



Politechnika Łódzka

Institut Informatyki

PRACA DYPLOMOWA

OPTYMALIZACJA WYDAJNOŚCI APLIKACJI INTERNETOWYCH

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Promotor: dr inż. Aneta Poniszewska - Marańda

Dyplomant: inż. Łukasz Adamczewski

Nr albumu: 170238

Kierunek: Informatyka

Specjalność: Systemy Informatyczne w Zarządzaniu i Handlu Elektronicznym

Łódź 01.09.2012



Institut Informatyki

90-924 Łódź, ul. Wólczańska 215, *budynek B9*

tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Chciałbym w tym miejscu podziękować:

Narzeczonej Oliwii

za wsparcie i wyrozumiałość podczas pisania pracy

Rodzicom

za pomoc w rozwijaniu pasji,

dbając o kondycję zdrowia psychicznego

Pani promotor dr inż. Anecie Poniszewskiej - Marańdzie

za wnikliwą analizę i pomoc podczas pisania pracy.

Spis treści

Wstęp	4
1 Analiza jakościowa i wydajnościowa aplikacji	11
1.1 Analiza wydajności serwera WWW	12
1.2 Analiza wydajności <i>front-endu</i>	20
1.3 Analiza wydajności bazy danych	24
1.3.1 Optymalizacja tabel przy użyciu indeksów	28
1.3.2 Wykorzystanie polecenia EXPLAIN do optymalizacji za- pytań.	31
1.3.3 Optymalizacja definicji tabel	36
1.4 Podsumowanie	39
2 Wymagania i budowa aplikacji	40
2.1 Zakres funkcjonalny	40
2.2 Budowa aplikacji	44
2.3 Projekt bazy danych	45
3 Architektura aplikacji	51
3.1 Rodzaje chmur	52
3.2 Zarządzanie wydajnością aplikacji na platformie <i>GAE</i>	55
3.3 Nierelacyjne bazy danych <i>NoSQL</i>	58
3.3.1 Specyfika <i>Google Datastore</i>	60
3.4 Aplikacja zorientowana na usługi	62
3.4.1 AJAX, a restrykcje bezpieczeństwa kodu <i>JavaScript</i>	63

4	Optymalizacja kodu klienta	67
4.1	Przetwarzanie danych pochodzących z <i>webserwisów</i>	69
4.2	Optymalizacja kodu <i>HTML</i> oraz zasobów	74
5	Optymalizacja aplikacji	79
5.1	Framework Django	79
5.2	Wystawienie usług jako webserwisów	80
5.3	Uruchomienie panelu administracyjnego aplikacji	83
5.4	Framework Symfony	85
5.5	Test wydajności <i>webserwisów</i>	89
	Podsumowanie	94
	Spis rysunków	98
	Spis listingów	100
	Bibliografia	102
	Abstract	104

Wstęp

Obecnie Internet przestał już być tylko wojskowym eksperymentem lub zaledwie miejscem na publikacje własnej strony domowej. Wyewoluował do medium globalnej komunikacji z gigantyczną ilością klientów docelowych. Większość firm, instytucji lub organizacji rządowych czuje się w obowiązku posiadania i utrzymywania strony internetowej, zazwyczaj spełniającej określone cele biznesowe. Pokazuje to jak bardzo powszechnym narzędziem codziennego użytku jest dzisiaj globalna pajęczyna.

Problematyka pracy

Wraz ze wzrostem popularności Internetu jako medium informacyjnego, istotnym problemem stało się obsłużenie napływającego ruchu sieciowego ze strony użytkowników. Pojęcie "czas to pieniądz" ma tutaj kluczowe znaczenie, ponieważ umiejętność przetworzenia jak największej ilości użytkowników w jak najkrótszym czasie będzie przekładała się na realne zyski.

Innym ważnym problemem dotyczącym stron jest zapewnienie wysokiej dostępności usługi, czyli wyeliminowanie do minimum wszelkiego rodzaju przerw wynikających z błędów aplikacji. Temat ten związany jest jednak głównie z odpowiednią konfiguracją sprzętową, czyli wykorzystaniem równolegle działających instancji sprzętowych, które będą wykorzystywane w celu zrównoważenia ruchu lub awaryjnie w wypadku uszkodzenia nośnika danych na jednym z urządzeń.

Należy zaznaczyć na wstępie, że statystycznie tylko **20%** czasu przetwarzania strony przez przeglądarkę, jest poświęcane na oczekiwanie na odpowiedź serwera. Implikuje to olbrzymie znaczenie optymalizacji kodu po stronie klienta w celu znacznego przyspieszenia czasu odpowiedzi strony. Nie bez znaczenia są też czynniki, takie jak lokalizacja geograficzna oraz konfiguracja serwera.

Ze względu na niekwestionowaną popularność języka PHP (według badań z drugiej połowy sierpnia 2012 roku udział wynosi aż **77.8%**) [3] oraz jego olbrzymią prostotę - w stosunkowo krótkim czasie od powstania języka, zaczęły pojawiać się proste strony, następnie aplikacje internetowe, a kończąc na portalach i usługach sieciowych. Język PHP stał się narzędziem na tyle uniwersalnym, że zagadnienia modelowane przy jego użyciu, można z powodzeniem przenieść na inne platformy, takie jak *Java Enterprise* lub *.NET*. W sieci istnieje ponadto wiele gotowych implementacji systemów e-commerce: sklepów (na przykład Magento czy osCommerce), systemów CRM (SugarCRM) lub na przykład platforma edukacyjna Moodle, będąca częścią infrastruktury edukacyjnej Politechniki Łódzkiej. Wymienione przykłady zostały w całości zaimplementowane w języku PHP, a o ich popularności świadczą miliony ściągnięć i wdrożeń.

Jedną z wad gotowych rozwiązań jest fakt, że nie zawsze są one dostosowane do wszystkich stawianych przed projektem wymagań. Powoduje to, że, projekt docelowy w rezultacie otrzyma więcej funkcjonalności niż jest to wymagane. Z drugiej jednak strony możliwe jest, że gotowe rozwiązanie będzie wymagało szeregu usprawnień lub dodania nowych funkcjonalności. W większości wypadków, rozwiązania gotowe nie są jednak od początku dostosowane do bardziej zaawansowanych zastosowań lub wymagań wydajnościowych. Dlatego ważne jest, by wykorzystując istniejące narzędzia wyskalować aplikacje do konkretnych potrzeb lub przewidzieć przyszłe obciążenie.

Ponadto, poza językiem PHP, dzięki stworzeniu platformy *Google Application Engine*, język Python zyskał dużą popularność wśród dostępnych rozwiązań. Jest on uważany za język, w którym można pisać łatwy w utrzymaniu kod w bardzo krótkim czasie, wszystko dzięki wbudowanym mechanizmom i wieloletniemu doświadczeniu twórców [7].

Analiza wydajnościowa przeprowadzana w wypadku prostych aplikacji, które nie będą w przyszłości podlegały intensywnemu obciążeniu nie ma w zasadzie sensu. Publikacja zyskuje na wartości w wypadku, kiedy projekt jest bardziej rozbudowany, a my potrzebujemy szybkiego i rzetelnego sposobu na wykrycie potencjalnych obszarów do optymalizacji. Problem wydajności aplikacji bardzo często ujawnia się w trudnych do przewidzenia momentach, dlatego kluczową umiejętnością jest nauka analizy logów systemowych, udostępnianych przez dostawców.

Cel pracy

W internecie istnieje stosunkowo dużo publikacji związanych z tematyką optymalizacji aplikacji webowych, jednakże w większości wypadków omawiany jest tylko nikły procent wszystkich zagadnień. Zazwyczaj pomijane są aspekty związane z kodem po stronie klienta, a także studium narzędzi i metod badawczych. Najpopularniejsze są publikacje dotyczące optymalizacji samego kodu lub zapytań bazodanowych, w zależności od użytego języka aplikacji lub bazy danych.

Praca ma na celu przedstawienie możliwie najszerszego wachlarza technik optymalizacji. W ramach pracy planowane jest połączenie kilku technologii i wymiana danych między nimi z wykorzystaniem łatwiejszych do przetworzenia obiektów JSON (ang. *JavaScript Object Notation*). Jest to lekki format wymiany danych, natywnie interpretowany przez język *JavaScript*.

Jednym z celów jest również przedstawienie stosunkowo nowych usług

sprzętowo - programistycznych znanych jako *PAAS* (ang. *Platform as a Service*). Takie rozwiązania dają możliwość elastycznego dostosowywania zaplecza sprzętowego do bieżących potrzeb.

Praca ma również na celu prezentację narzędzi, umożliwiających znalezienie wąskich gardeł aplikacji. Tylko sukcesywne łączenie różnych narzędzi oraz ciągłe monitorowanie działania, zapewni oprogramowaniu stabilność oraz wysoką dostępność.

Zakres pracy

Niniejsza praca dyplomowa dotyczy zagadnień inżynierii oprogramowania oraz technologii baz danych wykorzystanych w dziedzinie *e-commerce*. Główny cel badań stanowi przedstawienie realnych korzyści biznesowych wynikających z zastosowania szerokiego spektrum usprawnień aplikacji internetowych.

Treść pracy dyplomowej stanowi "wypadkową" informacji zawartych w dokumentacjach dotyczących użytych technologii, jak również wiedzy autora zdobytej podczas implementacji wielu zróżnicowanych aplikacji internetowych. Bardzo istotne dla publikacji były również informacje pochodzące ze sprawdzonych źródeł, takich jak na przykład oficjalny blog programistyczny Yahoo. Serwis Yahoo ze względu na olbrzymie doświadczenie w kwestii skalowania aplikacji internetowych postanowił podzielić się tą wiedzą na łamach swoich stron internetowych oraz kilku specjalistycznych książek.

Wiele stron takich jak **Twitter** czy **Facebook** na swoich blogach dzieli się doświadczeniami związanymi z implementacją i problemami związanymi z wydajnością. Jest to kolejne cenne źródło informacji o sposobach poprawy działania aplikacji.

W treści pracy przedstawiona jest analiza technik optymalizacji aplikacji internetowych z uwzględnieniem wykonywanych po stronie serwera (*server side*) oraz szeregu usprawnień po stronie przeglądarki *client side*. Równie istotny jest wybór metod, dających najlepsze rezultaty w świetle obecnych

technologii. Dodatkowo, ważne jest wyróżnienie wszystkich pośrednich czynników, które mogą w jakimkolwiek stopniu wpłynąć na działanie aplikacji.

Do implementacji systemu wykorzystano technologie skryptowe PHP oraz Python. Pierwsza z nich pozwoli na szybkie przedstawienie obrazu typowej aplikacji e-commerce (olbrzymi odsetek takich aplikacji w Internecie jest napisanych właśnie w tym języku). Python z kolei pozwoli wykorzystać zalety platformy Google Application Engine (GAE), która zapewnia skalowalną architekturę do późniejszych testów.

Obserwacja zostanie przeprowadzona na przykładzie prostej aplikacji z dziedziny *e-commerce* - księgarni elektronicznej. Wybór takiej tematyki jest celowy, ponieważ najczęściej właśnie w takich aplikacjach występują problemy natury optymalizacyjnej. Spowodowane jest to najczęściej koniecznością obsłużenia wielu klientów, transakcji bazodanowych lub przede wszystkim generowanie rozbudowanego wizualnie interfejsu użytkownika. Podczas generowania obrazów, wykonywania kodu serwera, czy interpretowania rozbudowanych struktur dokumentu *HTML*, czas oczekiwania na wynik może się zauważalnie wydłużyć.

Zaprojektowana w ramach pracy aplikacja stanowi studium przypadku analizy wydajnościowej aplikacji działającej w ściśle określonej architekturze. W ramach analizy omówione zostaną następujące zagadnienia:

- metody badawcze wydajności aplikacji
- dobór odpowiedniej technologii,
- wybór właściwej architektury sprzętowej,
- projekt i optymalizacja bazy danych,
- optymalizacja kodu klienta,
- optymalizacja aplikacji i testy wydajności

Układ pracy

Praca została podzielona na następujące rozdziały:

- **Rozdział pierwszy** opisuje narzędzia badające wydajność aplikacji, a także wyjaśnia teorię działania aplikacji opartych o protokół HTTP
- **W drugim rozdziale** przedstawiono wymagania stawiane przed stworzoną aplikacją oraz jej budowę.
- **Trzeci rozdział** dotyczy optymalizacji związanej z wyborem odpowiedniej architektury aplikacji.
- **Rozdział czwarty** dotyczy optymalizacji związanej z implementacją aplikacji.
- **Rozdział piąty** dotyczy optymalizacji kodu klienckiego.
- **Rozdział szósty** to podsumowanie i wnioski końcowe.

Rozdział 1

Analiza jakościowa i wydajnościowa aplikacji

Projektując aplikacje, już w fazie projektowej, należy myśleć o zapewnieniu wysokiej wydajności, a także o potencjalnych problemach, które mogą się pojawić po wdrożeniu oprogramowania. W celu zapewnienia tworzonej aplikacji najwyższej skuteczności pracy, należy wziąć pod uwagę wiele cech, wśród których najważniejsze zdefiniowane są poniżej.

Skalowalność (ang. *Scalability*)

Cecha aplikacji, określana jako zdolność do wzrostu wydajności aplikacji wraz ze zwiększeniem ilości dostępnych zasobów sprzętowych (serwery WWW, bazy danych, wydajniejsze procesory).

Niezawodność (ang. *High availability*)

Stanowi projekt, jak i odpowiednią implementację systemu, zapewniającą określony poziom ciągłości wykonywania operacji w czasie. Polega to na zapewnieniu jak największej dostępności usługi.

Wydajność (ang. *Performance*)

Przekłada się na możliwość szybkiego wykonywania kodu aplikacji oraz utrzymania czasu odpowiedzi aplikacji na stosownym poziomie.

Często skalowalność jest mylona z wydajnością, jednak przekładając to na bardziej życiowy przykład, wydajność aplikacji można porównać do szybkiego samochodu. Z drugiej strony, bez zapewnienia odpowiednich dróg, ten szybki samochód lub ich grupa, nie jest w stanie rozwinąć maksymalnej prędkości. W najgorszym wypadku może nawet utknąć w korku, blokowany przez inne pojazdy. Skalowalność jest więc zapewnieniem **odpowiedniej infrastruktury**, gwarantującej właściwy rozrost systemu.

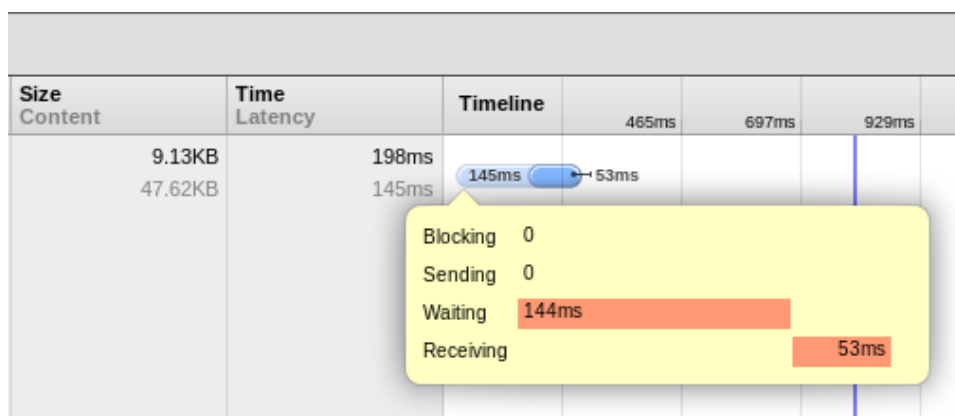
W celu zapewnienia możliwie najlepszej jakości tworzonej aplikacji, należy stale monitorować aktualny poziom wydajności aplikacji. Należy jednak mieć na uwadze, że na wydajność aplikacji składa się czas przetwarzania poszczególnych węzłów systemu. Należy więc testować każde z nich osobnymi metodami, omówionymi w dalszej części pracy.

Szukając przyczyn błędów, warto rozpocząć analizę od najbardziej ogólnego komponentu, czyli od serwera WWW, odpowiedzialnego za wysyłanie odpowiedzi na żądanie użytkownika. Następnie należy dokonać dekompozycji, wyróżniając kolejne węzły systemu, takie jak dalsze instancje serwera WWW lub serwery bazodanowe.

1.1 Analiza wydajności serwera WWW

Zadaniem serwera WWW jest wysłanie do inicjatora żądania wyniku przetwarzania zasobu określonego adresem URL. W najprostszym wypadku, analiza wydajności serwera, polega na odpytaniu go o określony zasób i zmierzenie czasu od rozpoczęcia tej akcji, do odebrania rezultatu. Taki proces można prześledzić i przeanalizować w większości popularnych przeglądarek, na przykład Google Chrome, które jest wyposażone w wiele przydatnych narzędzi analitycznych (Rys. 1.1).

Taki sposób analizy jest jednak przydatny jedynie w wypadku znacznych problemów z wydajnością aplikacji, ponieważ testowanie czasu odpowiedzi dla pojedynczego użytkownika, wykonującego pojedyncze żądanie, nie jest w



Rys. 1.1: Analiza czasu wykonywania strony <http://ftims.edu.p.lodz.pl/> wykonana w przeglądarce Google Chrome

żadnym stopniu miarodajne.

W celu zapewnienia bardziej rzetelnego testu, należy skorzystać z dedykowanych rozwiązań, takich jak **ab** oraz **siege**. Są to typowe narzędzia przeznaczone do sprawdzania, jak dobrze serwer radzi sobie z obsługą bardziej złożonego ruchu sieciowego. Przykładowo, dla wcześniej użytej strony, można zasymulować ruch równy wykonaniu 10 jednoczesnych żądań przez 10 niezależnych użytkowników. W tym celu należy wydać komendę `ab -n 10 -c 10 http://ftims.edu.p.lodz.pl/`. Rezultat działania komendy widoczny jest na listingu 1.1.

Jak można wywnioskować z powyższych danych, narzędzie wykonuje wiele przydatnych analiz, a także wyświetla informacje o badanym zasobie. Widać przede wszystkim, że czas oczekiwania na stronę przy 10 użytkownikach jest prawie trzykrotnie dłuższy, niż podczas jednego żądania wykonanego w przeglądarce.

Oczywiście na wyniki pomiarów ma także wpływ prędkość połączenia internetowego, dlatego w celu pominięcia dodatkowych czynników, testy docelowej aplikacji będą wykonywane przede wszystkim na lokalnym serwerze. Najbardziej miarodajną jednostką, określającą wydajność aplikacji w wypad-

Listing 1.1: Analiza strony z wykorzystaniem narzędzia ab

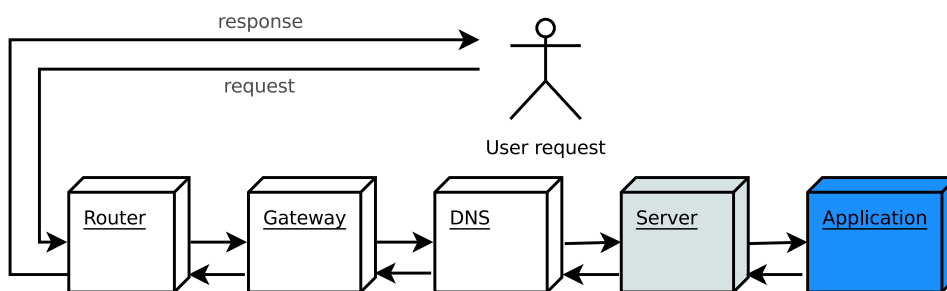
```

1 Server Software:      Apache/2.2.14
2 Server Hostname:      ftims.edu.p.lodz.pl
3 Server Port:          80
4
5 Document Path:        /
6 Document Length:      48759 bytes
7
8 Concurrency Level:    10
9 Time taken for tests:  0.695 seconds
10 Complete requests:    10
11 Failed requests:      0
12 Write errors:         0
13 Total transferred:    492510 bytes
14 HTML transferred:     487590 bytes
15 Requests per second:  14.38 [#/sec] (mean)
16 Time per request:     695.270 [ms] (mean)
17 Time per request:     69.527 [ms] (mean, across all
    concurrent requests)
18 Transfer rate:        691.77 [Kbytes/sec] received
19
20 Connection Times (ms)
21          min    mean[+/-sd]  median    max
22 Connect:      52     63    7.4      63     74
23 Processing:   300    503   90.9     532    632
24 Waiting:      127    233   62.8     235    381
25 Total:        354    565   92.9     593    695
26
27 Percentage of the requests served within a certain time (ms
    )
28  50%         593
29  66%         612
30  75%         625
31  80%         628
32  90%         695
33  95%         695
34  98%         695
35  99%         695
36 100%         695 (longest request)

```

ku narzędzia *ab* jest liczba zapytań na sekundę (*req/s*). Określa ona maksymalną ilość żądań w jednostce czasu, jaką aplikacja jest w stanie obsłużyć. Oczywiście im większa wartość, tym lepsza ogólna wydolność aplikacji.

