec6331fd/>

Search for **Artist** called Search

Van der Graaf Generator

VIDEOS

profile pictures **videos** news refresh

Youtube

There is no official video channel available, but some of YouTube videos displayed here using universal search may be related.

Rysunek A.4. MMDA: zasoby wideo dotyczące artysty

Źródło: maj 2010,

<http://music.local/artist/van-der-graaf-generator/videos/b892f72d-05e2-4ff7-b863-3d5dec6331fd/>

Search for **Artist** called Search

Van der Graaf Generator

NEWS STREAM

profile pictures videos **news** refresh

Feed

Sorry, no news about this artist is available at the moment.. ':-('

Subscribe to Atom feed

MMDA 1.0 powered by MusicBrainz, Wikipedia, Last.fm, Flickr and Google APIs.

Rysunek A.5. MMDA: pusty kanał wiadomości

Źródło: maj 2010,

<http://music.local/artist/van-der-graaf-generator/news/b892f72d-05e2-4ff7-b863-3d5dec6331fd/>

Search for **Artist** called

Search

Indukti

ARTIST REFRESH PAGE

profile

pictures

videos

news

refresh

Refresh settings


Select data to refresh:

☒ artist with releases
☐ pictures
☐ videos
☐ news sources

captured

ing

Type the two words:

 **reCAPTCHA**
stop spam,
read books.

Force refresh

What is artist refresh?

Long story short: removing *old* and fetching *fresh* artist-related data.

MMDA—an engine that runs this website—intensively caches all data. Because of that some of it can be out of date.

Usually data is quite okay (because of automatic refresh thresholds), but if you are sure that some of it requires refresh, you can do it here.

This artist's cache status

artists
bbc - May 28, 2010
mb - May 28, 2010
dbpedia - May 28, 2010
lastfm - May 28, 2010

pictures
lastfm - May 28, 2010

videos
youtube - May 28, 2010

MMDA 1.0 powered by MusicBrainz, Wikipedia, Last.fm, Flickr and Google APIs.

Rysunek A.6. MMDA: formularz odświeżania zasobów artysty

Źródło: maj 2010,

<http://music.local/artist/van-der-graaf-generator/b892f72d-05e2-4ff7-b863-3d5dec6331fd/refresh/>

Indukti

NEWS STREAM

[profile](#)
[pictures](#)
[videos](#)
[news](#)

News stream sources

Articles presented here are fetched from:

[blog.myspace.com](#)
[indukti.com](#)

Feed


[Subscribe to Atom feed](#)

Search for

Artist

called

Search

refresh

NIE ZAGRAMY NA BYTOMSKICH JUWENALIACH

APRIL 27, 2010

Niestety, z powodu zmiany terminu juwenaliów w Bytomiu, nie będziemy mogli zagrać. Strasznie załujemy." – read more at [indukti.com](#)

29-IV-2010 KONCERT Z HUNTEREM

APRIL 14, 2010

Odwołany w sobotę koncert został przeniesiony na czwartek, 29 kwietnia." – read more at [indukti.com](#)

KONCERT W STODOLE W INNYM TERMINIE

APRIL 10, 2010

W związku z katastrofą w Smoleńsku, zaplanowany na dziś koncert w Stodole odbędzie się w innym terminie." – read more at [indukti.com](#)

DEATHSTALKER O IDMEN

APRIL 1, 2010

Sympatyczna recenzja Idmena na stronie Metal Psalter. Odnośniki do innych recenzji znajdziecie na stronie "Recenzje" – read more at [indukti.com](#)

BYTOMSKIE JUWENALIA

MARCH 22, 2010

28 maja zagramy w Parku Miejskim w Bytomiu na tegorocznych Juwenaliach. Szczegóły w dziale 'Koncerty' oraz na stronie Juwenaliów" – read more at [indukti.com](#)

MAQAMAT

MARCH 7, 2010

Jesienią 2009 roku ukazał się debiutancki krążek formacji Maqama. W dwóch utworach gościnnie na skrzypcach zagrała Ewa. O płycie możecie przeczytać w styczniowym Teraz Rocku." – read more at [indukti.com](#)

IDMEN "DAS ALBUM DES JAHRES 2009!"