Na rysunku 1.2 przedstawiono cykl życia żądania od użytkownika inicjującego je, kończąc na odebraniu odpowiedzi serwera. Jak można zauważyć żądanie przebywa stosunkowo długą drogę, nim trafi do faktycznej aplikacji. Wynikiem tego są dodatkowe opóźnienia, zależne od stopnia skomplikowania architektury trzech pierwszych węzłów. Dlatego też należy mieć na uwadze, że problemy z szybkością działania aplikacji nie muszą leżeć wyłącznie po stronie aplikacji lub serwera. Do najbardziej popularnych należą: niska przepustowość łącza internetowego klienta, wolny serwer DNS, daleka lokalizacja geograficzna serwera WWW, źle skonfigurowany router lub bardzo obciążona sieć lokalna.



Rys. 1.2: Cykl życia żądania w typowej aplikacji webowej

Nawiązując do listingu 1.1, wartościami związanymi ze wspomnianymi w poprzednim akapicie węzłami są **Connect** oraz **Waiting**, czyli odpowiednio czas oczekiwania na połączenie z zasobem i czas pobierania odpowiedzi z zasobu. Istnieje pięć głównych czynników, wpływających na czas odpowiedzi serwera.

Położenie geograficzne i problemy z siecią komputerową

Nie bez znaczenia dla czasu odpowiedzi, jaki użytkownik odczuwa jest też lokalizacja serwerów stron. Jeśli serwery są zlokalizowane w USA, a użytkownicy odwiedzający stronę są na przykład z Europy, dystans, jaki musi

pokonać żądanie od momentu dotarcia do zasobu, oczekiwania, aż do jego pobrania jest nie współmiernie większy niż w wypadku stron hostowanych dla tego samego położenia geograficznego. Stopień opóźnienia jest zazwyczaj uzależniony od ilości routerów, serwerów pośrednich, a nawet oceanów, które pokonuje żądanie od punktu początkowego do odbiorcy i z powrotem.

Wielkość dokumentu odpowiedzi serwera

Zależność między wielkością dokumentu, a czasem odpowiedzi serwera jest oczywista, łatwo więc sprawdzić, że im większy dokument trzeba pobrać, tym więcej czasu potrzeba na zakończenie tego procesu. Implikuje to konieczność optymalizacji wielkości strony.

Wykonywanie kodu aplikacji

Najczęstsza przyczyna wolnego działania aplikacji to właśnie brak optymalizacji kodu klienta. Długi czas wykonywania kodu aplikacji implikuje długi czas łączny oczekiwania na odpowiedź serwera. Problem ten zostanie szczegółowiej poruszony w rozdziale 5.

Rodzaj użytej przeglądarki

Nie bez znaczenia dla ogólnego czasu ładowania strony jest również rodzaj użytej przeglądarki. Często wbudowane w przeglądarkę wewnętrzne mechanizmy buforowania zasobów pozwalają w znaczny sposób zredukować ilość zapytań wysyłanych do serwera. Dotyczy to zwłaszcza danych statycznych, takich jak arkusze CSS, pliki JavaScript lub zasoby graficzne, które nie zmieniają się zbyt często.

Konfiguracja serwera WWW

W zależności od użytej technologii, istnieje wiele różnych serwerów HTTP. Wśród najczęściej używanych, prym wiodzie serwer HTTP *Apache*. Dla rozwiązań napisanych w technologii Java często wykorzystywane są serwery *GlassFish*, *Tomcat*, *Jetty*. W większości wypadków zaraz po instalacji, oprogramowanie serwera nie nadaje się jeszcze do wykorzystania w produkcji. Należy wyważyć ustawienia serwera do bieżących potrzeb, ponieważ w więk-

szości wypadków domyślne ustawienia mogą znacznie obniżyć ogólną wydajność. Innym ważnym działaniem jest dostosowywanie serwera do konkretnych zastosowań - do serwowania plików statycznych lepszym rozwiązaniem jest wykorzystanie bardziej oszczędnego pamięciowo i operacyjnie serwera *Nginx*, natomiast do bardziej zaawansowanych zastosowań, w tym wykonywanie kodu aplikacji, serwera Apache lub osobnej instancji serwera *Nginx*.

Administratorzy serwerów WWW mają bezpośredni dostęp do statystyk odwiedzalności stron, przez co pozwala to zaobserwować pewne trendy odwiedzin użytkowników. Często jest tak, że dane zasoby są dużo intensywniej odpytywane przez użytkowników na przykład w czasie 10 minut stronę odwiedza 100 użytkowników. Łatwo to sobie wyobrazić na przykład w wypadku premiery jakiejś nowej gry lub publikacji wyników egzaminu na uczelni. Taki periodyczny, lecz bardzo wzmożony ruch może powodować pewne trudne do ustalenia problemy z działaniem aplikacji. Dlatego też twórcy narzędzia *ab*, zaimplementowali również możliwość testów czasowych (ang. *timed tests*). W ten sposób można zasymulować, jak strona będzie się zachowywała również w takich nagłych wypadkach.

Wydając komendę `ab -c 10 -t 30 http://ftims.edu.p.lodz.pl/` można sprawdzić, jak zachowa się aplikacja odwiedzana przez 10 użytkowników jednocześnie w czasie 30 sekund. Ta komenda pozbawiona jest parametru *-t ilość żądań*, co oznacza, że symulacja zakończy się po 30 sekundach lub po osiągnięciu limitu 50 000 żądań.

Listing 1.2: Test obciążenia czasowego

```
1 Benchmarking ftims.edu.p.lodz.pl (be patient)
2 Finished 504 requests
3
4 Server Software:      Apache/2.2.14
5 Server Hostname:      ftims.edu.p.lodz.pl
6 Server Port:          80
7
```

```

8 Document Path:      /
9 Document Length:    48759 bytes
10
11 Concurrency Level:  10
12 Time taken for tests: 40.180 seconds
13 Complete requests:  504
14 Failed requests:    0
15 Write errors:       0
16 Total transferred:  24822504 bytes
17 HTML transferred:   24574536 bytes
18 Requests per second: 12.54 [#/sec] (mean)
19 Time per request:    797.213 [ms] (mean)
20 Time per request:    79.721 [ms] (mean, across all
    requests)
21 Transfer rate:      603.31 [Kbytes/sec] received
22
23 Connection Times (ms)
24      min    mean[+/-sd] median    max
25 Connect:    48      65  14.3     61    145
26 Processing: 284     436 288.9    376   2957
27 Waiting:    119     199 281.5    151   2660
28 Total:      333     500 287.8    439   3007
29
30 Percentage of the requests served within a certain time (ms
    )
31  50%      439
32  66%      458
33  75%      477
34  80%      493
35  90%      563
36  95%      666
37  98%     1420
38  99%     2142
39 100%     3007 (longest request)

```

Listing 1.2 przedstawia wynik testów czasowych. Najważniejszą informacją z punktu widzenia optymalizacji jest ilość żądań na sekundę, która w tym wypadku wynosi 12.54. Narzędzie `ab` pozwala również zdiagnozować

potencjalne błędy aplikacji pod wpływem zbyt dużego ruchu. Pola takie, jak **Failed requests** oraz **Write errors** ułatwiają określenie prawidłowości wykonywania żądań. W powyższym przykładzie wartości są akceptowalne (średni czas żądania to 0.5 sekundy), co najważniejsze nie występują błędy na poziomie serwera WWW i z dużym prawdopodobieństwem również na poziomie aplikacji. Oczywiście zauważalny jest spadek wydajności w porównaniu z pierwszym testem. Co prawda wartości średnie są zbliżone, jednak widać większe rozbieżności między wartościami minimalnymi a maksymalnymi. Najdłuższe zapytanie zajęło ponad 3 sekundy.

W dokumentacji aplikacji **ab**, można znaleźć informację, że niektóre serwery mogą blokować wysyłane przez nią nagłówki HTTP. W tym celu można wykorzystać przełącznik, umożliwiający podanie się za inną przeglądarkę. Na przykład chcąc zasymulować odwiedziny przy użyciu przeglądarki Chrome, należy wykonać poniższą komendę:

```
ab -n 100 -c 5 -H "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US)AppleWebKit/534.2 (KHTML, like Gecko)Chrome/6.0.447.0 Safari/534.2" http://www.example.com.
```

Pomimo wielu zalet wynikających z korzystania z narzędzia **ab**, istnieje jedna zasadnicza wada. Aplikacja nie daje możliwości przetestowania pewnego scenariusza lub jest to bardzo niewygodne, a przypadku aplikacji z wykorzystaniem technologii *JavaScript* i *AJAX* wręcz niewykonalne.

Dlatego też na przełomie lat wyspecjalizowały się bardziej zaawansowane narzędzia przeznaczone do prześledzenia logicznej kolejności działań wykonywanych na stronie (pewnego przypadku użycia) i wykonywanie analiz właśnie w ramach logicznego zbioru akcji. W ten sposób możliwe jest przeprowadzenie tzw. *testów funkcjonalnych*, a także sprawdzenie, w jakim stopniu są one wrażliwe na zwiększony ruch sieciowy.

Jednym z przykładów takiego narzędzia o naprawdę olbrzymich możliwościach jest *Apache JMeter*. Z jego pomocą możliwe jest nagranie pewnego ciągu akcji wykonanych przy pomocy przeglądarki, a następnie przeprowa-

Tab. 1.1: Tabela z rezultatem działania aplikacji *JMeter*

URL	Średnia	Min	Max	Błąd (%)	Req/Min
..14543704,wiadomosc.html	2299	415	94329	2.2	43.7
...1028235,wiadomosci.html	2485	335	94434	2.8	43.7
...14545325,wiadomosc.html	2023	394	94285	1.8	43.7
Łącznie	2269	335	94434	2.27	131.1

dzenie ciągu analiz i testów na tak wyodrębnionym zbiorze. Tabela 1.1 pokazuje wynik działania przykładowego scenariusza, polegającego na symulowaniu wejścia na stronę <http://www.wp.pl>, a następnie kliknięciu jednego z linków wiadomości, po czym kliknięciu na kolejny link z dostępnych na bieżącej stronie. Ostatnim elementem łańcucha akcji jest wysłanie komentarza do artykułu. *JMeter* podobnie, jak **ab** umożliwia wykonywanie równoległych połączeń użytkowników określanych, jako wątki (*threads*).

W zdefiniowanym przypadku użycia pięciu użytkowników wykonuje jednocześnie tą logikę 500 razy. Na zakończenie dane możliwe są do wyeksportowania do formatu *CSV* lub wyświetlone bezpośrednio na ekranie. *JMeter* jest narzędziem bardzo rozbudowanym, przez co idealnie nadaje się do testowania zaawansowanych scenariuszy zarówno pod kątem poprawności działania, jak również ogólnej wydajności.

1.2 Analiza wydajności *front-endu*

W poprzednim podrozdziale została omówiona tematyka testowania czasu działania zasobów serwowanych przez serwer WWW. Wykorzystując wspomniane wcześniej narzędzia można jednoznacznie ustalić, czy aplikacja funkcjonuje w sposób prawidłowy, czy nie występują błędy w pracy serwera oraz jak dobrze oprogramowanie radzi sobie ze wzmożonym obciążeniem.

Wydawać by się mogło, że problem testowania wydajności aplikacji został

jednoznacznie omówiony. Nic bardziej mylnego. Jedynie w idealnym świecie strona WWW składałaby się wyłącznie z tekstu, a użytkownicy do przeglądania Internetu, korzystaliby wyłącznie z terminali.

Współczesny użytkownik Internetu do przeglądania jego zasobów wykorzystuje przeglądarkę internetową, zdolną do wyświetlania zarówno tekstu, jak i mediów wszelakiego typu. Dlatego też, dla większej precyzji, konieczne jest wyszczególnienie kluczowego komponentu oprogramowania, jakim jest *front-end* aplikacji.

W przypadku aplikacji internetowych, *front-end* to interfejs graficzny służący do komunikacji użytkownika ze stroną i prezentacji danych opracowanych przez zaplecze systemu (*back-end*) w sposób przystępny i zrozumiały.

Odpowiednia optymalizacja *front-endu* jest o tyle ważna, że jest to pierwsza technologia, z jaką użytkownik ma kontakt w momencie korzystania z aplikacji *webowej* [10].

Front-end realizuje przetwarzanie i analizę wyniku odpowiedzi serwera, przy czym obowiązującym językiem komunikacji jest język HTML (*Hiper-Text Markup Language*). Od kilku lat w wyniku rozwoju trendu WEB 2.0, istotnym zabiegiem projektowanych aplikacji, staje się przeniesienie części logiki na stronę przeglądarki (*frontendu*). *Cienkie* do tej pory aplikacje webowe (*thin client*), zaczynają - dzięki zdobyczom technologicznym, takim jak *AJAX*, realizując znacznie szerszy zakres funkcjonalny niż dotychczas. Oznacza to, że przetwarzanie danych może mieć miejsce przy wykorzystaniu przeglądarki internetowej i języka *JavaScript*. Dlatego też, kolejnym istotnym elementem analizy wydajnościowej staje się analiza *front-endu*.

Na podstawie badań firmy *Juniper*, stwierdzono, że średni czas, jaki użytkownik jest w stanie poczekać na "załadowanie" strony to zaledwie 4 sekundy. Nie warto więc tracić potencjalnych klientów strony, tylko i wyłącznie z powodu braku optymalizacji po stronie przeglądarki.

Wśród istniejących na rynku rozwiązań, służących do analizy po stronie klienta, najpopularniejszymi są te, które są wbudowane bezpośrednio w interfejs przeglądarki internetowej (jest to przecież najbardziej intuicyjne podejście). Jednym z pierwszych rozwiązań tego typu była wtyczka **Firebug**, napisana dla przeglądarki Firefox. Jest to obecnie najbardziej zaawansowane narzędzie tego typu, rywalizujące jednocześnie z natywnymi dodatkami deweloperskimi dla przeglądarki Chrome.

Interfejs Firebuga pozwala na szczegółową inspekcję kodu HTML wraz z możliwością dynamicznej operacji na węzłach DOM dokumentu HTML. Nie mniej ważnymi funkcjami są: możliwość wykonywania i debugowania kodu JavaScript na stronie, inspekcja związanego z dokumentem HTML obiektu DOM, edycja i rewizja kodu JavaScript oraz narzędzie do monitorowania ruchu sieciowego wykonywanego przez aplikację.

Ten ostatni moduł pełni podobną rolę, jak narzędzie **ab**, jednak wyświetla wszystkie zasoby, które mają bezpośrednie powiązanie z bieżącym dokumentem HTML. Omawiane wcześniej narzędzia pokazywały jedynie czas renderowania dokumentu HTML, jednak należy mieć na uwadze, że strona internetowa składa się z wielu różnych zasobów, wśród których nie sposób pominąć grafik, arkuszy CSS, kodu JavaScript, apletów Java czy obiektów Adobe Flash.

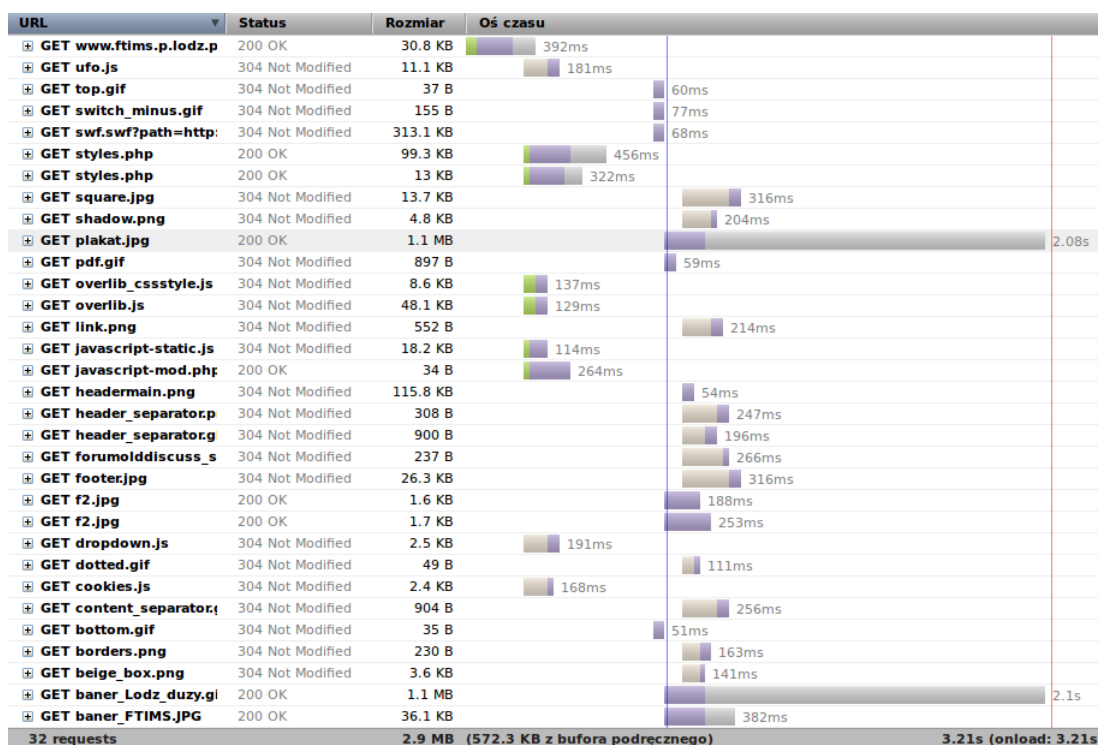
Każda strona może zawierać zróżnicowaną ilość takich zasobów, dlatego na łączny czas ładowania strony składa się zarówno czas oczekiwania na dokument HTML, jak również czas konieczny na pobranie każdego z powiązanych z nim zasobów.

Nawiązując do [14] istnieje zasada, która mówi, że tylko 10-20% czasu odpowiedzi jest spędzane na oczekiwaniu dokumentu HTML, natomiast pozostałe 80-90% to czas na pobieranie pozostałych zasobów i "ładowanie" zawartości DOM.

Przykład analizy ruchu sieciowego na rysunku 1.3 jest najlepszym przykładem tej zasady. Strona wykonuje 32 zapytania do serwera, z czego tylko

jedno to żądanie dokumentu HTML. Pobranie tego dokumentu zajęło około 400ms, tymczasem łączny czas wczytywania strony wyniósł **3.21 sekundy**. Oznacza to, że generowanie dokumentu zajęło zaledwie **12%** łącznego czasu oczekiwania.

Na podstawie danych z Firebuga łatwo stwierdzić pewne nieprawidłowości, bowiem w ramach strony wczytywane są dwa stosunkowo duże (1,1 MB) dokumenty graficzne, które prawdopodobnie nie zostały wymiarowane do odpowiednich rozmiarów. Firebug stanowi więc cenne narzędzie przy diagnostyce *front-endu* strony. Narzędzie staje się jeszcze przydatniejszy przy pojawianiu się elementów dynamicznych JavaScript, ponieważ pozwala śledzić zarówno aktualnie wykonywany kod, jak również nasłuchiwać zapytań asynchronicznych wykonywanych przez AJAX. Narzędzie to może być również przydatne podczas śledzenia zmian dokonywanych w dokumencie HTML, za pomocą narzędzia inspekcji, umożliwia bowiem zbadanie każdego węzła.



Rys. 1.3: Analiza ruchu sieciowego na stronie <http://ftims.p.lodz.pl>

1.3 Analiza wydajności bazy danych

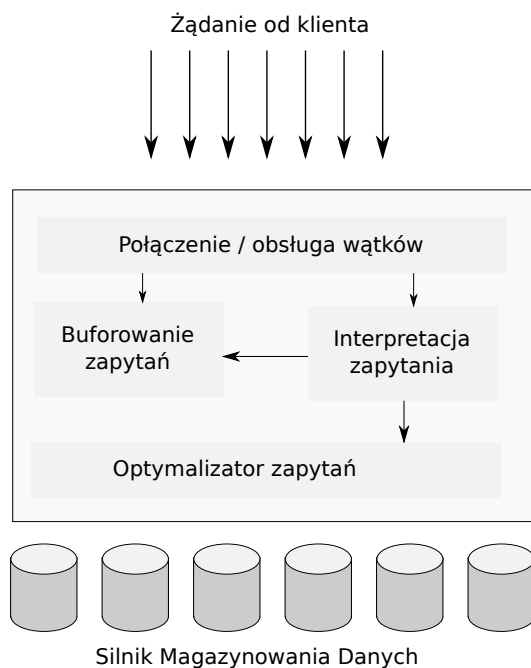
Bardzo często ogólna wydajność aplikacji uzależniona jest od szybkości operacji odczytu / zapisu bazy danych. W pewnym momencie twórcy aplikacji zaczynają oczekiwać od niej większej wydajności. Należy jednak zadać pytanie, co należy tak naprawdę optymalizować? Konkretne zapytanie? Schemat bazy? Czy może sprzęt, na którym baza danych pracuje? Jedyнным sposobem na znalezienie jednoznacznej odpowiedzi, jest zmierzenie pracy wykonywanej przez bazę i sprawdzenie wydajności pod wpływem różnych czynników [13].

Bazy danych ewoluowały na przestrzeni kilkunastu lat, początkowo były to po prostu pliki o określonej strukturze, jednak wraz ze wzrostem wymagań zaczęto stosować równoległy dostęp do danych, a także dbając o spójność danych, zaimplementowano system transakcji.

Obecnie obserwuje się specjalizację baz danych do konkretnych zastosowań. Pomimo dominacji na rynku baz relacyjnych, opartych o standard *SQL/93*, zaczęto również torować drogę nowym rozwiązaniom, takim jak *NoSQL*, czyli bazy danych o dynamicznej strukturze, pozbawionych ściśle zdefiniowanych schematów, przez co zyskują większą elastyczność. Dodatkowo, w określonych zastosowaniach tego typu bazy danych okazują się dużo szybsze niż bazy relacyjne. Szybkość ta jest tym większa, im większa jest przechowywana kolekcja danych. Są to więc bazy wysoce skalowalne, choć mniej spójne niż standardowe.

Wśród systemów bazodanowych, wykorzystywanych w aplikacji *e-commerce*, bardzo dużą popularnością cieszy się oprogramowanie MySQL. Pomimo dużo mniejszych możliwości niż na przykład komercyjne rozwiązania Oracle lub MS SQL Server, omawiana baza danych jest elastyczna i łatwo zaadaptować ją do własnych potrzeb. Od czasu wprowadzenia zgodności ze standardem ACID, MySQL zaczął być szeroko wykorzystywany w *e-biznesie*.

Rysunek 1.4 przedstawia, jak wygląda architektura systemu baz danych MySQL, z punktu widzenia funkcjonalnych komponentów [10]. Pierwsza warstwa zawiera usługi, które wbrew pozorom nie są unikalne tylko dla omawia-



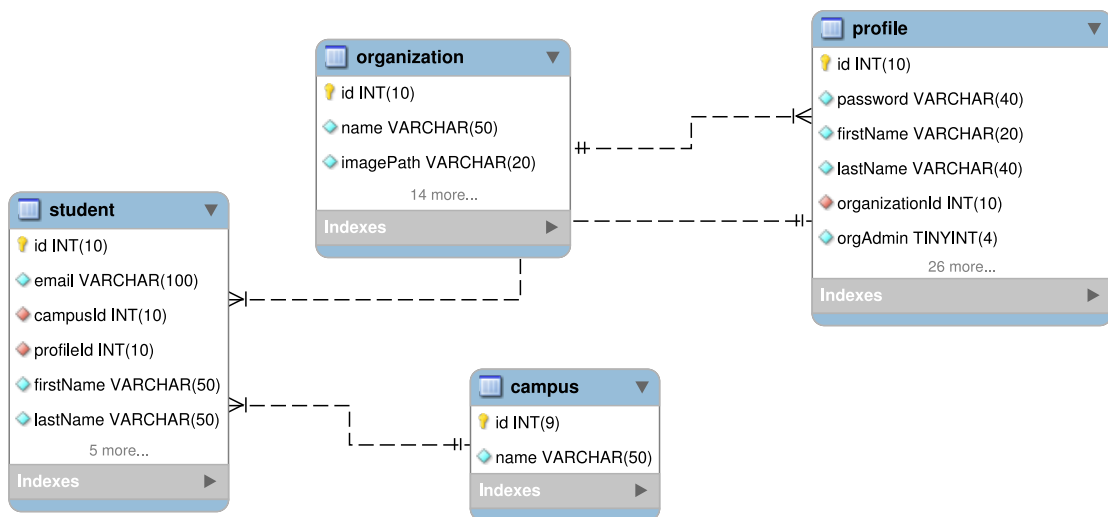
Rys. 1.4: Schemat architektury MySQL

nego oprogramowania. Są to usługi charakterystyczne dla większości narzędzi w architekturze sieciowej. Wyróżniono więc obsługę połączenia, uwierzytelnianie. Druga warstwa wprowadza wiele zmian i komponentów specyficznych dla MySQL. Wyróżniamy więc komponenty odpowiedzialne za parsowanie zapytań, a także za ich optymalizację, buforowanie oraz kod odpowiedzialny za implementację wbudowanych funkcji (data, czas, funkcje matematyczne i kryptograficzne). Każda funkcjonalność oferowana przez którykolwiek z silników składowania danych (*storage engine*), ma tu swoje miejsce (procedury użytkownika, wyzwacze oraz widoki) [13].

Trzecia warstwa wyróżnia wszystkie silniki składowania, które odpowiedzialne są bezpośrednio za przechowywanie i pobieranie danych w MySQL. Każdy z silników ma różne zastosowania (podobnie, jak różne systemy plików w systemach operacyjnych). Komunikacja z każdym z nich odbywa się wykorzystując wewnętrzne API. Wspomniany interfejs ukrywa różnice w specyfice każdego z mechanizmów, przez co zapytania są bardziej abstrakcyjne i prostsze w użyciu dla użytkownika końcowego. Podobnie, jak w wypadku

serwerów WWW, między mechanizmem składowania a serwerem MySQL występuje zależność: *żądanie* - *odpowiedź*. Oznacza to, że serwer wysyła żądanie i oczekuje na odpowiedź.

Najlepszą strategią optymalizacji, jest szukanie miejsc aplikacji, które działają najwolniej. Utworzono przykładowy schemat bazy danych (Rys. 1.5). Na diagramie widać zależność *studenta*, który przynależy do jednego *profilu* (nauczyciela), który z kolei należy do organizacji. Podobnie *student* może należeć do jednego z istniejących *kampusów*.



Rys. 1.5: Schemat testowej bazy danych

Jak zachowa się baza danych przy próbie stworzenia alfabetycznego indeksu studentów, których nazwiska zaczynają się na daną literę (Listing 1.3)?

Listing 1.3: Zapytanie do wyświetlenia menu książki adresowej uczniów

```
1 SELECT left(s.lastName, 1) letter, COUNT(s.id) students
2 FROM student s
3 WHERE s.lastName > 'A'
4 GROUP BY letter
```

Rezultat działania zapytania SQL widoczny jest na listingu 1.4. Jak wi-
dać zapytanie wykonuje się w czasie 40 milisekund. Jest to dosyć krótko, ale

wynika to głównie z rozmiarów kolekcji danych (20244 studentów). Wraz z rozrastaniem się tej tabeli, czas potrzebny na wykonanie tego zapytania będzie się systematycznie powiększał. Dzieje się tak, ponieważ tabela studentów nie posiada żadnego indeksu.

Listing 1.4: Wynik zapytania z listingu 1.3

```
1 +-----+-----+
2 | letter | students |
3 +-----+-----+
4 | A      |      999 |
5 | B      |     1749 |
6 | C      |      814 |
7 | D      |      670 |
8 | E      |      385 |
9 | F      |      918 |
10 | G      |     1599 |
11 | H      |      808 |
12 | I      |      245 |
13 | J      |      194 |
14 | K      |     1695 |
15 | L      |     1156 |
16 | M      |     1411 |
17 | N      |      492 |
18 | O      |      240 |
19 | P      |      686 |
20 | Q      |       14 |
21 | R      |     1259 |
22 | S      |     2662 |
23 | T      |      499 |
24 | U      |       57 |
25 | V      |      244 |
26 | W      |      698 |
27 | Y      |      316 |
28 | Z      |      400 |
29 +-----+-----+
30 25 rows in set (0.04 sec)
```

Czym jest indeks?

Indeks jest strukturą danych przeznaczoną do pomocy systemowi baz danych w efektywnym pobieraniu informacji z tabel. Są one często wymagane dla zapewnienia dobrej wydajności. Indeksy są szczególnie ważne w momencie kiedy baza danych się rozrasta, ponieważ ilość elementów do przeszukiwania wierszowego ulega zwielokrotnieniu.

Podczas zwykłego wyszukiwania wartości w bazie danych, program musi przeszukać każdą kolumnę każdego wiersza w poszukiwaniu określonej wartości. W wypadku indeksów sprawa jest uproszczona, ponieważ dysponujemy pewnym podzbiorem wartości, na przykład z danej kolumny lub wyrażenia. W ten sposób silnik bazy danych wie, że dana wartość znaleziona w określonym indeksie powiązana jest z określonym rekordem, więc odpowiedź jest błyskawiczna.

Oczywiście wykorzystywanie indeksów wiąże się również z dodatkową alokacją danych, ponieważ oprócz danych w tabeli, trzeba dodatkowo przechowywać dane indeksów. Przeliczając jednak zyski do strat, większość przemawia za stosowaniem indeksów.

1.3.1 Optymalizacja tabel przy użyciu indeksów

Optymalizacja zapytań przy użyciu indeksów jest stosunkowo prosta i polega na stworzeniu indeksu, który najlepiej pasuje do wyszukiwanej zawartości. W naszym wypadku potrzeba indeksu przechowującego pierwszą literę nazwiska. Z drugiej strony sortowanie po nazwisku lub nawet wyszukiwanie po nim jest dość częstą operacją, dlatego warto stworzyć kompletny indeks dla pola `lastName` tabeli `student` (Listing 1.5). Na podstawie wyniku uzyskanego w listingu 1.6, widać znaczną poprawę czasu wykonywania. Wszystkie zapytania wykorzystujące kolumnę `lastName`, powinny wykonywać się zdecydowanie szybciej.

Listing 1.5: Utworzenie indeksu na polu `lastName` dla tabeli `student`

```
1 ALTER TABLE 'student' ADD INDEX 'lastName_idx' (('lastName');
```

Listing 1.6: Wynik zapytania z listingu 1.3 po optymalizacji indeksu

```

1 +-----+-----+
2 | letter | students |
3 +-----+-----+
4 | A      |      999 |
5 | B      |     1749 |
6 | C      |      814 |
7 | D      |      670 |
8 | E      |      385 |
9 | F      |      918 |
10 | G      |     1599 |
11 | H      |      808 |
12 | I      |      245 |
13 | J      |      194 |
14 | K      |     1695 |
15 | L      |     1156 |
16 | M      |     1411 |
17 | N      |      492 |
18 | O      |      240 |
19 | P      |      686 |
20 | Q      |       14 |
21 | R      |     1259 |
22 | S      |     2662 |
23 | T      |      499 |
24 | U      |       57 |
25 | V      |      244 |
26 | W      |      698 |
27 | Y      |      316 |
28 | Z      |      400 |
29 +-----+-----+
30 25 rows in set (0.01 sec)

```

Kolejność złączeń

Złączenia (ang. *joins*) są operacją spajającą ze sobą dwie lub więcej tabel. Dopuszczalne są łączenia dwóch różnych tabel lub tej samej tabeli (*self-*

joins). Używanie złączeń wynika w dużej mierze z normalizacji bazy danych, a co za tym idzie usunięcia nadmiarowości z rekordów tabeli. Takie podejście w dużej mierze poprawia spójność danych, ale również może pogorszyć znacznie wydajność zapytań.

Listing 1.7: Kilka możliwych do wykorzystania *złączeń* między tabelą `profile` a `student`

```
1 SELECT SQL_NO_CACHE * FROM profile p inner join student s
   on s.profileId = p.id
2 SELECT SQL_NO_CACHE * FROM profile p left join student s on
   s.profileId = p.id
3 SELECT SQL_NO_CACHE * FROM profile p right join student s
   on s.profileId = p.id
4 SELECT SQL_NO_CACHE * FROM student s inner join profile p
   on s.profileId = p.id
5 SELECT SQL_NO_CACHE * FROM student s left join profile p on
   s.profileId = p.id
6 SELECT SQL_NO_CACHE * FROM student s right join profile p
   on s.profileId = p.id
```

Standard ANSI wyróżnia cztery rodzaje złączeń: INNER, OUTER, LEFT, RIGHT. W specjalnych okolicznościach tabela może być również połączona sama ze sobą. Zasadnicza różnica w specyfice polega na kryterium łączenia - w wypadku *INNER JOINÓW* wymagane jest istnienie odpowiadających sobie krotek po dwóch stronach relacji, natomiast *OUTER JOIN* wymaga spełnienia tego kryterium przynajmniej po jednej stronie relacji - odpowiednio lewej lub prawej.

Różnice między złączeniami przekładają się również na wydajność działania. Najpopularniejsze ze złączeń *inner join'y* są najszybsze, podczas gdy pozostałe przeznaczone są do bardziej specyficznych zastosowań.

Kolejny z przeprowadzanych testów, będzie polegał na złączeniu ze sobą studentów oraz profili, innymi słowy wyświetleniu wszystkich studentów przynależnych do któregoś z profili. Wynik działania poszczególnych testów

Tab. 1.2: Tabela z rezultatem działania poszczególnych zapytań

Typ	Czas wykonywania [sec]	Ilość rekordów
inner join student	0.2557	20244
left join student	5.2125 / 0.5635	20463
right join student	0.2634	20244
inner join profile	0.2726	20244
left join profile	0.3272	20463
right join profile	5.2236 / 0.5704	20463

(Listing 1.7) został zestawiony w tabeli 1.2. Wyniki zapytań stanowią potwierdzenie ogólnie przyjętych zasad optymalizacji zapytań - *inner join* okazał się najszybszym ze złączeń. Jednocześnie widać, że kolejność wykonywania złączeń również ma wpływ na wydajność. Zazwyczaj powinno się zaczynać od tabeli, która posiada mniej rekordów (w tym wypadku tabela *profile*). Ogromna różnica między czasem wykonywania *left join* oraz *right join* w odwrotnej kolejności łączenia wynika z braku indeksu dla pola `profileId` tabeli `student`. Po dodaniu indeksów widać 10-krotną poprawę szybkości wykonywania tego zapytania.

1.3.2 Wykorzystanie polecenia EXPLAIN do optymalizacji zapytań.

Przedstawione dotychczas przypadki były stosunkowo proste do naprawy. Często jednak zapytania są dużo bardziej rozbudowane i trudniejsze do szybkiej dekompozycji. Warto wtedy skorzystać z udostępnianego przez MySQL narzędzia `EXPLAIN`. Oferuje on pomoc w zakresie dekompozycji bardziej skomplikowanych zapytań. Narzędzie to pokazuje między innymi wykorzystane klucze dla łączeń, liczebność łączonych tabel, proponowane usprawnienia indeksów.

Jednym z zapytań, które może stanowić potencjalny problem w analizie, jest zapytanie zaczerpnięte z istniejącej aplikacji, opartej o przedstawiony na rysunku 1.5 diagram *ERD*. Przedstawione na listingu 1.8 zapytanie wyświetla szczegółowe statystyki liczebności, punktacji opartej o zsumowane wartości obecności na danych zajęciach, a także statystyki uczęszczanych programów, praktyk oraz wielu innych specjalistycznych kategorii.

Omawiane zapytanie jest prawdopodobnie jednym z trudniejszych, na jakie można kiedykolwiek trafić podczas analizy bazodanowej. Głównym problemem takich zapytań jest istnienie wielu powiązanych z nim *podzapytań skorelowanych*, czyli takich, których wynik zależy od zapytania głównego. W celu analitycznej analizy tego zapytania użyto narzędzie **EXPLAIN**. Składnia tej dyrektywy jest prosta i polega na dodaniu jej na początku wcześniejszego zapytania. Wynik takiego zapytania zwróci ilość wierszy równą ilości wszystkich podzapytań wykonywanych w trakcie przetwarzania (Tabela 1.3).

Na podstawie tabeli 1.3 widać, że na wynik zapytania składają się rezultaty łącznie 21 mniejszych zapytań. Za każdym razem, kiedy złączamy jedną tabelę z drugą lub wykonujemy zapytanie wewnętrzne albo skorelowane, ilość pracy do wykonania ulega zwiększeniu. Otrzymane zapytanie zawiera informacje podzielone na sześć kolumn. Każda z nich ma inne znaczenie i prezentuje określoną cechę badanego fragmentu zapytania [5].

Listing 1.8: Bardziej rozbudowane zapytanie SQL

```
1 SELECT o.id, o.name ,
2
3 (SELECT group_concat(oc.campusId SEPARATOR ' ' ) FROM
   organizationCampus oc WHERE oc.organizationId = o.id) as
   campusIds ,
4
5 (SELECT count(s5.id)
6 FROM profile p5
7 INNER JOIN student s5 on s5.profileId = p5.id
8 WHERE p5.organizationId = o.id) as students ,
```



```

9
10 IFNULL((SELECT count(DISTINCT s3.id)
11 FROM profile p3
12 INNER JOIN student s3 on p3.id = s3.profileId
13 INNER JOIN studentIntensiveProgram sip on sip.studentId =
    s3.id
14 WHERE p3.organizationId = o.id
15 GROUP BY p3.organizationId
16 ),0) as programs,
17
18 IFNULL((SELECT round(SUM(classes*1 + lon1*3 + shabbaton*5 +
    socialEvents*0.5 + shabbosMeals*2))
19 FROM
20 student s2
21 INNER JOIN profile p2 on p2.id = s2.profileId
22 INNER JOIN 'reportStudentAttendance' AS 'r1' ON r1.
    studentId = s2.id
23 INNER JOIN 'report' AS 'ra'
24 ON ra.id = r1.reportId
25 and ((ra.month >= 9 and ra.year = 2011) or (ra.month <=8
    and ra.year = 2012))
26 WHERE p2.organizationId = o.id
27 GROUP BY p2.organizationId ), 0) as score,
28
29 (SELECT ifnull(sum(datediff("2012-08-30", sy.startDate)
    BETWEEN 30 and 90
30     and sy.startDate between "2011-09-01" and "2012-08-30
    "),0)
31 FROM profile p4
32 INNER JOIN student s4 on s4.profileId = p4.id
33 INNER JOIN studentYeshiva sy on sy.studentId = s4.id
34 WHERE p4.organizationId = o.id) as yeshiva_1_3,
35
36 (SELECT ifnull(sum(datediff("2012-08-30", sy.startDate)
    BETWEEN 91 and 180
37     and sy.startDate between "2011-09-01" and "2012-08-30
    "),0)
38 FROM profile p4

```

```
39 INNER JOIN student s4 on s4.profileId = p4.id
40 INNER JOIN studentYeshiva sy on sy.studentId = s4.id
41 WHERE p4.organizationId = o.id) as yeshiva_4_6,
42
43 (SELECT ifnull(sum(datediff("2012-08-30", sy.startDate) >
181
44     and sy.startDate > 2000),0)
45 FROM profile p4
46 INNER JOIN student s4 on s4.profileId = p4.id
47 INNER JOIN studentYeshiva sy on sy.studentId = s4.id
48 WHERE p4.organizationId = o.id) as yeshiva_6,
49
50 IFNULL((SELECT sum(case
51 when s1.beganSo = "spring/2011" then "2011-01-01"
52 when s1.beganSo = "summer/2011" then "2011-05-01"
53 when s1.beganSo = "fall/2011" then "2011-09-01"
54 else "2012-08-30"
55 end >= "2011-09-01" and right(s1.beganSo, 4) >= 2011 and s1
    .beganSo != "before")
56 FROM student s1
57 INNER JOIN profile p1 on s1.profileId = p1.id
58 WHERE p1.organizationId = o.id
59 GROUP BY o.id),0) AS 'so'
60 FROM organization o
61 GROUP BY o.id
62 HAVING score > 0
63 ORDER BY o.name
```

Opis pól wynikowych polecenia EXPLAIN.

id

Określa numer porządkowy zapytania w kontekście planu wykonywania.

select_type

Określa użyty rodzaj zapytania. Dostępne wartości to:

PRIMARY - zapytanie główne.