FEB. 27, 2010

"Idmen" został wybrany albumem roku przez Die deutschsprachige Prog-Community. Wzruszenie nie pozwala nam ścisnąć Was za gardło." – read more at [indukti.com](#)

IDMEN W FINALE PROG AWARDS 2009

FEB. 15, 2010

Rysunek A.7. MMDA: aktywny kanał wiadomości artysty (dwa źródła)

Źródło: maj 2010,

<http://music.local/artist/indukti/news/216de703-21b9-4396-a01f-7927a754c4a7/>

Search for **Artist** called

“progressive”
TAG

Top artists

Dream Theater	CMX
Porcupine Tree	Portugal. The Man
King Crimson	Emerson, Lake & Palmer
Pain of Salvation	Pure Reason Revolution
The Fall of Troy	Spock's Beard
Riverside	Chroma Key
Tool	Van der Graaf Generator
Ayreon	Devin Townsend
Coheed and Cambria	Fair to Midland
Marillion	Cynic
The Mars Volta	Indukti
Opeth	Premiata Forneria Marconi
Genesis	Mike Oldfield
Blackfield	James LaBrie
Rush	Paatos
Yes	Gazpacho
dredg	Rishloo
Camel	Peter Gabriel
Gentle Giant	Pineapple Thief
Pink Floyd	Jethro Tull
Liquid Tension Experiment	Quidam
Transatlantic	Threshold
Oceansize	Änglagård
The Sound of Animals Fighting	Jordan Rudess

MMDA 1.0 powered by MusicBrainz, Wikipedia, Last.fm, Flickr and Google APIs.

Rysunek A.8. MMDA: lista artystów określonych danym tagiem

Źródło: maj 2010,
<http://music.local/tag/progressive/>

Search for

Artist

called

Search

“Rapid Eye Movement”

by **Riverside**

ALBUM RELEASE



Track listing

01. Beyond the Eyelids	7:56
02. Rainbow Box	3:36
03. 02 Panic Room	5:29
04. Schizophrenic Prayer	4:20
05. Parasomnia	8:10
06. Through the Other Side	4:05
07. Embryonic	4:10
08. Cybernetic Pillow	4:45
09. Ultimate Trip	13:13

Release events

2007-09-24 PL	2007-09-28 DE CD	2007-09-28 DE CD
---------------	------------------	------------------

Release related websites

-  Amazon Asin
-  MusicBrainz

Relations with other releases

next in set: [Rapid Eye Movement \(bonus disc\)](#)

Credits

booking: Robert Palmen
 composer: Riverside
 design illustration: Travis Smith
 engineer: Maciej Mularczyk, Magda Szredniczy, Robert Szredniczy
 instrument: Michał Łapaj, Piotr Grudziński, Mariusz Duda, Piotr Kozieradski
 lyricist: Mariusz Duda
 mastering: Grzegorz Pielkowski
 mix: Magda Szredniczy, Robert Szredniczy
 producer: Magda Szredniczy, Riverside, Robert Szredniczy
 recording: Maciej Mularczyk, Magda Szredniczy, Robert Szredniczy
 vocal: Mariusz Duda

Alternative track listings

[Rapid Eye Movement \(12 tracks\)](#)
[Rapid Eye Movement \(bonus disc\) \(6 tracks\)](#)

MMDA 1.0 powered by MusicBrainz, Wikipedia, Last.fm, Flickr and Google APIs.

Rysunek A.9. MMDA: strona publikacji muzycznej

Źródło: maj 2010,

<http://music.local/artist/riverside/release/rapid-eye-movement/159b2371-001e-4879-9cc3-efdac1a1ab4c/>