UNION (zapytanie typu UNION, drugie lub dalsze zapytanie w kolejności, pole-

gające na dodanie rezultatu jednego zapytania do drugiego, bez wyróżnienia kryteriów łączenia).

DEPENDENT UNION zależy dodatkowo od zapytania głównego.

SUBQUERY - pierwsze użycie **SELECT** w podzapytaniu.

DEPENDENT SUBQUERY - podobnie, jak w poprzednim przypadku, ale zależy od zapytania głównego.

DERIVED - podzapytanie umieszczone w sekcji **FROM**.

UNCACHEABLE SUBQUERY / UNION to dwa najgorsze typy zapytania, ponieważ ich wynik musi być wykonywany dla każdego wiersza głównego zapytania osobno.

table

Odnosi się do nazwy tabeli lub aliasu aktualnie przetwarzanego zapytania.

type

Rodzaj zasobu użytego do złączenia, najpopularniejsze wartości to:

system (złączenie z systemową tabelą, posiadającą jeden wiersz, specjalna odmiana złączenia **const**), najszybszy rodzaj złączenia, ale mało jest praktycznych zastosowań.

Const - złączana tabela ma co najwyżej jeden pasujący element, przez co pobierana jest tylko raz, przez co wiersz jest traktowany jako stała. Dotyczy porównania pewnej stałej wartości z **pełną** wartościami *klucza głównego* lub unikalnego indeksu.

Eq_ref jest dostępny w sytuacjach, kiedy porównujemy wartości z indeksów przy pomocy operatora "=" i istnieje dokładnie jedna wartość do pobrania dla każdego klucza z tabeli łączącej. Przyrównana wartość może być stałą lub wcześniej wczytaną kolumną innej tabeli. Kiedy istnieje więcej niż jedna wartość dla danego klucza tabeli łączącej, to wtedy mamy do czynienia z typem **ref**.

Ref występuje również wtedy, kiedy wykorzystano operatory "<=>". Inną odmianą **ref** jest **ref_or_null**, występujące wówczas, gdy szukano również wierszy które są równe wartości **null**.

ALL - najgorszy możliwy przypadek złączenia. Cała tabela jest przetwarzana i tworzone są wszystkie kombinacje wartości z tabeli łączącej. Trochę lepsza sytuacja jest w wypadku INDEX, kiedy to zapytanie wykorzystuje kolumny, które są częścią pojedynczego indeksu.

possible_keys

Pokazuje proponowane klucze, które można wykorzystać podczas łączenia. W praktyce nie wszystkie z nich można faktycznie użyć.

key

Wykorzystane do łączenia klucze. Jeśli to pole jest puste, podobnie jak poprzednie, to wskazówka, że należy dodać indeks.

Na podstawie dokonanej analizy, widać, że dla podzapytania nr 2 występuje rodzaj złączenia typu *index*. Oznacza to, że można je usprawnić dodając indeks szczegółowy. Ponieważ tabela *organizationCampus*, do której odnosi się alias *oc* jest tabelą łącznikową, to jest tam użyty klucz główny złączony. Dlatego też tylko część tego indeksu jest wykorzystana. Po dodaniu szczegółowego indeksu na pole *organizationId*, osiągnięto pożądany typ *ref*, a czas wykonywania zapytania zredukował się o 40ms (Tabela 1.3).

1.3.3 Optymalizacja definicji tabel

Nie bez znaczenia dla ogólnej wydajności aplikacji jest również wykorzystanie właściwych dla przechowywanych treści typów pól. Istnieje szereg reguł pomocnych w tworzeniu zoptymalizowanych schematów danych. Są to reguły niezależne od wykorzystanego systemu baz danych.

Małe znaczy lepsze

Zazwyczaj należy wykorzystywać możliwie najmniejszy typ danych, który jest w stanie prawidłowo przechować powierzone dane. Mniejsze typy danych są szybsze, ponieważ zużywają mniej miejsca na dysku, w pamięci i w pamięci podręcznej procesora. Dodatkowo, mniejsza ilość cykli procesora jest potrzebna do ich przetworzenia.

Tab. 1.3: Wynik działania polecenia EXPLAIN

id	select_type	table	type	possible_keys	key
1	PRIMARY	o	ALL		
9	DEPENDENT SUBQUERY	p1	ref	PRIMARY,organizationId	organizationId
9	DEPENDENT SUBQUERY	s1	ref	profile_idx	profile_idx
8	DEPENDENT SUBQUERY	p4	ref	PRIMARY,organizationId	organizationId
8	DEPENDENT SUBQUERY	s4	ref	PRIMARY,profile_idx	profile_idx
8	DEPENDENT SUBQUERY	sy	ref	studentIdx	studentIdx
7	DEPENDENT SUBQUERY	p4	ref	PRIMARY,organizationId	organizationId
7	DEPENDENT SUBQUERY	s4	ref	PRIMARY,profile_idx	profile_idx
7	DEPENDENT SUBQUERY	sy	ref	studentIdx	studentIdx
6	DEPENDENT SUBQUERY	p4	ref	PRIMARY,organizationId	organizationId
6	DEPENDENT SUBQUERY	s4	ref	PRIMARY,profile_idx	profile_idx
6	DEPENDENT SUBQUERY	sy	ref	studentIdx	studentIdx
5	DEPENDENT SUBQUERY	p2	ref	PRIMARY,organizationId	organizationId
5	DEPENDENT SUBQUERY	s2	ref	PRIMARY,profile_idx	profile_idx
5	DEPENDENT SUBQUERY	r1	ref	student_idx,report_idx	student_idx
5	DEPENDENT SUBQUERY	ra	eq_ref	PRIMARY,month_idx,year_idx	PRIMARY
4	DEPENDENT SUBQUERY	p3	ref	PRIMARY,organizationId	organizationId
4	DEPENDENT SUBQUERY	s3	ref	PRIMARY,profile_idx	profile_idx
4	DEPENDENT SUBQUERY	sip	ref	student_idx	student_idx
3	DEPENDENT SUBQUERY	p5	ref	PRIMARY,organizationId	organizationId
3	DEPENDENT SUBQUERY	s5	ref	profile_idx	profile_idx
2	DEPENDENT SUBQUERY	oc	index		PRIMARY

Proste typy są wydajniejsze

Podobnie, jak w poprzednim przykładzie, proste typy danych wymagają mniejszych nakładów pracy procesora do przetworzenia. Dla porównania, typ całkowity jest szybszy niż typ znakowy. Dzieje się tak, ponieważ razem ze znakiem powiązane są informacje o kodowaniu oraz tzw. *collations*, czyli regułach sortowania. Inne ważne przykłady dotyczą na przykład przechowywania daty i godziny w dedykowanych typach danych, zamiast typu łańcuchowego. Z drugiej strony, adres IP powinien być przechowywany jako liczba całkowita [8].

Unikanie wartości NULL

Tak często, jak to tylko możliwe, pola tabeli powinny być definiowane jako NOT NULL. Niejednokrotnie schematy baz danych przechowują wartość NULL, nawet jeśli tego nie potrzebują. Należy mieć na uwadze, że optymalizator zapytań dużo gorzej radzi sobie z takimi kolumnami, ponieważ sprawiają one że indeksy, ich statystyki i porównywanie z wartościami jest bardziej skomplikowane. Nawet, jeśli konieczne jest odnotowanie w polu pustej wartości, lepiej użyć jakiegoś arbitralnego symbolu na przykład "0" lub pustego łańcucha znakowego.

Dodatkowo, istnieje kilka przesłanek, dotyczących wykorzystywania pól typu VARCHAR i CHAR. Zazwyczaj lepiej jest wykorzystywać typ VARCHAR (typ znakowy o zmiennej długości), kiedy maksymalna szerokość kolumny jest dużo większa niż wartość średnia, a ilość aktualizacji tego pola jest mała, przez co fragmentacja nie jest problemem. Nie bez znaczenia jest też wybrane kodowanie. W standardzie *UTF-8* każdy znak wykorzystuje różną wartość bajtów przestrzeni dyskowej.

Typ CHAR z kolei najlepiej spisuje się w sytuacjach, kiedy przechowywane mają być bardzo krótkie łańcuchy lub każdy łańcuch jest praktycznie takiej samej długości. Dla przykładu, CHAR jest dobrym wyborem do przechowywania zahaszowanych wartości haseł użytkowników. W odróżnieniu od VARCHAR, ten typ danych nie ulega fragmentacji ze względu na stałą szerokość pola.

Inną cenną radą jest stosowanie pól wyliczeniowych `ENUM`, jeśli zbiór dostępnych wartości znakowych pola jest stały i nie będzie podlegał późniejszym zmianom (wartość zbioru, można jednak modyfikować poleceniem `ALTER` analogicznie jak inne modyfikacje schematu).

1.4 Podsumowanie

W ramach pierwszego rozdziału zostały przedstawione podstawowe metody optymalizacji aplikacji WWW. Omówienie miało na celu ogólne wprowadzenie w tematykę analizy oprogramowania, działającego w sieci. Istnieje wiele technik optymalizacji *front-endu*, baz danych lub serwera WWW. Większość z nich jest dostosowana do konkretnych problemów wydajnościowych lub skonfigurowanej infrastruktury. Omówione zagadnienia są jednak na tyle ogólne, że można je odnieść z powodzeniem do aplikacji w dowolnym języku programowania, a także dowolnym silniku bazodanowym. Znając jednak podstawy, z powodzeniem można wykonać dalsze etapy optymalizacji na własną rękę. Przedstawione porady zapewnią natomiast solidny fundament tworzonej aplikacji.

Kolejny rozdział ma na celu przedstawienie architektury tworzonego oprogramowania. Będzie to rozwinięcie o praktyczne wskazówki powyższego rozdziału dotyczącego optymalizacji serwera WWW.

Rozdział 2

Wymagania i budowa aplikacji

Nie sposób rozpocząć implementacji rozwiązania, bez kompletnego opisu wymagań i zaplanowania budowy aplikacji. Jest to jeszcze ważniejsze, kiedy celem jest stworzenie wydajnej aplikacji.

Aplikacja powinna spełniać wszystkie oczekiwane cele biznesowe, jak również zapewniać prawidłowe funkcjonowanie bez względu na aktualnie panujące obciążenie. Aplikacja demonstracyjna będzie księgarnią internetową, oferującą wiele typowych funkcji, charakterystycznych dla tej branży.

Głównym celem tworzonego oprogramowania jest odpowiednie wyważenie pracy wykonanej po stronie serwera, jak również po stronie przeglądarki. Dlatego też duży nacisk pracy został położony na utworzenie usług, udostępniających interfejs dostępowy do bazy danych. W ten sposób wykorzystując architekturę REST kod JavaScript może dokonywać modyfikacji modelu, przy znikomym udziale serwera WWW.

2.1 Zakres funkcjonalny

Projektowana aplikacja stanowi wirtualną księgarnię. Podobnie, jak w wypadku jej realnego odpowiednika, zapewnia możliwość przeglądania zasobów, podzielonych na kategorie lub oznaczonych określonymi słowami kluczowymi. W odróżnieniu od prawdziwej księgarni, w internetowych aplikacjach koniecz-

ne jest zdefiniowanie pewnej tożsamości, która będzie później podstawą do zakupu książki lub sprawdzenia stanu zamówienia.

Dlatego też istotne jest stworzenie spójnego systemu uwierzytelniania, zapewniającego zarówno bezpieczeństwo, jak również łatwość ewentualnego przypomnienia zapomnianego hasła.

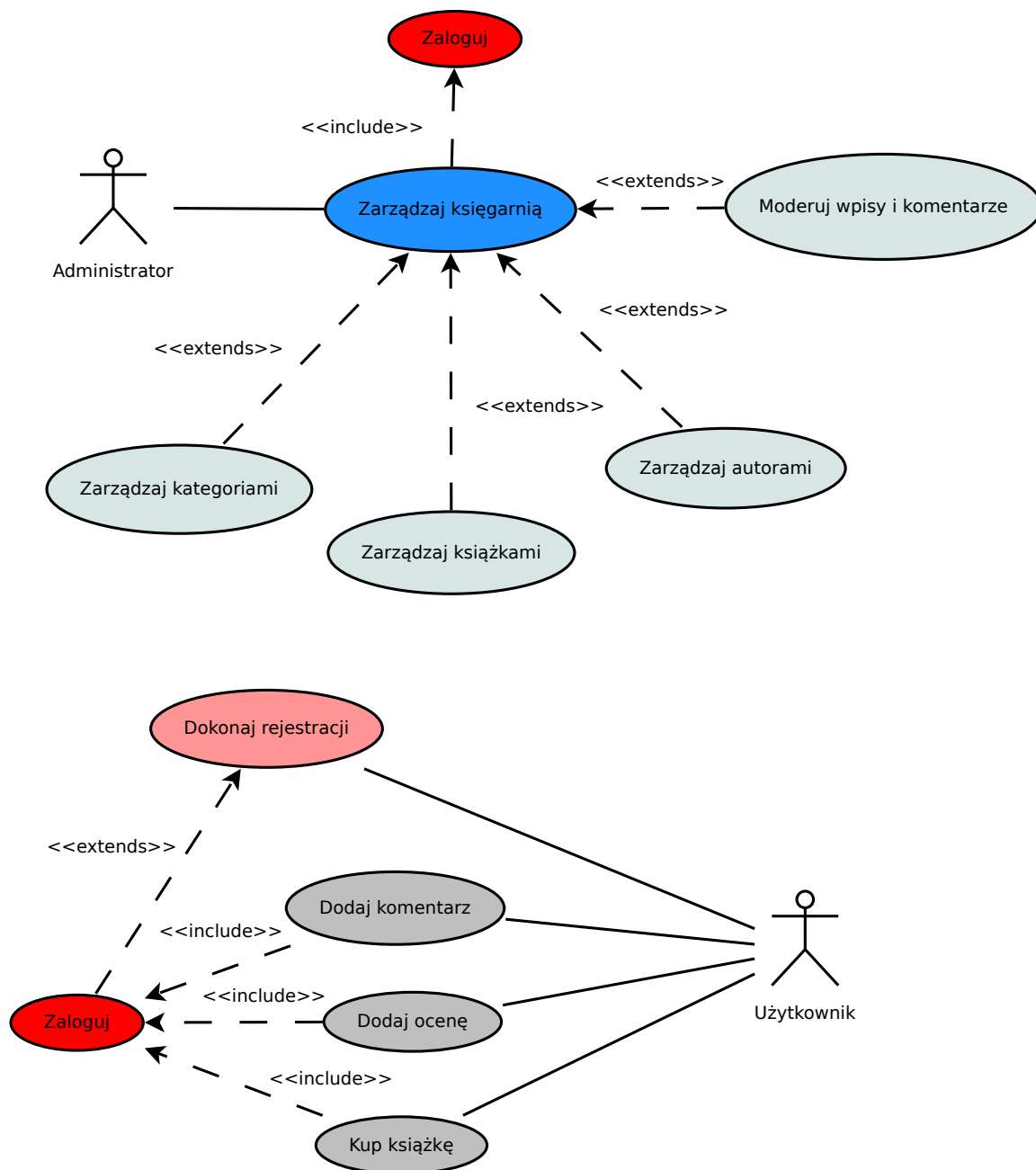
Ponieważ oferta książkowa ciągle się zmienia, na rynek trafiają nowe książki, a nawet tworzą się nowe gatunki, istotną rolę w tworzonej aplikacji stanowi współistnienie zarówno części przeznaczonej dla zwykłego użytkownika, jak również części administracyjnej (*back-end*). Sekcja ta powinna być również zabezpieczona przed potencjalnym włamaniem lub złośliwym atakiem ze strony intruzów.

Ponieważ obecnie w dobie ery WEB 2.0 strony oferują dużo większą interakcję, niż było to możliwe na początku ery Internetu, projektowana aplikacja daje również możliwość oceny i komentowania istniejących zbiorów. Jest to bardzo ważne z punktu widzenia biznesu i sprzedaży, ponieważ popularność i przychylne noty z pewnością napędzą chętnych do kupna.

Wymagania podzielono na część administracyjną i część użytkownika, natomiast zbiór funkcjonalności przedstawiony jest w tabeli 2.1.

Rysunek 2.1 przedstawia przypadki użycia wyszczególnione dla tworzonej aplikacji. Warto nadmienić, że większość akcji użytkownika wymaga wcześniejszego zalogowania, a na początku rejestracji konta w wypadku użytkownika księgarni.

Aplikacja powinna w jak największym stopniu odciążać serwer WWW od niepotrzebnych zapytań. W tym celu akcje usuwania, oceniania i komentowania książek będą realizowane przez kod JavaScript. Dodatkowo, część walidacji danych będzie w pierwszej fazie wykonywana również przy użyciu *front-endu*. Dopiero w wypadku wyłączenia wykonywania skryptów, wysłane zostanie żądanie do sprawdzenia przez serwer. Takie podejście powinno znacznie zredukować obciążenie na przykład podczas rejestracji użytkowników, ponieważ serwer wykonuje tylko minimum swojej pracy (*Lazy Loading*).



Rys. 2.1: Przypadki użycia stworzonej aplikacji

Tab. 2.1: Tabela z wykazem funkcjonalności stworzonej aplikacji księgarni internetowej

Część Administracyjna	Sekcja użytkownika
Dodawanie książek	Rejestracja
Edycja książek	Edycja i podgląd konta
Usuwanie książek	Ocenianie książek
Moderacja wpisów	Komentowanie książek
Dodawanie kategorii	Przeglądanie kategorii
Dodawanie plików do książek	Pobieranie streszczeń książek
Nadzorowanie kont użytkowników	Wyszukiwanie książek w wyszukiwarce

Należy jednak mieć na uwadze bezpieczeństwo oprogramowania, dlatego nie należy polegać w 100% na walidacji wykonywanej przez JavaScript, ponieważ użytkownik może bez problemu wyłączyć działanie skryptów. Dlatego też podczas tworzenia tego typu rozwiązań, walidacja po stronie klienta powinna iść zawsze w parze z walidacją po stronie serwera.

Ze względu na konserwację i rozwój tworzonej aplikacji, jest ona zarządzana przy pomocy systemu kontroli wersji *git* w ramach serwisu <http://github.com>. W ten sposób tworzony kod jest pod ścisłą kontrolą, może być potem rozwijany przez więcej niż jednego programisty, eliminując konflikty podczas pracy na wspólnych zasobach.

Aplikacja wykorzystuje język HTML5, który staje się powoli standardem tworzenia kodu po stronie klienta. Zawiera bowiem wiele nowoczesnych mechanizmów i udogodnień w stosunku do poprzedniej wersji na przykład *HTML4* oraz *XHTML 1.0*. Mając na uwadze prawidłowe funkcjonowanie w poszczególnych przeglądarkach i różną implementację nowego standardu, aplikacja wykorzystuje dodatek *HTML5 Boilerplate* oraz *Twitter Bootstrap* jako metodę na otrzymanie podobnych rezultatów w szerokiej gamie urzą-

dzeń (w tym urządzenia mobilne). Jednocześnie komponenty wchodzące w skład **Twitter Bootstrap** umożliwiają łatwe tworzenie komponentów interfejsu użytkownika z już istniejących elementów.

Dostęp do danych w aplikacji możliwy jest dzięki *webserwisom* udostępniających pewien interfejs dostępowy. W ten sposób szczegóły implementacyjne są ukryte, a użytkownik wykorzystuje tylko te funkcjonalności, które są mu faktycznie potrzebne. W celu uproszczenia implementacji *webserwisów* wykorzystano rozszerzenie *Piston* dla frameworka *Django*.

Projektowana aplikacja wykorzystuje technologię AJAX (ang. *Asynchronous JavaScript and XML - asynchroniczny JavaScript i XML*), dzięki czemu możliwa jest symulacja zachowania aplikacji desktopowej. Reakcja na akcje użytkownika następuje bez potrzeby przeładowania strony. Pozwala to na dużą oszczędność przepustowości i czasu koniecznego do przetworzenia całej strony. Dzięki zastosowaniu frameworka *jQuery* dla języka JavaScript, możliwe jest stosowanie bardziej przyjaznego interfejsu użytkownika, przy jednoczesnym zapewnieniu kompatybilności wstecz z istniejącymi wersjami przeglądarek internetowych. Dzięki zaimplementowanym *webserwisom*, zwracającym dane w formacie *JSON*, przetwarzanie i wyświetlanie danych może być wykonane właśnie po stronie klienta z użyciem *JavaScript* i wbudowanych mechanizmów przetwarzania żądań.

2.2 Budowa aplikacji

Zaprojektowana aplikacja składa się tak naprawdę z dwóch niezależnych podaplikacji, jedna z nich hostowana jest wykorzystując usługę *Google Application Engine*, która zostanie omówiona w rozdziale 3. Aplikacja ta będzie udostępniała usługi oparte o architekturę *REST* (ang. *Representation State Transfer*). Będzie zawierała również panel administracyjny do zarządzania księgarnią. Jest ona napisana w języku *Python*, wykorzystując framework *Django*, a także wtyczkę *Piston*, umożliwiającą proste wystawienie webser-

wisów na podstawie modelu bazy danych.

Druga aplikacja będzie zawierała zakres funkcjonalny zwykłego użytkownika, natomiast wszystkie dane będzie pobierała przy użyciu udostępnionych przez wcześniej omawianą aplikację *webserwisów*. Aplikacja ta będzie napisana w języku PHP z wykorzystaniem frameworka *symfony*.

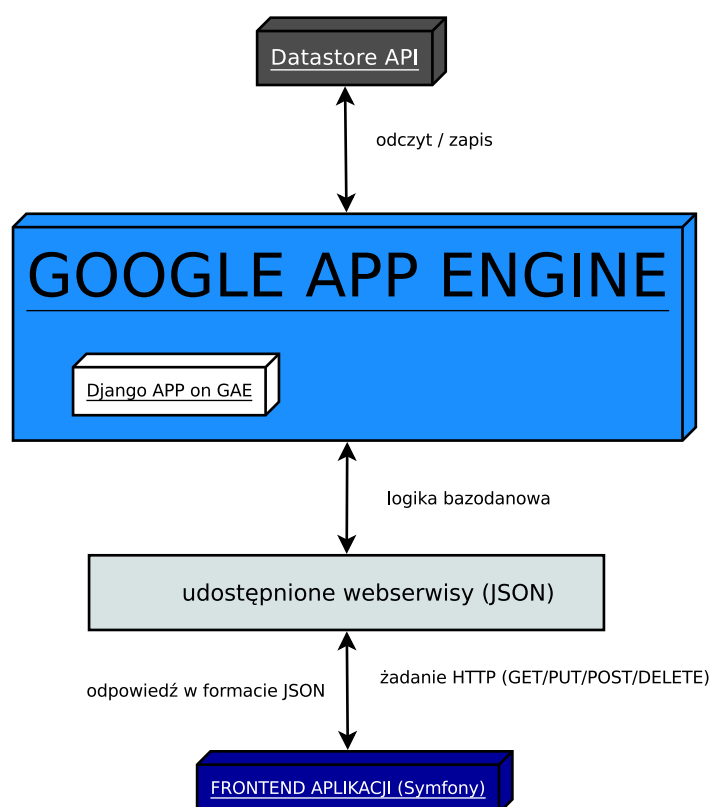
Obydwie aplikacje wykorzystują wzorzec MVC do separacji warstw logiki, prezentacji i danych. Dodatkowo, wykorzystane frameworki są stworzone do szybkiego tworzenia oprogramowania, dzięki założeniom *scaffoldingu* oraz konwencjom *DRY* (ang. *Don't repeat Yourself*) i *KISS* (ang. *Keep it simple stupid*).

Schemat budowy aplikacji przedstawia rysunek 2.2. Każdy element w systemie posiada szereg zależności, których nieprawidłowe działanie pociąga wiele daleko idących konsekwencji. Dlatego istotne jest, by kod aplikacji był spójny i posiadał modularną budowę. Umożliwi to łatwiejsze wykrywanie problematycznych aspektów aplikacji, istotnych dla prawidłowego działania całego projektu.

2.3 Projekt bazy danych

Rysunek 2.4 przedstawia projekt bazy danych wykorzystanej w ramach projektu. Diagram ERD przedstawia jednak bazę w podejściu relacyjnym, w późniejszej części pracy pokazana zostanie baza NoSQL oraz proces de-normalizacji, który musiał zostać dokonany. Na szczęście dzięki opisanemu w następnym rozdziale frameworkowi *Django-norel*, możliwe jest łatwe przechodzenie między istniejącymi implementacjami baz danych.

Baza zawiera książki, powiązane z nimi słowa kluczowe i kategorie. Każda książka ma swojego wydawcę. Dodatkowo, użytkownicy mogą komentować wybrane książki, łącznie z wystawieniem oceny. Istotnym zamierzeniem projektowanego schematu była jak największa optymalizacja użytych pól. Na przykład ocena w komentarzu ma wartości między 1 a 5, więc wykorzystano



Rys. 2.2: Diagram budowy aplikacji księgarni internetowej

typ danych `UNSIGNED INT(1) NOT NULL`, który przechowuje wartości od 0 do 255, bez składowania wartości pustej. Do przechowywania hasła wykorzystano typ `CHAR(32)`, ponieważ w bazie przechowywana jest wartość skrótu hasła przy użyciu algorytmu *SHA1*.

Ze względu na pewne ograniczenia bazy *Google Datastore*, konieczne było stworzenie pewnej nadmiarowości w schemacie `book` oraz `category`. Ponieważ w *GAE* nie funkcjonują *złączenia*, tabela `book` przechowuje listę identyfikatorów kategorii (`categories`) w celu imitacji zachowania dla relacji *wiele do wielu*. Należy dodać, że *GAE*, posiada tym danych zdolny do przechowywania list wskaźników do poszczególnych encji, jak również liczb całkowitych i łańcuchów znakowych. Dodatkowo posiada metody do sprawdzania przynależności zadanych wartości do zbiorów opisanych przez te listy. Kolejnym elementem specyficznym dla tej platform są pola zliczające - imitujące działanie funkcji grupujących. Pole `count` zawiera, więc informację o ilości książek w danej kategorii. Podobne zastosowanie ma pole `average` tabeli `book`. Zawiera ono średnią głosów wystawionych w komentarzach.

W celu osiągnięcia zachowania funkcji grupujących konieczne jest *przeciążenie* metod zapisu dla obiektów reprezentujących bazodanowe encje (Listing 2.1).

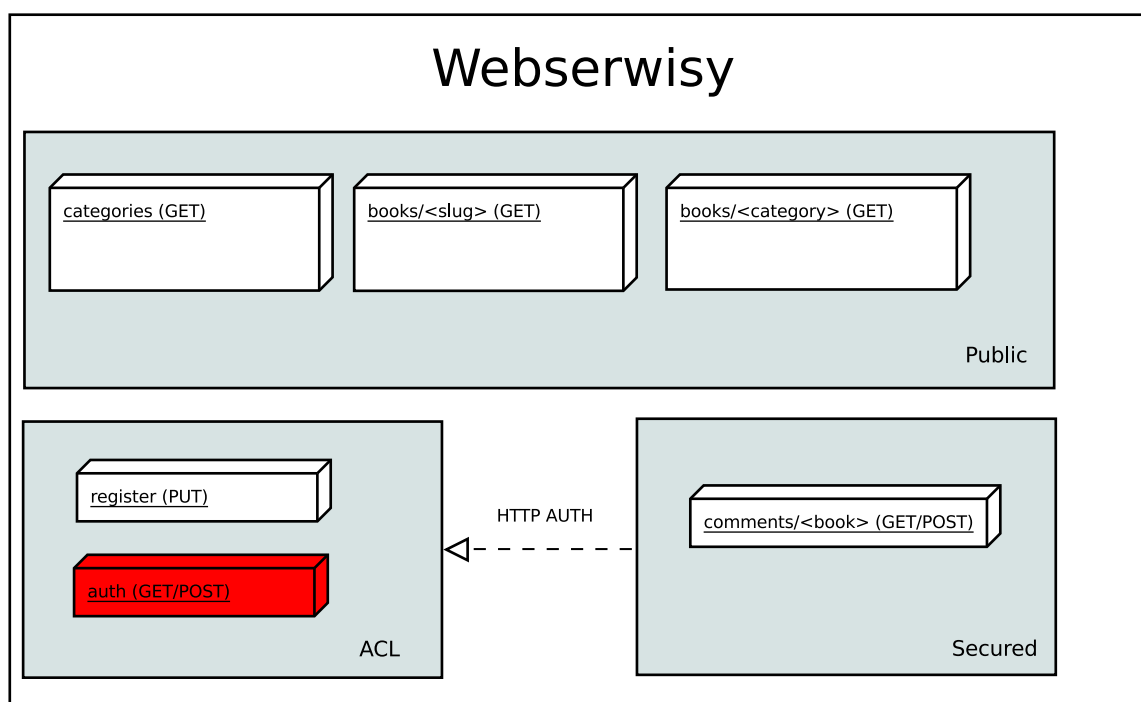
Listing 2.1: Przeciążenie metody zapisu dla klas reprezentujących encje `book` oraz `comment`

```
1 class Comment(models.Model):
2
3     def save(self, force_insert=False, force_update=False,
4               using=None):
5
6         comments = Comment.objects.filter(book = self.book)
7         average = 0.00
8
9         for comment in comments:
10             average += comment.grade
```

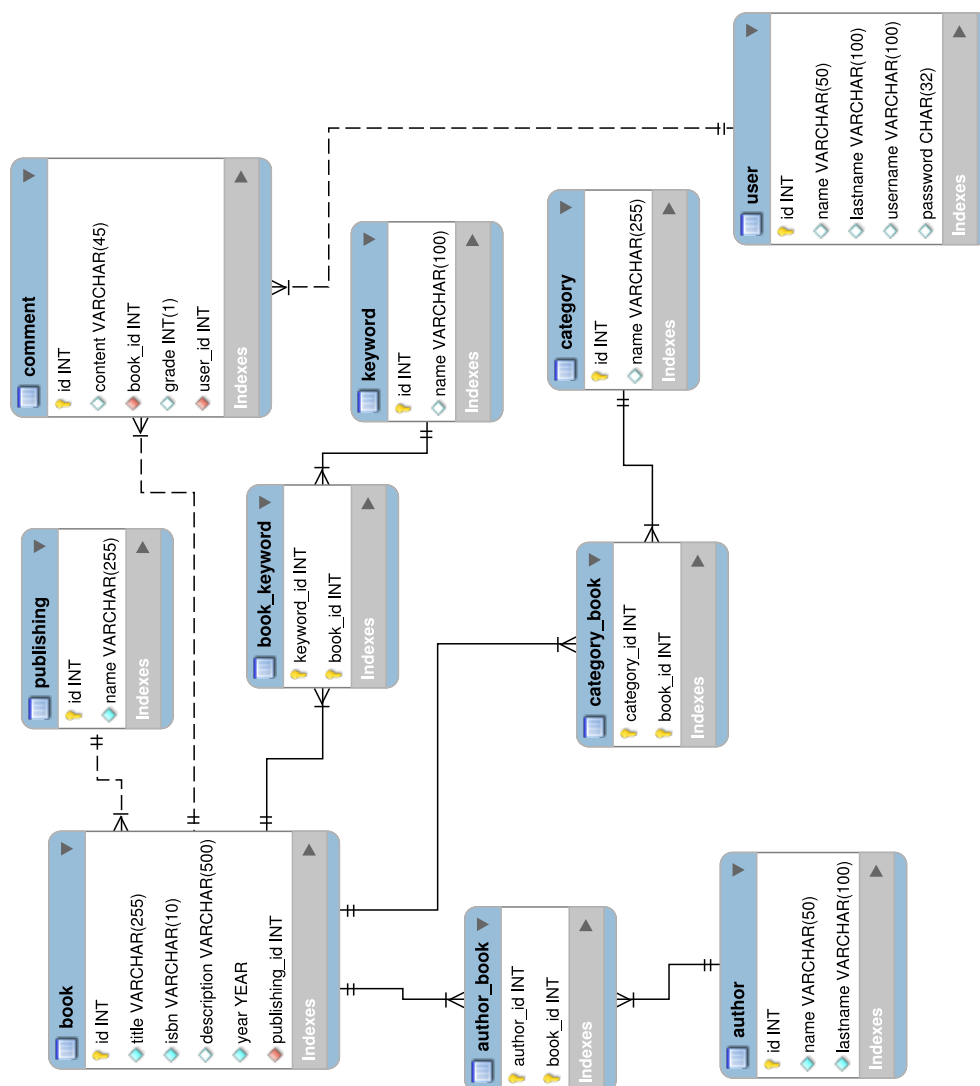
```
10
11     self.book.average = self.grade if not average else
        (average + float(self.grade)) / (comments.count
        () + 1)
12     self.book.save()
13
14     super(Comment, self).save()
15
16
17 class Book(models.Model):
18     def save(self):
19
20         if self.id is None:
21             for category in self.categories:
22                 c = Category.objects.get(id = category)
23                 c.count = 1 if not c.count else c.count + 1
24                 c.save()
25
26         super(Book, self).save()
```

Podczas projektowania należy mieć na uwadze specyfikę użycia poszczególnych tabel. Dla tabel **kategorii** i **książek** dominować będą operacje odczytu, natomiast dla tabeli komentarzy operacje zapisu.

Ponieważ z założenia dostęp do danych będzie realizowany poprzez udostępnione usługi, rysunek 2.3 przedstawia ich definicje, a także metody komunikacji. W ten sposób tylko za pomocą odpowiednich metod HTTP (PUT - tworzenie, POST - tworzenie / aktualizacja, GET - pobieranie), możliwe jest wykonywanie odpowiedniej logiki.



Rys. 2.3: Usługi udostępnione na platformie *GAE* (*Google Application Engine*)



Rys. 2.4: Projekt bazy danych aplikacji księgarni internetowej

Rozdział 3

Architektura aplikacji

Istotnym celem pracy jest zapewnienie możliwie najlepszej w danej chwili wydajności. Twierdzenie to powinno być prawdziwe również wówczas, kiedy aplikacja znajduje się pod silnym ruchem sieciowym. Aby to zapewnić, konieczne jest wykorzystanie odpowiedniej architektury aplikacji.

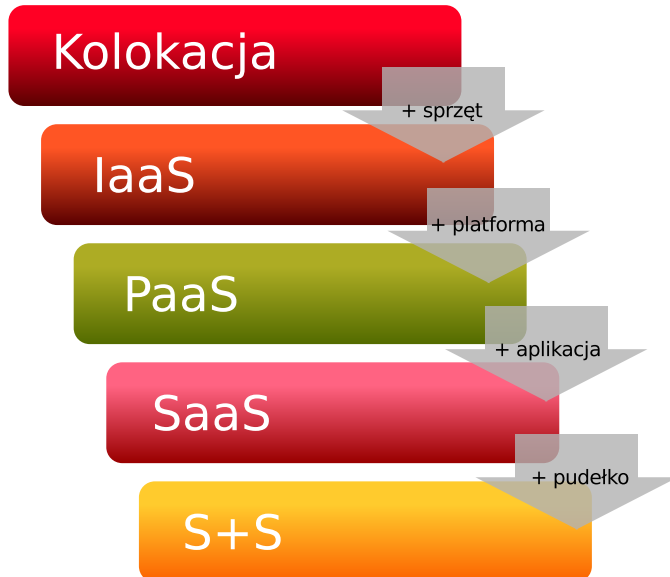
Od kilku lat na rynku IT można zaobserwować duże zainteresowanie związane z chmurami obliczeniowymi (ang. *Cloud Computing*). Jeszcze większe zamieszanie na rynku spowodowało udostępnienie przez Google oraz Microsoft ich flagowych produktów znanych jako *Google Application Engine* (GAE) oraz *Windows Azure*. Są to usługi kwalifikowane jako *PaaS*, czyli *Platform as a Service*. Oznacza to, że korzystając z ich produktów otrzymujemy kompletne środowisko uruchomieniowe aplikacji oraz zaplecze technologiczne umożliwiające uruchamianie aplikacji. W wypadku GAE możliwe jest programowanie aplikacji z wykorzystaniem udostępnionego przez usługę zbioru bibliotek, napisanych w trzech językach: Java, Python oraz Go. Następnie przy pomocy udostępnionych narzędzi możliwe jest wdrożenie aplikacji (*Deployment*).

Takie podejście do tworzenia aplikacji internetowych zyskało olbrzymią ilość zwolenników, ponieważ pozwoliło na całkowite przeniesienie ciężaru zarządzania skomplikowaną infrastrukturą na producentów rozwiązań *Paas*.

Innym ważnym powodem, dla którego wielu ludzi zdecydowało się na wykorzystanie nowych usług, jest możliwość konfiguracji architektury do własnych potrzeb, a także (co było kluczowym powodem), do aktualnego obciążenia aplikacji.

Preferowanie *cloud computingu* jest napędzane przez wiele czynników, wśród których należy wymienić łatwość dostępu - wszystko czego potrzeba to przeglądarka. Z drugiej strony równie ważna jest łatwość zarządzania - nie potrzeba etatu do doświadczonego administratora. Możliwości chmur obliczeniowych są doceniane także przez dostawców usług ze względu na łatwe zarządzanie infrastrukturą, ponieważ centra danych posiadają jednorodny sprzęt i oprogramowanie, co więcej są one pod kontrolą jednego kompetentnego podmiotu [6].

3.1 Rodzaje chmur



Rys. 3.1: Zestawienie rodzajów chmury ze względu na udostępniane zasoby

Najogólniej chmury dzielą się na *prywatne* i *publiczne*. Prywatne wchodzą w skład części organizacji, ale jednocześnie stanowią autonomiczną usługę.

Publiczne natomiast udostępniane są przez zewnętrznych dostawców usług [11].

Rysunek 3.1 pokazuje zasadnicze różnice między poszczególnymi rodzajami chmur. Chmura oferowana przez *Google Application Engine*, zapewnia wsparcie w zakresie architektury serwera, systemu plików, a także systemu bazodanowego. Dodatkowo, *GAE* udostępnia również możliwość korzystania z usług w tle oraz serwerów mailowych, a także wiele mniejszych usług takich jak wykonywanie żądań do zewnętrznych usług. Nowością w fazie testów jest także możliwość, rozdzielenie obciążenia na kilka wersji aplikacji (*traffic splitting*).