A.2. Logi wykonanych testów

Listing A.1. Test aplikacji uruchomionej na jednej instancji Gunicorn

```

1 This is ApacheBench, Version 2.3 <$Revision: 655654 $>
2 Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
3 Licensed to The Apache Software Foundation, http://www.apache.org/
4
5 Benchmarking music.local (be patient)
6 Completed 100 requests
7 Completed 200 requests
8 Completed 300 requests
9 Completed 400 requests
10 Completed 500 requests
11 Completed 600 requests
12 Completed 700 requests
13 Completed 800 requests
14 Completed 900 requests
15 Completed 1000 requests
16 Finished 1000 requests
17
18
19 Server Software:      nginx
20 Server Hostname:      music.local
21 Server Port:          80
22
23 Document Path:        /artist/nine-inch-nails/b7ffd2af-418f-4be2-bdd1-22f8b48613da/
24 Document Length:      27120 bytes
25
26 Concurrency Level:    100
27 Time taken for tests:  391.843 seconds
28 Complete requests:    1000
29 Failed requests:      0
30 Write errors:         0
31 Total transferred:    27273000 bytes
32 HTML transferred:    27120000 bytes
33 Requests per second:  2.55 [#/sec] (mean)
34 Time per request:     39184.346 [ms] (mean)
35 Time per request:     391.843 [ms] (mean, across all concurrent requests)
36 Transfer rate:        67.97 [Kbytes/sec] received
37
38 Connection Times (ms)
39           min  mean [+/-sd] median    max
40 Connect:      0    1   3.0      0      12
41 Processing:   479 37269 6903.0  38835  41080
42 Waiting:     478 37268 6902.9  38834  41080
43 Total:       491 37270 6900.2  38836  41081
44
45 Percentage of the requests served within a certain time (ms)
46   50%   38836
47   66%   38984
48   75%   39627
49   80%   39827
50   90%   40768
51   95%   40915
52   98%   41005
53   99%   41071
54  100%   41081 (longest request)

```

Listing A.2. Test aplikacji uruchomionej na trzech instancjach Gunicorn

```

1 This is ApacheBench, Version 2.3 <$Revision: 655654 $>
2 Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
3 Licensed to The Apache Software Foundation, http://www.apache.org/
4
5 Benchmarking music.local (be patient)
6 Completed 100 requests
7 Completed 200 requests
8 Completed 300 requests
9 Completed 400 requests
10 Completed 500 requests
11 Completed 600 requests
12 Completed 700 requests
13 Completed 800 requests
14 Completed 900 requests
15 Completed 1000 requests
16 Finished 1000 requests
17
18
19 Server Software:      nginx
20 Server Hostname:      music.local
21 Server Port:          80
22
23 Document Path:        /artist/nine-inch-nails/b7ffd2af-418f-4be2-bdd1-22f8b48613da/
24 Document Length:      27120 bytes
25
26 Concurrency Level:     100
27 Time taken for tests:   300.738 seconds
28 Complete requests:     1000
29 Failed requests:       0
30 Write errors:          0
31 Total transferred:     27273000 bytes
32 HTML transferred:     27120000 bytes
33 Requests per second:   3.33 [#/sec] (mean)
34 Time per request:      30073.848 [ms] (mean)
35 Time per request:      300.738 [ms] (mean, across all concurrent requests)
36 Transfer rate:         88.56 [Kbytes/sec] received
37
38 Connection Times (ms)
39           min   mean[+/-sd] median   max
40 Connect:        0     1    3.1      0     23
41 Processing:    4041 28629 4341.3  29742  32969
42 Waiting:       4040 28625 4341.0  29739  32968
43 Total:         4053 28630 4338.8  29742  32977
44
45 Percentage of the requests served within a certain time (ms)
46  50%    29742
47  66%    29870
48  75%    29945
49  80%    29991
50  90%    30159
51  95%    30277
52  98%    30452
53  99%    30626
54 100%    32977 (longest request)

```

Listing A.3. Test aplikacji uruchomionej na trzech instancjach Gunicorn oraz Memcached (brak strony w cache)

```

1 This is ApacheBench, Version 2.3 <$Revision: 655654 $>
2 Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
3 Licensed to The Apache Software Foundation, http://www.apache.org/
4
5 Benchmarking music.local (be patient)
6 Completed 100 requests
7 Completed 200 requests
8 Completed 300 requests
9 Completed 400 requests
10 Completed 500 requests
11 Completed 600 requests
12 Completed 700 requests
13 Completed 800 requests
14 Completed 900 requests
15 Completed 1000 requests
16 Finished 1000 requests
17
18
19 Server Software:      nginx
20 Server Hostname:      music.local
21 Server Port:          80
22
23 Document Path:         /artist/nine-inch-nails/b7ffd2af-418f-4be2-bdd1-22f8b48613da/
24 Document Length:       27120 bytes
25
26 Concurrency Level:     100
27 Time taken for tests:   34.287 seconds
28 Complete requests:     1000
29 Failed requests:       0
30 Write errors:          0
31 Total transferred:     27273000 bytes
32 HTML transferred:     27120000 bytes
33 Requests per second:   29.17 [#/sec] (mean)
34 Time per request:      3428.696 [ms] (mean)
35 Time per request:      34.287 [ms] (mean, across all concurrent requests)
36 Transfer rate:         776.79 [Kbytes/sec] received
37
38 Connection Times (ms)
39           min  mean[+/-sd] median   max
40 Connect:    0    1    3.3      0    13
41 Processing:  2 1984 6528.1      7  34265
42 Waiting:    1 1983 6527.8      6  34264
43 Total:      2 1986 6530.9      7  34274
44
45 Percentage of the requests served within a certain time (ms)
46  50%        7
47  66%       13
48  75%       15
49  80%       34
50  90%      3871
51  95%     19821
52  98%     28732
53  99%     31988
54 100%     34274 (longest request)