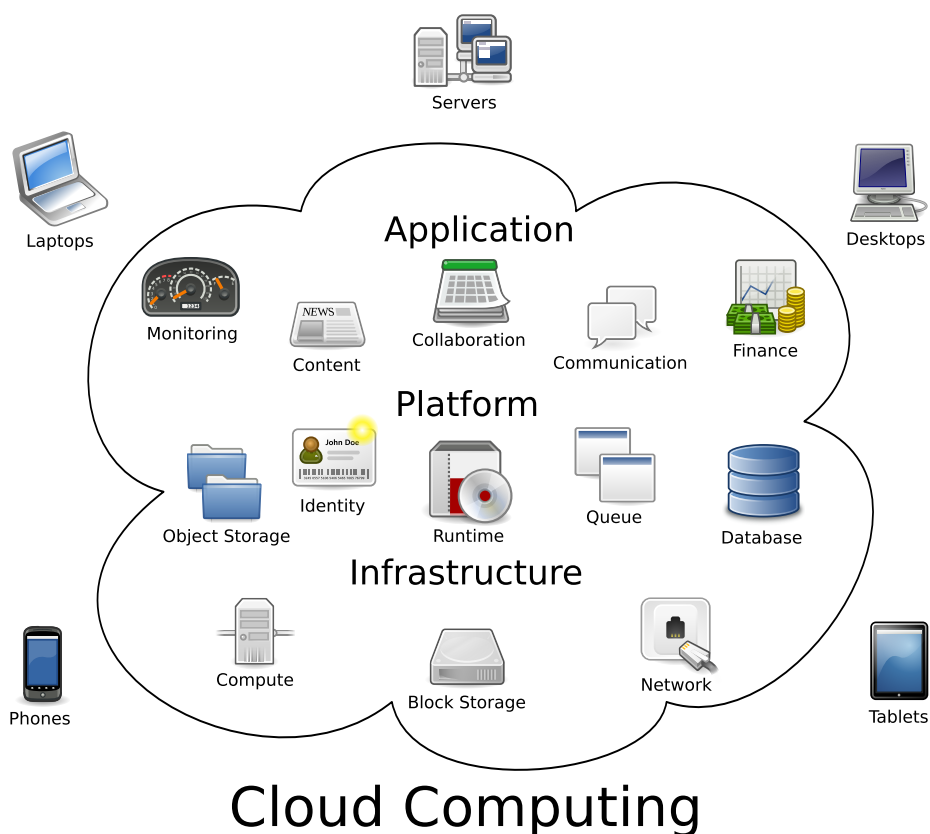
Najprostszym i najstarszym rodzajem usług w "chmurze" jest *Kolokacja*. Polega ona na dzierżawieniu miejsca w serwerowni, prądu, klimatyzacji i połączenia do Internetu. W gestii dzierżawcy jest zaopatrzenie się w sprzęt, zabezpieczenia, system operacyjny kończąc na oprogramowaniu i aplikacjach.

Bardziej rozbudowanym rodzajem chmury jest *IaaS* (ang. Infrastructure as a Service), czyli rozszerzenie kolokacji o zapewnienie sprzętu przez dostawcę. To dostawca udostępnia sprzęt, niejednokrotnie również zabezpieczenia. Zadaniem odbiorcy jest dostarczenie systemu operacyjnego, oprogramowania i aplikacji. Pewną grupą usług, kwalifikowaną do tego zbioru, mogą być serwery dedykowane, które u wielu dostawców można kupić od dawna. Obecnie jednak zakres możliwości serwerów dedykowanych został poszerzony, dzięki wykorzystaniu technologii wirtualizacji. Za jej sprawą, użytkownik otrzymuje funkcjonalną maszynę wirtualną, którą może skonfigurować w dowolny sposób wykorzystując zdalny dostęp.

Jak wynika z zestawienia rodzajów chmur, usługa *PaaS*, a wraz z nią również *GAE* oferuje dużo więcej ponad standardową infrastrukturę sprzętową. Użytkownik musi się skupić jedynie na dostarczeniu oprogramowania pracującego w określonym środowisku roboczym.

Polityką chmur typu *PaaS* jest płacenie tylko za aktualne zużycie zasobów. Rozliczani jesteśmy więc, za moc obliczeniową serwerów i czas ich

pracy, ilość zapytań do bazy, a także za bieżące wykorzystanie dysku. Dodatkowo architektura dynamicznie zwiększa ilość zasobów, jak również instancji serwerów aktualnie wykorzystywanych, by jak najbardziej zmniejszyć czas odpowiedzi strony (Rys. 3.2). Całość można dynamicznie kontrolować balansując koszty do optymalnej szybkości działania aplikacji.



Rys. 3.2: Komponenty wchodzące w skład architektury w chmurze

W wypadku GAE w ramach darmowych limitów do dyspozycji dostajemy 28h instancji. Oznacza to, że w wypadku pojedynczej najwolniejszej instancji będziemy mogli z niej korzystać 28h (600Mhz), w wypadku szybszej instancji (1200Mhz) czas ten zostanie 2-krotnie zmniejszony. W wypadku najszybszej instancji (2400Mhz) czas ten ulega 4-krotnemu skróceniu. Do oceny użytkownika zostaje decyzja, z jakiej instancji chce korzystać i oczywiście, jak bardzo ilość serwerów będzie się zwiększała przy natężeniu ruchu. Jeśli darmowa

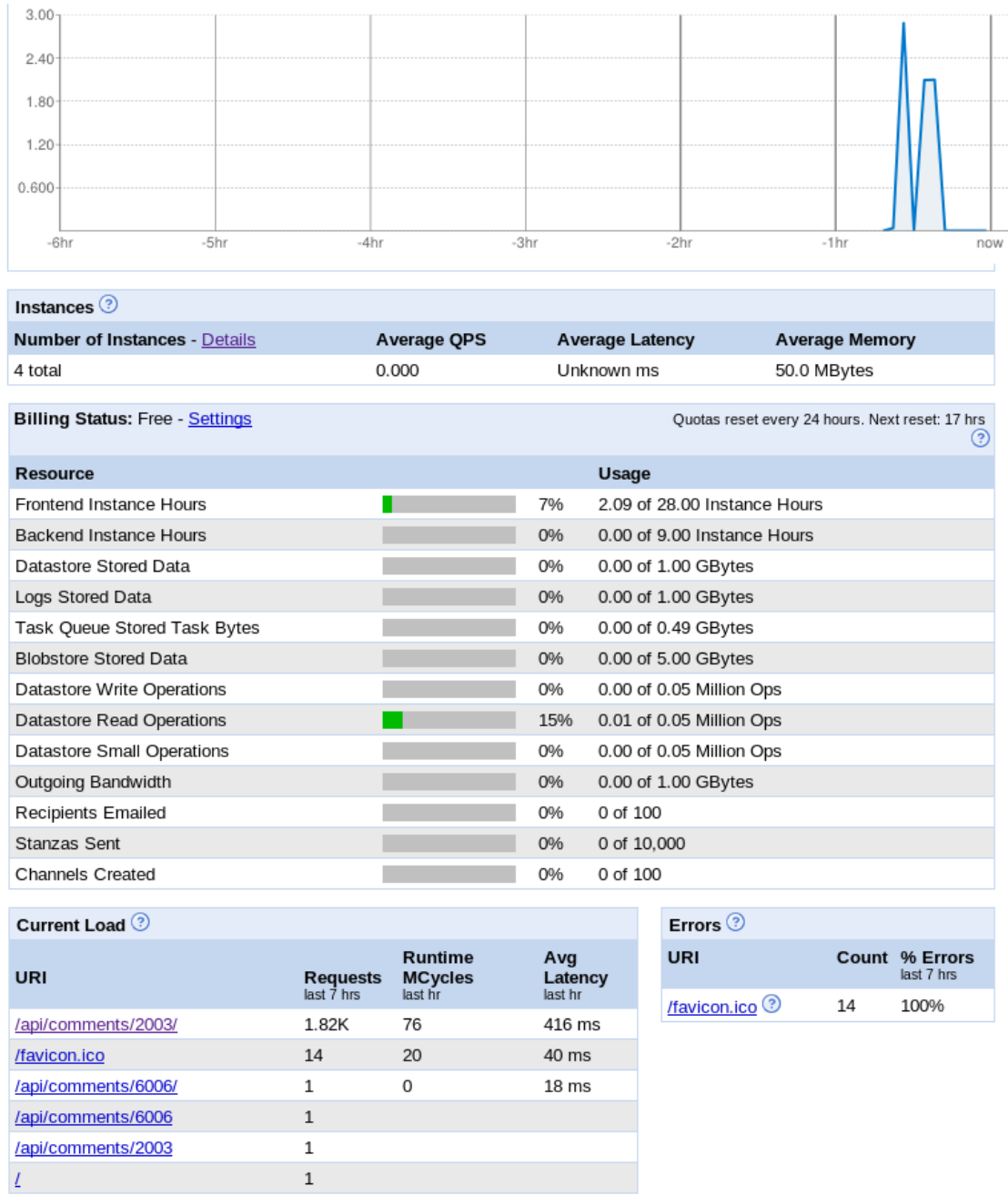
oferta GAE jest niewystarczająca, w każdej chwili można przejść na opcję płatną, w której określamy tygodniowe / miesięczne limity płatności. Jedną z ważnych zmian w porównaniu z normalnymi hostingami jest taryfikator operacji bazodanowych. Wyróżniono ceny za odczyt i zapis do bazy, odpowiednio 0.10\$ za 100,0000 operacji zapisu i 0.07\$ za podobną ilość operacji odczytu. System liczenia operacji zapisu opera się na dodatkowych operacjach, związanych z indeksowaniem danych. Przykładowo pobranie pojedynczego rekordu z bazy to koszt jednego odczytu, ale wykonanie zapytania zwracającego 20 rezultatów to koszt 21 operacji odczytu. W wypadku zapisu do bazy nowej encji oznacza to 2 operacji zapisu + 2 operacje zapisu na każdy indeks dla pola + 1 zapis dla wartości indeksu złożonego.

Rysunek 3.3 przedstawia poglądowy widok bieżących statystyk aplikacji, udostępnianych przez GAE. Użytkownik ma dostęp do stanów bieżących limitów, a także szczegółowych wykresów wydajności aplikacji określanych na podstawie różnych czynników (zajętość pamięci, ilość operacji na sekundę, wykorzystanie instancji serwerów). Bardzo przydatny jest również wykres bieżącej aktywności aplikacji. W ten sposób można szybko ocenić, czy aplikacja jest obecnie pod dużym obciążeniem, a także porównać, czy na przykład wdrożone zmiany poprawiły jej wydajność.

3.2 Zarządzanie wydajnością aplikacji na platformie *GAE*

Jak zostało omówione we wstępie w podrozdziale 3.1, olbrzymią zaletą rozwiązań *PaaS* jest możliwość kompleksowego zarządzania aplikacją z poziomu przeglądarki i udostępnionego interfejsu klienta. Rysunek 3.4 pokazuje jak to wygląda w praktyce. Za pomocą listy dostępnej w sekcji *Frontend Instance Class* określamy jak bardzo wydajnej instancji chcemy użyć dla naszej aplikacji. Każda z instancji jest opisana za pomocą umownego symbolu, częstotliwości pracy procesora oraz maksymalnej dostępnej pamięci. Przy obecnym ustawieniu każda utworzona instancja serwera posiadać będzie parametry in-

Rozdział 3. Architektura aplikacji



Rys. 3.3: Wygląd panelu statystyk w ramach usługi *GAE*

stancji *F1*.

W celu zapewnienia wydajności aplikacji na określonym poziomie, ważne jest wyważenie ustawień dwóch suwaków, odpowiednio *Max Idle Instances* oraz *Min Pending Latency*. Pierwszy suwak określa początkową ilość instancji dostępną zawsze (tak zwani *rezydenci*). W wypadku kolejnych instancji, ich ilość jest warunkowana parametrami określonymi przez punkty brzegowe drugiego suwaka. Określa on jak długo *dyspozytor* żądań poczeka, nim zdecyduje o odesłaniu żądania do innej instancji serwera. Jest to bardzo logiczne założenie, że jeśli żądanie wykonuje się długo, prawdopodobnie strona jest obciążona, dlatego trzeba odciążyć serwery i rozłożyć ruch na nowo dodane.

The screenshot shows the 'Performance' settings for a Google App Engine application. It includes three main sections: 'Frontend Instance Class', 'Idle Instances', and 'Pending Latency'. The 'Frontend Instance Class' is set to 'F1 (600MHz, 128MB)'. The 'Idle Instances' slider is set to 'Automatic' (range 2-4). The 'Pending Latency' slider is set to 'Automatic' (range 990ms to Automatic). There is also a 'PageSpeed Service' section with an 'Enable PageSpeed Service' checkbox and a 'Flush Cache' button. A 'Save Settings' button is at the bottom.

Performance

Frontend Instance Class: F1 (600MHz, 128MB) ▼

Adjusting your Frontend Instance Class will affect all frontend versions of your application. Your frontends will have more memory and processing power, but also consume frontend instance hours at an increased rate, which will lead to increased costs. For example an F2 consumes instance hours at twice the rate of an F1. [Learn more.](#)

Idle Instances: (2 – 4)

The Idle Instances slider allows you to control the number of idle instances available to the default version of your application at any given time. Idle Instances are pre-loaded with your application code, so when a new Instance is needed, it can serve traffic immediately. You will not be charged for idle instances over the specified maximum. A smaller number of idle Instances means your application costs less to run, but may encounter more startup latency during load spikes. [Learn more.](#)

Min Automatic 50 100 500 Automatic Max

Pending Latency: (990ms – Automatic)

The Pending Latency slider controls how long requests spend in the pending queue before being served by an Instance of the default version of your application. If the minimum pending latency is high App Engine will allow requests to wait rather than start new Instances to process them. This can reduce the number of instance hours your application uses, but can result in more user-visible latency. [Learn more.](#)

Min Automatic 500ms 1s 7.5s Automatic Max

PageSpeed Service:

The PageSpeed Service automatically optimizes and caches your site for improved performance. To configure advanced settings, edit the pagespeed section in your app.yaml file.

☐ Enable PageSpeed Service Flush Cache

Save Settings

Rys. 3.4: Kontrola wydajności aplikacji w *GAE*

Jeśli skonfigurowano serwer zgodnie z rysunkiem 3.4, w zakładce *Instances* powinny zostać wyświetlone wszystkie aktualnie wykorzystywane instancje serwerów. Dwie z nich to instancje stałe - rezydenci, więc ich czas pracy jest równy dobie. Dodatkowo widoczne są również instancje dynamiczne (Rys. 3.5), czyli takie, które używane są doraźnie. Zgodnie z ustawieniami, łączna ilość instancji nie przekroczy górnego maksimum czyli czterech serwerów.

W wypadku instancji dynamicznych należy mieć na uwadze, że na opłatę składa się sumaryczny czas pracy oraz 15-minutowy czas opłaty rozruchowej. Dla przykładu jeśli dynamiczna instancja pracuje przez pięć minut to koszt wynosi $0.08/60 \times (15 + 5) = 2.6\text{¢}$.

Version: staging ▾

Total number of instances	Average QPS*	Average Latency*	Average Memory
3 total (2 Resident)	0.067	1992.8 ms	36.6 MBytes

Instances ?								
QPS*	Latency*	Requests	Errors	Age	Memory	Logs	Availability	Shutdown
0.033	6725.0 ms	1	0	0:00:21	36.5 MBytes	View Logs	Resident	<button>Shutdown</button>
0.033	5156.0 ms	1	0	0:00:14	36.1 MBytes	View Logs	Resident	<button>Shutdown</button>
0.133	19.0 ms	14	0	0:13:42	37.3 MBytes	View Logs	Dynamic	<button>Shutdown</button>

* QPS and latency values are an average over the last minute.

Rys. 3.5: Nadzór i zarządzanie instancjami serwerów w *GAE*

W zakładce *Instances* można również zobaczyć takie statystyki jak ilość operacji na minutę (*Average QPS*), średnie zużycie pamięci (*Average Memory*) czy średni czas przetwarzania żądania (*Average Latency*). Dodatkowo jeśli któraś z instancji uległa przeciążeniu, w skutek czego nie reaguje, można ją zamknąć ręcznie, klikając przycisk *Shutdown*.

3.3 Nierelacyjne bazy danych *NoSQL*

Od początku ery informatyzacji, informacja i jej przechowywanie było najważniejszym zadaniem powierzonym komputerom. Bazy danych muszą zarówno przechowywać dane (zapis), jak i udostępniać interfejs służący ich odcytowi, a nawet wyszukiwaniu w oparciu o zadane kryteria. Dotychczas wykorzystywano głównie relacyjne bazy danych, które łączyły pojedyncze tabele w różnego rodzaju asocjacje. Wcześniej jednak konieczne jest zdefiniowanie określonego schematu bazy danych, który będzie obowiązywał i nakładał ograniczenia na dane.

Obecnie jednak bardzo popularne stają się bazy bez określonej struktury, niesamowicie skalowalne, a przy tym dostęp do nich odbywa się przy pomocy *REST*owego API. Wyróżniono kilka konwencji:

- *Key Value Stores* - HDOOP, BigData,
- *Graph Databases* - Neo'tj, AllegroGraph,
- *BigTables* - HBase, Cassandra,
- *Document Database* - MongoDB, CouchDB, SimpleDB

W projektach *e-commerce* największym zastosowaniem cieszą się bazy dokumentowe, skupiające kolekcje obiektów - dokumentów. Każdy z nich może zawierać odpowiednie metadane i słowa kluczowe. Każdy dokument posiada dowolną definicję i może być dowolnie zagnieżdżany i łączony z innymi dokumentami.

Wraz z wykorzystaniem nie-relacyjnych baz danych, zmienił się zupełnie pomysł na organizację danych. Siłą baz danych NoSQL jest brak ściśle określonej struktury, przez co schemat danych może się dowolnie zmieniać w trakcie rozwoju aplikacji. Innym ważnym powodem wykorzystania bazy *Google Datastore*, oferowanej przez GAE jest szybkość i skalowalność. W odróżnieniu od zwykłego hostingu, baza danych w GAE może być rozproszona na dowolną ilość instancji.

Początkowo osoby korzystające z baz nierelacyjnych zarzucały takiemu rozwiązaniu następujące wady:

- problem z ponownym wykorzystaniem kodu dla innych rozwiązań nie relacyjnym nawet kiedy implementacje są podobne,
- nie można wykorzystać kodu dla relacyjnej bazy danych, nawet jeśli schemat pozostałby nietknięty,
- konieczność zarządzania indeksami, a także problem denormalizacji,

- konieczność samodzielnej implementacji łączenia rezultatów wielu zapytań,
- niektóre z rozwiązań są zintegrowane ze specyficznym dostawcą (*App Engine*, *SimpleDB*), więc użytkownik nie ma możliwości migracji na inną platformę bez dużego nakładu pracy.

W stworzonej aplikacji wykorzystano framework *Django-nonrel* dodający obsługę baz NoSQL do frameworka *Django*. W ten sposób możliwe jest wykorzystanie abstrakcji bazodanowej i tworzenie rozwiązań niezależnie od technologii bazy. Wszystko to możliwe jest dzięki wbudowanemu narzędziu *ORM* (ang. *Object-Relational Mapping*), które udostępnia obiektowy interfejs dostępu do danych. W ten sposób pisana logika nie jest zależna od konkretnej platformy lub rozwiązania składującego.

3.3.1 Specyfika *Google Datastore*

Google App Engine posiada swoją wewnętrzną bazę danych *Datastore*, która przechowuje obiekty danych, znane jako *encje*. Każda encja ma jedną lub więcej pól różnego typu podobnie, jak w relacyjnych bazach. Każda encja jest identyfikowana po jej typie, ma to na celu katalogowanie ich z względu na pełnioną funkcję logiczną. Jest to pewna analogia do tabel, ale raczej należy to postrzegać jako kontenery na luźno powiązane obiekty. Każda encja ma klucz, który unikalnie identyfikuje zasób niezależnie od typu [12].

Datastore może wykonywać wiele operacji w pojedynczej transakcji. Z założenia transakcja nie może się zakończyć do czasu zakończenia ostatniej operacji w jej ramach. Oczywiście w wypadku niepowodzenia transakcja zostałaby wycofana, co jest szczególnie przydatne w wypadku rozproszonych aplikacji webowych, gdzie wielu użytkowników może mieć dostęp lub wykonywać operacje na tym samym obiekcie.

W GAE głównym miejscem składowania jest HRD (*High Replication Datastore*). W skrócie, jest to miejsce, gdzie dane są replikowane do wielu centrów danych wykorzystując system oparty na algorytmie *Paxos*. Taki zabieg zapewnia wysoki poziom dostępności dla operacji odczytu i zapisu. Jedyny problem może występować ze spójnością danych ze względu na czas replikacji do poszczególnych węzłów.

W odróżnieniu od standardowych baz danych, GAE wykorzystuje architekturę rozproszoną, aby zapewnić automatyczne zarządzanie skalowaniem nawet w wypadku bardzo obszernych zbiorów danych. Pomimo podobieństw w mechanizmach dostępu do danych między poszczególnymi bazami, istotną różnicą jest sposób opisywania relacji pomiędzy obiektami. Na przykład encje tego samego typu mogą mieć różne pola, z kolei różne encje mogą mieć właściwości tej samej nazwy, ale przechowywać wartości różnego typu. Te unikalne cechy sprawiają, że projektowanie i zarządzanie danymi różni się znacząco w wypadku tej platformy. Ponieważ GAE jest stworzone do skalowania, w wypadku wzmożonego zapisu, *Datastore* automatycznie rozproszy dane, jeśli jest to konieczne na więcej instancji. Również operacje odczytu są dostosowane do szybkiego działania, ponieważ jedyne dozwolone zapytania to takie, które działają niezależnie od wielkości danych.

Oznacza to, że wyszukiwanie w zbiorze 100 elementowym zajmie dokładnie tyle samo czasu, co w zbiorze zawierającym ich milion. *Datastore* robi również duży użytek z indeksów dla kolekcji danych. Istotną cechą jest fakt, że w procesie tworzenia aplikacji, biblioteka deweloperska bada, które zapytania wymagają dodania nowych indeksów i robi to za użytkownika w specjalnym pliku definicji indeksów. W ten sposób najważniejsze zapytania wykonywane przez aplikację posiadają dane uporządkowane w odpowiednich indeksach. Jeśli w aplikacji wykonywane są operacje sortowania danych po jakiejś kolumnie to w GAE wyniki sortowania rosnącego i malejącego będą przechowywane w zupełnie różnych indeksach, nawet jeśli dotyczą tego samego pola.

3.4 Aplikacja zorientowana na usługi

Inną ważną cechą tworzonej aplikacji jest wyszczególnienie usług (*webservicew*), które pomogą w wykonywaniu operacji na modelu przy pomocy metod protokołu HTTP. Będą to więc popularne w dzisiejszych czasach usługi REST. Ze względu na specyfikę aplikacji i duży udział języka JavaScript w tworzonej logice, w większości wypadków usługi te będą zwracały dane w notacji *JSON*.

Zaletą takiego podejścia, określanego później jako *Service Oriented Architecture* jest wydzielenie pewnych usług, zawierających pewną część / zakres funkcjonalny. W ten sposób aplikacja jest podzielona na grupę takich usług, z których każda realizuje tylko i wyłącznie swoje zadanie. Dzięki takiej dekompozycji, możliwym staje się wykorzystanie wspomnianych usług, wszędzie tam gdzie jest to potrzebne. Inną zaletą jest łatwość testowania takich usług, ponieważ nie jest konieczne stworzenie całego zestawu wykonawczego, a jedynie test konkretnego działania, spełniającego określone cele.

Ponieważ komunikacja między *front-endem* aplikacji, napisanym w języku *PHP*, a zapleczem umiejscowionym na platformie GAE odbywa się poprzez webserwisy, istotne było wyróżnienie następujących usług:

- usługa zwracająca dostępne kategorie, a także kategorie dla danej podkategorii,
- usługa wyświetlająca książki z danej kategorii,
- usługa wyświetlająca komentarze do danej książki,
- usługa dodająca komentarz z oceną do książki,
- usługa wyszukująca książki na podstawie słów kluczowych, tytułu lub kategorii

Każda usługa zwraca obiekt *JSON*, który następnie jest analizowany przez kod *JavaScript*. Takie podejście czyni przetwarzanie strony niezwykle

wydajnym, bo obciążenie jest dzielone na dwie aplikacje.

3.4.1 AJAX, a restrykcje bezpieczeństwa kodu *JavaScript*

Na początku projektowania zaistniał jeden problem wynikający z komunikacji przy użyciu technologii AJAX. Problem *same origin policy* jest bardzo istotnym założeniem bezpieczeństwa dotyczącym języków interpretowanych po stronie przeglądarki, takich jak *JavaScript*. Polityka bezpieczeństwa zezwala skryptom uruchamianym w ramach tej samej strony na dostęp do metod i właściwości powiązanych obiektów *JavaScript* bez specjalnych restrykcji. Z drugiej strony polityka ta odmawia dostępu do większości metod i własności podczas takich prób na innych stronach (inna domena, subdomena).

Mechanizm ten odciska znaczące piętno na nowoczesnych aplikacjach internetowych, które w znacznym stopniu polegają na przykład na danych uwierzytelniających, zapisanych w sesji *HTTP cookie*, jak również wykorzystujących technologię AJAX do pobierania rezultatów działania usług.

JSONP (JSON with padding)

JSONP stanowi dopełnienie bazowego formatu danych *JSON*. Jest sposobem na obejście problemu dostępu do danych z różnych domen związanych z *same origin policy*. Zgodnie z zasadą, strona umieszczona pod adresem `server1.example.com` nie może normalnie połączyć się lub komunikować z serwerem innym niż `server1.example.com`. Jedynym wyjątkiem od tej reguły jest HTML-owy element `<script>`.

Obchodząc regułę bezpieczeństwa, możliwe jest pobranie zwracanego kodu *JavaScript*, który operuje na dynamicznie wygenerowanych danych w formacie *JSON*. Tak naprawdę takie żądanie nie zwraca wcale obiektu *JSON*, tylko dynamicznie utworzony kod *JavaScript* i odnosi się do metod istniejących na stronie wykonującej żądanie.

Przykładowy kod obrazujący ten sposób widoczny jest na listingach 3.2 oraz 3.3. Pierwszy skrypt pokazuje *front-endową* część rozwiązania, drugi natomiast zwraca kod *JavaScript*, który wykona metodę `parseJSON` z przetworzonym obiektem *JSON*, zawierającym wynik działania zapytania bazodanowego jako parametr funkcji. Następnie metoda `parseJSON` iterując w pętli po elementach obiektu *JSON* (w języku *JavaScript* obiekty są również tablicami, więc można stosować iteratory tablicowe), wyświetli po kolei wartość dla każdego klucza obiektu w ramach poszczególnych wierszy wyniku zapytania.

Oczywiście omawiane zagadnienie sprawia trochę problemów w praktyce, ale korzystając z wewnętrznej implementacji oferowanej przez *jQuery*, cała praca wykonywana jest za programistę. Dodatkowo, oprócz standardowych zapytań typu `GET`, można wykonywać również inne metody protokołu *HTTP*.

W późniejszym etapie projektowania aplikacji okazało się, że *JSONP* pomimo wielu zalet, ma również kilka wad. Z założenia jest on umieszczany przy pomocy znaczników *HTML*, przez co jedyna obsługiwana przez niego metoda, to *HTTP GET*. Jest to niewystarczające do realizacji bardziej złożonej logiki obsługi webserwisów, dlatego zastosowano dodatkowo technologie *CORS* (ang. Cross-origin resource sharing).

Dzięki wspólnej pracy WAWG (*Web Applications Working Group*), wspólnie z W3C (*World Wide Web Consortium*), udało się opracować nową rekomendację standardu *Cross-Origin Resource Sharing*. W ten sposób możliwe jest kontrolowanie dostępu do poszczególnych stron. Dzięki odpowiednim nagłówkom *HTTP* możliwe jest zezwolenie obiektowi *XMLHttpRequest* na dostęp do aplikacji, udostępniających odpowiednie nagłówki (Listing 3.1). W ten sposób udostępniając poprzez serwer *WWW* poniższe nagłówki, można zezwolić jednej aplikacji (dostępnej pod adresem *http://foo.example*) na dostęp do tej, która przy użyciu nagłówków `Access-Control-Allow-Origin`

oraz `Access-Control-Allow-Methods` deklaruje współpracę.

Listing 3.1: Realizacja *CORS* w praktyce

```
1 Access-Control-Allow-Origin: http://foo.example
2 Access-Control-Allow-Methods: POST, GET, OPTIONS
```

Listing 3.2: Kod JavaScript realizujący JSONP

```
1 <html>
2 <head>
3 <script src="http://someserver.com/jsonService?category_id
   =5"></script>
4 </head>
5 <body>
6 <script>
7
8 function parseJSON(object)
9 {
10  for(key in row)
11  {
12    document.write('Wiersz zapytania nr' + key + 1);
13    for(subkey in row) {
14      document.write(key + " = " + row[subkey]);
15    }
16  }
17 }
18
19 </script>
20 </body>
21 </html>
```

Listing 3.3: Logika po stronie zewnętrznego serwera

```
1 <?php
2
3 $categoryId = (int) $_GET['category_id'];
4
```

```
5 $sql = sprintf('SELECT * from category where category_id =  
    %d', $categoryId);  
6  
7 $stmt = $con->query($sql); // dla uproszczenia pominięto  
    łączenie się z bazą danych  
8  
9 $results = array();  
10  
11 while(false !== $row = $stmt->fetch())  
12 {  
13  
14     $results[] = $row;  
15  
16 }  
17  
18  
19 echo sprintf("parseJSON(%s);", json_encode($results));
```

Podsumowując, założeniem tworzonej aplikacji jest uzyskanie jak największej separacji logiki na zbiór wyspecjalizowanych usług. Usługi te zostaną opakowane w webservisy w architekturze *REST*, co zapewni możliwość komunikacji z wykorzystaniem kodu *JavaScript*. W ten sposób aplikacja została podzielona na części, z których każda ma zapewnioną skalowalność dzięki architekturze *Google Application Engine*.

Rozdział 4

Optymalizacja kodu klienta

Ponieważ głównym medium komunikacji w Internecie jest przeglądarka, konieczna jest odpowiednia optymalizacja kodu klienta. Na początku warto zacząć od wyboru odpowiedniego nośnika informacji. Jak wiadomo, nośnikiem informacji w wypadku stron internetowych jest język znaczników *HTML* (ang. *HyperText Markup Language*). Obecnie najbardziej popularne są trzy wersje standardu, określane jako *HTML 4*, *XHTML 1.0* oraz nowy standard *HTML 5*. Każda wersja ma swoją specyfikę, jednak należy nadmienić, że tylko *HTML 5* pozbawiony jest starych naleciałości tego języka (znaczniki związane z formatowaniem, a nie semantyką, różna implementacja specyfikacji) i powoli staje się standardem na rynku.

Wśród przeciwników wykorzystywania nowego standardu dominują opinie, że nie jest on wspierany we wszystkich przeglądarkach. Warto jednak określić kilka istotnych faktów, związanych z obecnie wykorzystywanymi przeglądarkami. Według statystyk na 5 lutego 2012, udział przeglądarek Internet Explorer 6 i 7 w rynku wynosi odpowiednio **0,69%** oraz **2,34%**. O ile przeglądarkę w wersji 7 i powyżej należy mieć jeszcze na uwadze, to IE6 można uznać już za przestarzałą i stopniowo ograniczać ilość czasu poświęcanego na implementację kompatybilnej wstecz witryny internetowej.

Dobrym zwyczajem jest jednak zapewnienie witrynie jak najlepszego dostosowania do obecnych na rynku przeglądarek, dlatego rewolucyjnym pomysłem wykazał się Nicolas Gallagher, Paul Irish oraz Divya Manian, czyli czołowi projektanci, programiści takich firm, jak Twitter, Opera, Chrome. Stworzyli oni bibliotekę *HTML5 Boilerplate*. Stanowi ona fundament tworzenia dokumentów HTML/CSS/JS, wykorzystując najnowszą wiedzę w dziedzinie optymalizacji *front-endu* aplikacji. Wykorzystanie tej biblioteki zapewni znormalizowanie wyglądu, zachowania oraz wydajności każdej aplikacji, chcącej używać technologii HTML5 i nie tylko. Biblioteki zapewniają kompatybilność z innymi przeglądarkami (nawet IE6), zbiór reguł optymalizujących czas reakcji strony, dzięki kompresji i buforowaniu zasobów. Możliwość ukrywania pewnych komponentów strony, które nie są wspierane przez niektóre przeglądarki, na przykład znacznik `<canvas>` lub nowe znaczniki do prezentacji plików multimedialnych. Biblioteka zawiera również wiele udogodnień związanych z wyświetlaniem stron na ekranie telefonów komórkowych.

Stworzona w ramach pracy aplikacja wykorzystuje udogodnienia oferowane przez *HTML5 Boilerplate*. Dodatkowo, aplikacja wykorzystuje bibliotekę *Twitter Bootstrap*, której celem jest zapewnienie funkcjonalnych komponentów, które można wykorzystać do łatwej i spójnej prezentacji treści. Jedną z zalet wykorzystania tego rozwiązania, jest oparcie szablonu strony na *siatce* (ang. *grid*). W ten sposób wszystkie elementy są wymiarowane w sposób jednaki. Dodatkowo, udostępniana jest możliwość tworzenia dynamicznych szablonów, które adaptują się do rozdzielczości lub po prostu wielkości okna. Jest to cenne udogodnienie dla posiadaczy telefonów komórkowych, ponieważ nie trzeba poświęcać dodatkowego czasu na adaptację do wersji mobilnej.

Twitter Bootstrap zawiera zbiór elementów graficznych i funkcjonalnych, takich jak przyciski, okienka informacyjne, dynamicznie animowane galerie, komponenty autouzupełniania treści pól, komponenty nawigacyjne i wiele innych typowych dla specyfiki aplikacji webowych elementów. Całość jest spójna i pozwala na zapewnienie jednolitego interfejsu na całej stronie. Dodatkowo, podobnie jak w wypadku *HTML5 Boilerplate*, użytkownik ma za-

pewnioną kompatybilność wstecz dla prawie wszystkich dostępnych na rynku rozwiązań.

Ponieważ biblioteka realizuje założenia semantyczne języka *HTML5*, ilość kodu potrzebna do stworzenia strony jest naprawdę niewielka. W ten sposób dokumenty są lekkie i szybciej się wczytują. Dodatkowo, dzięki możliwościom CSS (kaskadowe arkusze styli) w wersji 3, możliwe jest tworzenie atrakcyjnych wizualnie efektów bez konieczności tworzenia przez projektantów dodatkowych grafik. W ten sposób ilość żądań jest redukowana do niezbędnego minimum.

Rysunek 4.1 przedstawia wygląd interfejsu użytkownika po zastosowaniu wspomnianych wcześniej technologii. Jak widać strona dostosowuje się do różnych szerokości przeglądarki, dzięki wykorzystaniu płynnych szablonów (ang. *fluid templates*).

4.1 Przetwarzanie danych pochodzących z *webserwisów*

W celu rozszerzenia możliwości języka *JavaScript* zastosowano framework *jQuery*. Pomaga on w zapewnieniu kompatybilności tworzonego kodu dla różnych przeglądarek. Oczywiście włączając przeglądarkę IE6. Framework *jQuery* jest szczególnie przydatny ze względu na zaawansowane funkcjonalności do obsługi zapytań asynchronicznych AJAX. Jest to szczególnie ważne, ponieważ razem z żądaniem i jego specyfikacją konieczne jest, by wysłać również zahaszowane dane autoryzacyjne. Całość opiera się na pierwszym zalogowaniu, podczas którego do sesji użytkownika trafia kod autoryzacyjny użyty do pomyślnego zalogowania. W ten sposób usługi wiedzą, że dostęp do nich został zweryfikowany. Jego brak poskutkowałby błędem autoryzacji - kod 401. W przyszłości można tę funkcjonalność rozszerzyć o autoryzację wykorzystując standard *OAuth 2.0*, zaimplementowany między innymi w Twitterze lub w *Facebook Graph API*.

czeK

Hi (tworzenieweb@gmail.com) ▾

BOOczeK

Elektroniczna księgarnia na miarę twoich potrzeb

Zobacz nowości »

Effective Java Java

Ocena: brak oceny

Język Java jest językiem obiektowym z dziedziczeniem jednobazowym. Wewnątrz każdej metody korzysta on ze zorientowanego na instrukcje stylu kodowania. Aby dobrze poznać jakikolwiek język, należy nauczyć się posługiwać jego regułami, zasadami i składnią -- podobnie jest z językiem programowania. Jeśli chcesz zyskać możliwość efektywnego programowania w języku Java, powinieneś poznać struktury danych, operacje i udogodnienia, oferowane przez biblioteki standardowe, a także często stosowane i efektywne sposoby tworzenia kodu. Całą potrzebną Ci wiedzę znajdziesz właśnie w tym podręczniku. W książce "Java. Efektywne programowanie" w sposób zrozumiały i klarowny przedstawiono zasady opisujące mechanizmy używane w najlepszych technikach programowania. Ten podręcznik podpowie Ci, jak najbardziej racjonalnie korzystać z języka Java oraz jego podstawowych bibliotek. Dowiesz się, jak stosować wyjątki przechwytywalne i wyjątki czasu wykonania, poznasz także zalety stosowania statycznych klas składowych. Opanujesz metody sprawdzania poprawności parametrów i projektowania sygnatur oraz wszelkie instrukcje, które pozwolą Ci na wydajne i profesjonalne programowanie.

Authors: Joshua Bloch,

Dodaj komentarz

Tytuł

Ciekawa książka

Ocena

4 ▾

Chyba jedna z lepszych publikacji dotyczących Javy na rynku

Dodaj

Rys. 4.1: Ekran przedstawiający interfejs użytkownika stworzonej księgarni internetowej

Listing 4.1 przedstawia implementację logiki realizującej wysyłanie komentarzy do webserwisu, odpowiedzialnego za dodawanie komentarzy. Do poprawnego działania konieczna jest znajomość nagłówka autoryzacyjnego, więc musi on być przekazany do obiektu *XMLHttpRequest* jeszcze przed wysłaniem zapytania. Dodatkową funkcjonalnością jest wyświetlanie uaktualnionych komentarzy zaraz po zapisaniu danych. Odpowiedzialna jest za to funkcja `reloadComments()`, która dodatkowo wykorzystuje nową możliwość języka *JavaScript*, możliwą dzięki odpowiednim wtyczkom. Polega ona na tworzeniu szablonów, które są później łączone z danymi, tak jak to odbywa się na przykład w innych technologiach po stronie serwera. Mowa tu o *Jquery Templates*, które mogą w przyszłości trafić do standardu języka *JavaScript*.