```

Listing A.4. Test aplikacji uruchomionej na trzech instancjach Gunicorn oraz Memcached (strona obecna w cache)

```

1 This is ApacheBench, Version 2.3 <$Revision: 655654 $>
2 Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
3 Licensed to The Apache Software Foundation, http://www.apache.org/
4
5 Benchmarking music.local (be patient)
6 Completed 100 requests
7 Completed 200 requests
8 Completed 300 requests
9 Completed 400 requests
10 Completed 500 requests
11 Completed 600 requests
12 Completed 700 requests
13 Completed 800 requests
14 Completed 900 requests
15 Completed 1000 requests
16 Finished 1000 requests
17
18
19 Server Software:      nginx
20 Server Hostname:      music.local
21 Server Port:          80
22
23 Document Path:         /artist/nine-inch-nails/b7ffd2af-418f-4be2-bdd1-22f8b48613da/
24 Document Length:       27120 bytes
25
26 Concurrency Level:     100
27 Time taken for tests:   1.410 seconds
28 Complete requests:     1000
29 Failed requests:       0
30 Write errors:          0
31 Total transferred:     27273000 bytes
32 HTML transferred:     27120000 bytes
33 Requests per second:   709.11 [#/sec] (mean)
34 Time per request:      141.022 [ms] (mean)
35 Time per request:      1.410 [ms] (mean, across all concurrent requests)
36 Transfer rate:         18886.23 [Kbytes/sec] received
37
38 Connection Times (ms)
39           min  mean[+/-sd] median   max
40 Connect:     0    1    3.0      0    13
41 Processing:   8   133   24.9    142   160
42 Waiting:     8   132   25.0    141   159
43 Total:       20   134   22.4    142   160
44
45 Percentage of the requests served within a certain time (ms)
46   50%    142
47   66%    144
48   75%    145
49   80%    146
50   90%    149
51   95%    155
52   98%    158
53   99%    159
54  100%    160 (longest request)

```

Spis rysunków

2.1 Cykl życia statycznej strony internetowej	6
2.2 Cykl życia aplikacji internetowej	7
2.3 Historia rozwoju technologii i frameworków sieciowych	9
3.1 Relacje między URI, a URL i URN	39
3.2 Obsługa zapytań HTTP w Django	42
4.1 Rozwój formatów publikacji kanałów wiadomości	64
5.1 Uproszczony proces generowania odpowiedzi dotyczącej zasobu przez MMDA . . .	70
5.2 Schemat agregacji treści wykorzystywanych przez profil artysty	73
5.3 Schemat agregacji treści wykorzystywanych przez stronę publikacji	73
5.4 Schemat agregacji zdjęć w MMDA	74
5.5 Schemat agregacji wideo w MMDA	75
5.6 Schemat tworzenia kanału wiadomości w MMDA	76
5.7 Mechanizm wyszukiwania w MMDA	77
6.1 Unicorn jako serwer WSGI dla aplikacji Django	86
6.2 Memcached jako cache dla odpowiedzi HTML	87
A.1 MMDA: strona wyszukiwania artystów	96
A.2 MMDA: profil artysty	97
A.3 MMDA: zdjęcia artysty	98
A.4 MMDA: zasoby wideo dotyczące artysty	99
A.5 MMDA: pusty kanał wiadomości	99
A.6 MMDA: formularz odświeżania zasobów artysty	100
A.7 MMDA: aktywny kanał wiadomości artysty (dwa źródła)	101
A.8 MMDA: lista artystów określonych danym tagiem	102
A.9 MMDA: strona publikacji muzycznej	103
A.10 MMDA: profil japońskiego artysty	104

Spis listingów

3.1	Uproszczona struktura przykładowego projektu Django	19
3.2	Przykładowy dokument CouchDB w formacie JSON	22
3.3	Odczyt dokumentu z bazy CouchDB	24
3.4	Przykład użycia <i>Couchdbkit</i>	26
3.5	Przykładowy model dokumentu Django/Couchdbkit	28
3.6	Praca z modelem dokumentu Django/Couchdbkit	28
3.7	Konwersja słownika na dokument CouchDB	29
3.8	Funkcja Map definiująca prosty widok CouchDB	30
3.9	Fragment widoku CouchDB w formacie JSON	31
3.10	Dostęp do danych widoku za pomocą <i>Couchdbkit</i>	32
3.11	Pobieranie pełnych dokumentów za pomocą <i>Couchdbkit</i>	33
3.12	Implementacja przykładowego filtra szablonu Django	35
3.13	Przykład użycia wbudowanych tagów <code>url</code> oraz <code>for</code>	36
3.14	Wygenerowany przez szablon przykładowy kod HTML	36
3.15	Tag <code>abstract_for</code> – implementacja	37
3.16	Dziedziczenie szablonów: szablon bazowy <code>base.html</code>	38
3.17	Dziedziczenie szablonów: szablon potomny	38
3.18	Konfiguracja URL w Django: <code>urls.py</code>	41
3.19	Przykładowy formularz POST	43
3.20	Przykładowa funkcja przetwarzająca zapytanie POST	43
4.1	Podstawowa odpowiedź XML MusicBrainz API	49
4.2	Rozszerzona odpowiedź XML MusicBrainz API	49
4.3	Przykładowy wynik wyszukiwania za pomocą MusicBrainz API	50
4.4	Prosty przykład użycia <i>python-musicbrainz2</i>	51
4.5	Dodanie brakującej funkcji <i>python-musicbrainz2</i>	52
4.6	Przykład użycia biblioteki <i>pylast</i>	53
4.7	Przykład rozszerzenia biblioteki <i>pylast</i>	54
4.8	Wyszukiwanie zdjęć za pomocą <i>flickrapi</i>	56
4.9	Pobranie kanału wideo przy użyciu biblioteki <i>gdata</i>	57
4.10	Wyszukiwanie wideo przy użyciu biblioteki <i>gdata</i>	57
4.11	Przetwarzanie obiektu <code>feed</code> biblioteki <i>gdata</i>	58
4.12	Wykonanie zapytania SPARQL przy użyciu biblioteki <i>surf</i>	59
4.13	Kod osadzający test reCAPTCHA	60
4.14	Kod walidujący test reCAPTCHA	61
4.15	Przykładowa odpowiedź XML <i>BBC Music API</i>	61
4.16	Parsowanie XML przy użyciu biblioteki <i>BeautifulSoup</i>	62
4.17	Parsowanie RSS/Atom przy użyciu biblioteki <i>feedparser</i>	65
6.1	Skrypt startowy serwera Gunicorn dla dystrybucji Gentoo	85
6.2	Konfiguracja Nginx do pracy z MMDA	87
6.3	Django middleware – klasa współpracująca z <i>Memcached</i>	89
6.4	Usuwanie nieaktualnych danych z <i>Memcached</i> w Django	89

A.1	Test aplikacji uruchomionej na jednej instancji Gunicorn	105
A.2	Test aplikacji uruchomionej na trzech instancjach Gunicorn	106
A.3	Test aplikacji uruchomionej na trzech instancjach Gunicorn oraz Memcached (brak strony w cache)	107
A.4	Test aplikacji uruchomionej na trzech instancjach Gunicorn oraz Memcached (strona obecna w cache)	108

Spis tabel

2.1 Porównanie wybranych języków programowania	14
6.1 Izolacja procesów wykorzystywanych w aplikacji	84
6.2 Wyniki testów obsługi dużej liczby równoczesnych żądań HTTP	91