Listing 4.1: Kod odpowiedzialny za realizację dodawania komentarzy

```
1 <script id="commentTemplate" type="text/x-jquery-templ">
2     <div class="row-fluid">
3         <h4>${title} {{html $item.getStars()}}</h4>
4         <p>${content}</p>
5         <p>${date}${user.first_name} ${user.last_name} ($
            {user.username})</p>
6     </div>
7 </script>
8
9 <script>
10     var url = "<?php echo sfConfig::get('app_gae') . '
            comments/' . $book['id'] . '/' . '?'>";
11
12     var reloadComments = function() {
13         $("#comments").html('');
14         $.getJSON(url, function(data) {
15
16             /* Render the template with the data */
17             $( "#commentTemplate" ).tmpl( data, {
18                 getStars: function( ) {
19                     var str = '';
```

```
20         for(i=0; i < this.data.grade; i++) {
21             str += '<i class="icon-star"></i>';
22         }
23
24         for(j=i; j < 5; j++) {
25             str += '<i class="icon-star-empty"></i>';
26         }
27
28         return str;
29     }
30 }).appendTo( "#comments" );
31
32     });
33
34 }
35
36 $(function() {
37
38     reloadComments();
39
40
41     $('#commentForm').submit(function (e){
42         e.preventDefault();
43
44         $.ajax({
45             url: url,
46             data: $(this).serialize(),
47             xhrFields: {
48                 withCredentials: true
49
50             },
51             type: 'POST',
52
53             beforeSend: function(xhr) {
54                 xhr.setRequestHeader("Authorization", "
                    <?php echo $sf_user->getAttribute('
                    user_authorization', null, '
                    sfGuardSecurityUser') ?>");
```



```
55         },
56
57         success: function(data) {
58             reloadComments();
59         }
60     });
61
62 });
63
64 });
65 </script>
```

Jak wynika z listingu 4.1, metoda `reloadComments()` odpowiedzialna jest za przetwarzanie danych komentarzy powiązanych z konkretną książką i wyświetlenie odpowiedniej ilości "gwiazdek" na podstawie oceny użytkownika. Jest to przykład odciążenia *back-endu* aplikacji, na rzecz przetwarzania po stronie klienta.

Podobny sposób przetwarzania danych wykorzystany jest do pobierania listy kategorii dostępnych dla książek (4.2). Do prezentacji pojedynczego wpisu stworzono szablon. W ten sposób w miejsca oznaczone symbolami `$title` i podobnymi, wstawiane są dane z obiektu *JSON*.

Listing 4.2: Kod odpowiedzialny za wyświetlanie kategorii

```
1 <script id="categoryTemplate" type="text/x-jquery-tmpl">
2     <li>
3         <a href="{{$item.url()}}">${name}
4             <span class="badge badge-success">${count}</span>
5         </a>
6     </li>
7 </script>
8
9
10 <script>
11     var urlCat = "<?php echo sfConfig::get('app_gae'); ?>
    categories/";
```

```
12     var link = "<?php echo url_for('@category_slug?slug=' .  
13         'slug'); ?>";  
14  
15     $(function() {  
16  
17         $.getJSON(urlCat, function(data) {  
18  
19             /* Render the template with the movies data */  
20             $( "#categoryTemplate" ).tmpl( data, {  
21                 url: function() {  
22                     return link.replace('slug', this.data.  
23                         slug);  
24                 }  
25             }).appendTo( "#category" );  
26         });  
27  
28     });  
29  
30 </script>
```

4.2 Optymalizacja kodu *HTML* oraz zasobów

Optymalizacja kodu *HTML* ma na celu zapewnienie szybkiego przetwarzania odpowiedzi serwera przez przeglądarkę. Jedną z ważniejszych optymalizacji jest usunięcie definicji stylów zawartych w znacznikach, na rzecz ujednoliconych definicji w arkuszu stylów. Optymalizacja ta wprowadza uporządkowanie do tworzonego kodu, zmniejsza rozmiar kodu *HTML* oraz przyspiesza renderowanie strony przez przeglądarkę. Łatwiej jest bowiem wczytać wszystkie reguły dekoracji znaczników w jednym miejscu niż sprawdzać każdy znacznik z osobna [14].

Kolejną ważną optymalizacją polega na redukcji ilości żądań jakie wykonuje przeglądarka. Aby to osiągnąć należy wykorzystać narzędzia służące do












kompresji kodu wynikowego *JavaScript* oraz *CSS*. Jeśli złączymy wszystkie arkusze styli w jeden plik, podobnie jak kod *JavaScript*, ilość żądań ulegnie zmniejszeniu, przyspieszając jednocześnie czas wczytywania strony. Zgodnie z [14], dobrze jest wykorzystać osobne serwery do składowania plików statycznych. Pojęcie to znane jako CDN (ang. Content Delivery Network) jest szczególnie rozpowszechnione w wypadku popularnych bibliotek *JavaScript*. Kod frameworka *jQuery* umieszczony jest właśnie na jednym z takich serwerów (*google apis*), podobnie jak biblioteka *jQuery Validate*, dokonująca weryfikacji danych formularza.

Jednym z przydatnych dodatków do frameworka *Symfony*, jest wtyczka *sfCombine*, oferująca duże możliwości w zakresie optymalizacji zasobów. Za jej pomocą możliwe jest skompresowanie i złączenie wszystkich zasobów *JavaScript* w jeden plik, podobnie jak arkuszy *CSS*. Dodatkowo wtyczka kontroluje nagłówki *HTTP* wysyłane do przeglądarki. W ten sposób przeglądarka pobiera nową wersję zasobów, tylko w sytuacji, kiedy te uległy zmianie, w przeciwnym wypadku korzysta z lokalnej wersji. Myślę, że nie trzeba nadmieniać jak dużo można w ten sposób osiągnąć.










Na rysunku 4.2 widać czas oczekiwania oraz ilość żądań wysyłanych do serwera w celu pobrania strony. Są to statystyki bez włączonej wtyczki kompresującej. Oczywiście w wypadku naszej aplikacji mamy tylko trzy arkusze styli i jeden plik *Javascript* nie pochodzący z *CDN* i przechowywany lokalnie. Korzyści przy włączeni wtyczki nie powinny być, więc tak znaczne, jak w przypadku aplikacji złożonej z kilkunastu plików z zasobami. Dane zawarte na rysunku 4.3, pokazują jednak, że nawet w wypadku prostej aplikacji, warto kompresować zawartość. Przewidywana redukcja żądań do 9 nie może, aż jest tak imponująca, jak fakt, że łączny czas oczekiwania udało się zredukować o 50%, a transfer danych zmniejszył się 50 razy. Jak widać, wszystkie zasoby zwracają status *304* określający, że zasoby nie zostały zmodyfikowane, dzięki czemu serwer korzysta z lokalnej wersji.

Optymalizacje po stronie klienta są bardzo ważne, ponieważ bezpośrednio wpływają na czas po jakim klient zobaczy stronę. Każdy kilobajt danych

Rozdział 4. Optymalizacja kodu klienta

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
 effective-java /page	GET	200 OK	text/html	Other	3.26KB 7.49KB	131ms 125ms
 bootstrap.min.css /css	GET	200 OK	text/css	effective-java:10 Parser	80.23KB 80.02KB	27ms 1ms
 bootstrap-responsive.min.css /css	GET	200 OK	text/css	effective-java:11 Parser	10.48KB 10.27KB	37ms 2ms
 main.css /css	GET	200 OK	text/css	effective-java:12 Parser	245B 31B	38ms 10ms
 jquery.min.js ajax.googleapis.com/ajax/libs/jquery/1.7.2	GET	200 OK	text/jav...	effective-java:137 Parser	33.27KB 92.62KB	229ms 143ms
 bootstrap.js /js	GET	200 OK	applica...	effective-java:137 Parser	49.14KB 48.92KB	28ms 5ms
 jquery.validate.js ajax.aspnetcdn.com/ajax/jquery.validate/1.9	GET	200 OK	applica...	effective-java:137 Parser	13.05KB 37.37KB	1.01s 701ms
 jquery.tmpl.min.js ajax.microsoft.com/ajax/jquery.templates/beta1	GET	200 OK	applica...	effective-java:137 Parser	3.82KB 5.87KB	679ms 458ms
 glyphicons-halflings.png /img	GET	200 OK	image/...	effective-java:137 Parser	13.71KB 13.50KB	71ms 14ms
 tworzenieweb2.appspot.com tworzenieweb2.appspot.com/api/comments/6	GET	200 OK	applica...	jquery.min.js:4 Script	924B 2.80KB	155ms 155ms
 tworzenieweb2.appspot.com tworzenieweb2.appspot.com/api/categories	GET	200 OK	applica...	jquery.min.js:4 Script	560B 327B	157ms 153ms
11 requests 208.66KB transferred 1.39s (onload: 1.23s, DOMContentLoaded: 1.23s)						

Rys. 4.2: Analiza ilości żądań aplikacji księgarni elektronicznej bez włączonych optymalizacji

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
 effective-java /page	GET	200 OK	text/html	Other	3.25KB 7.36KB	59ms 59ms
 1338332111.css /css-min/key/9753fabdf4faf	GET	304 Not Modi	text/css	<u>effective-java:10</u> Parser	303B 90.40KB	89ms 89ms
 jquery.min.js ajax.googleapis.com/ajax/lit	GET	304 Not Modi	text/ja...	<u>effective-java:13</u> Parser	136B 92.62KB	76ms 76ms
 jquery.validate.js ajax.aspnetcdn.com/ajax/jq	GET	304 Not Modi	applic...	<u>effective-java:13</u> Parser	182B 37.37KB	377ms 377ms
 jquery.tmpl.min.js ajax.microsoft.com/ajax/jqu	GET	304 Not Modi	applic...	<u>effective-java:13</u> Parser	182B 5.87KB	390ms 384ms
 1335417363.js /js-min/key/d56800ed0ee11	GET	304 Not Modi	applic...	<u>effective-java:13</u> Parser	303B 26.37KB	147ms 92ms
 glyphicons-halflings.png /img	GET	304 Not Modi	image/...	<u>effective-java:13</u> Parser	158B 13.50KB	2ms 2ms
 tworzenieweb2.appspot.c tworzenieweb2.appspot.cor	GET	304 Not Modi	applic...	<u>jquery.min.js:4</u> Script	91B 2.80KB	87ms 87ms
 tworzenieweb2.appspot.c tworzenieweb2.appspot.cor	GET	304 Not Modi	applic...	<u>jquery.min.js:4</u> Script	91B 327B	106ms 106ms
9 requests 4.66KB transferred 605ms (onload: 501ms, DOMContentLoaded: 500ms)						

Rys. 4.3: Analiza żądań aplikacji księgarni elektronicznej z włączoną optymalizacją

więcej do pobrania, odpowiednio wydłuża czas wczytywania strony, dlatego zastosowano kompresję zasobów oraz ich złączenie. Dodatkowo dzięki wykorzystaniu nagłówków *HTTP*, możliwe jest ograniczenie danych, które pobiera przeglądarka w chwili pobierania strony.

Rozdział 5

Optymalizacja aplikacji

W ramach poprzednich rozdziałów prześledzono szczegółowo wszystkie istotne kwestie związane z optymalizacją aplikacji webowych. Czas więc rozpocząć praktyczną implementację oprogramowania.

5.1 Framework Django

Framework *Django* powstał na bazie języka Python i w niedługim czasie zrewolucjonizował proces tworzenia aplikacji internetowych w tymże języku.

Django posiada wszystkie cechy, które charakteryzują narzędzia do szybkiego tworzenia oprogramowania (*ang. Rapid Application Development*). Podobnie, jak *Ruby on Rails* lub *Symfony*, posiada bibliotekę do mapowania obiektowo - relacyjnego. W ten sposób definiując obiektywne modele i ich metody, można bez wiedzy z dziedziny baz danych wykonywać na nich operacje. Oczywiście w niniejszej pracy, istotna jest kompleksowa znajomość zagadnień bazodanowych, dlatego też z jednej strony wykorzystano to narzędzie w celu przyspieszenia procesu implementacji tworzonej aplikacji, a z drugiej strony trzeba mieć na uwadze wszystkie omówione w poprzednich rozdziałach optymalizacje po stronie silnika bazodanowego [1].

Frameworki pokroju *Ruby on Rails* posiadają atut w postaci *scaffoldingu*,

czyli zdolności do generowania podstaw aplikacji przy użyciu odpowiednich narzędzi linii komend. *Django* nie jest pod tym względem wyjątkiem, ponieważ umożliwia generowanie zarówno bazy aplikacji, jak również kompletnego panelu administracyjnego, oferującego zaawansowane funkcje, przydatne na przykład do redagowania strony przez użytkownika końcowego bez znajomości technologii.

W przypadku projektowanej księgarni, za pomocą generatorów *Django* można w prosty sposób zarządzać aplikacją w oparciu o wcześniej zdefiniowane modele. Oczywiście omówione w poprzednim rozdziale rozszerzenie *Django-Norel* umożliwią rozszerzenie zakresu dostępnych baz danych o bazy nierelacyjne.

Django-Norel zapewnia również narzędzia, umożliwiające łatwe wdrożenie tworzonego oprogramowania na platformę *Google Application Engine*. Aby jednak możliwe było umieszczenie projektu na tej platformie należy zarejestrować darmowe konto. Po zweryfikowaniu jego poprawności możliwe jest utworzenie do 10 darmowych aplikacji, każdej z własną bazą *Google Datastore* i zbiorem powiązanych usług [2].

5.2 Wystawienie usług jako webserwisów

Jednym z powodów wyboru frameworka *Django* jest łatwa możliwość tworzenia *REST*owych usług w oparciu o istniejące modele. Zapewnia to wtyczka *Django-Piston*, która dokonuje *serializacji* do kilku popularnych formatów, w tym do notacji *JSON*. Listing 5.1 przedstawia implementację jednej z usług (Komentarzy). Jak widać implementacja takiej usługi jest dosyć prosta z wykorzystaniem wspomnianej wcześniej wtyczki. Wystarczy stworzyć nową klasę, która dziedziczy po bazowej *BaseHandler*. Następnie określono dozwolone metody HTTP, odpowiedzialne za pobieranie danych, ich tworzenie, aktualizację, a kończąc na usuwaniu [9].

Piston oferuje możliwość zabezpieczania dostępu do konkretnych zasobów. W ten sposób tylko osoba znająca hasło może dodać komentarz lub oce-

nę. Z drugiej strony możliwe jest stworzenie osobnej wersji usługi udostępniającej tylko określone operacje. Jak widać na listingu 5.1, odpowiedzialna jest za to klasa `AnonymousCommentHandler`. Wymieniona usługa pozwala na dodanie komentarza lub ich podgląd. Ostatnia czynność konieczna do wystawienia usługi, to jej zmapowanie na określony adres zasobu. Odpowiedzialny jest za to kod z listingu 5.2. Po wdrożeniu aplikacji możliwe jest sprawdzenie działania usługi na przykład pod adresem `http://tworzenieweb.appspot.com/api/categories/`. Podany adres zasobu zwraca listę wszystkich kategorii, łącznie z ilością przypisanych książek. Listing 5.3 przedstawia zwrócony kod w notacji *JSON*.

Listing 5.1: Implementacja usługi odpowiedzialnej za komentarze

```
1 class AnonymousCommentHandler(AnonymousBaseHandler):
2     model = Comment
3     fields = ('title', 'content', ('user', ('id', 'first_name', 'last_name', 'username')), 'date', 'grade')
4
5     def read(self, request, book_id):
6         return self.model.objects.filter(book = book_id)
7
8 class CommentHandler(BaseHandler):
9     anonymous = AnonymousCommentHandler
10    allowed_methods = ('GET', 'PUT', 'DELETE', 'POST')
11    model = Comment
12    fields = ('title', 'content', ('user', ('id', 'first_name', 'last_name', 'username')), 'date', 'grade')
13
14    def read(self, request, book_id):
15
16        self.anonymous.read(request, book_id)
17
18    @validate(CommentForm)
19    def create(self, request, book_id):
20        data = request.data
```

```
21
22     book = Book.objects.filter(id = book_id).exists()
23
24     if not book:
25         return rc.NOT_FOUND;
26
27     em = self.model(
28         title=data['title'],
29         content=data['content'],
30         grade = data['grade'],
31         book_id = book_id,
32         user_id = request.user.id
33     )
34     em.save()
35
36
37     return rc.CREATED
```

Listing 5.2: Wystawienie usługi komentarzy do publicznego dostępu

```
1 comment_handler = Resource(CommentHandler, authentication =
    auth)
2
3 urlpatterns = patterns('',
4     url(r'^comments/(?P<book_id>\d+)/', comment_handler, { '
        emitter_format': 'json' }),
5 )
```

Listing 5.3: Rezultat zwrócony przez usługę kategorii

```
1 [
2     {
3         "count": 0,
4         "id": 3003,
5         "name": "Bazy Danych",
6         "slug": "bazy-danych"
7     },
8     {
```

```
9         "count": 2,
10        "id": 4005,
11        "name": "Java",
12        "slug": "java"
13    },
14    {
15        "count": 1,
16        "id": 13004,
17        "name": "PHP",
18        "slug": "php"
19    }
20 ]
```

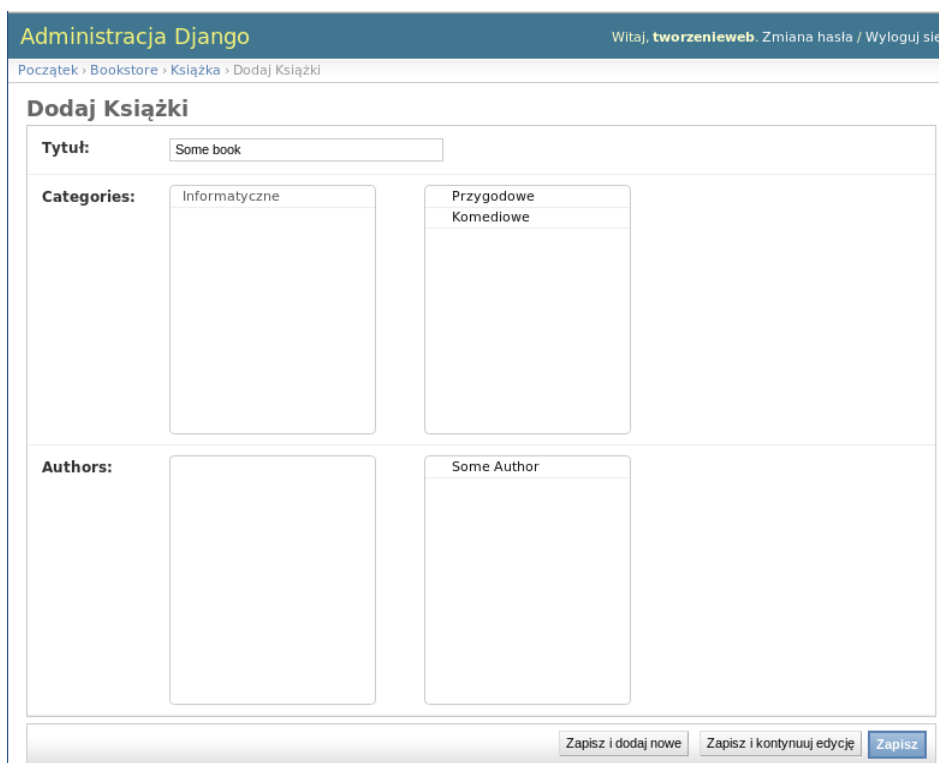
5.3 Uruchomienie panelu administracyjnego aplikacji

Zgodnie z omówionymi w podrozdziale 5.1 możliwościami frameworka *Django*, po utworzeniu odpowiednich modeli, możliwe jest dodanie opcji administracyjnych. Rysunek 5.1 pokazuje wygląd zaplecza administracyjnego. Natomiast rysunek 5.2 pokazuje przykładowy formularz dodawania książki. Ciekawym udogodnieniem jest wykorzystanie biblioteki *SelectMultiple* napisanej dla frameworka *jQuery*, w celu wygodniejszej pracy z polami list wielokrotnego wyboru. W ten sposób po lewej stronie umieszczone są kategorie, autorzy niewybrani, natomiast po prawej autorzy aktualnie zaznaczeni.

Jak zostało wykazane, framework *Django* w dużym stopniu przyspieszył proces implementacji docelowej aplikacji na platformie *Google Application Engine*. Jednym z ciekawych udogodnień oferowanych przez GAE jest system numeracji wersji. Przykładowo dodano do aplikacji nową funkcjonalność, która wymaga testów akceptacyjnych, edytując specjalny plik `app.yaml`. Parametr `version` pozwala na przykład na wdrożenie aplikacji pod specjalną subdomeną testową. Przykładowo `version: staging` spowoduje, że osobna wersja aplikacji dostępna będzie pod adresem `staging.tworzenieweb.appspot.com`.

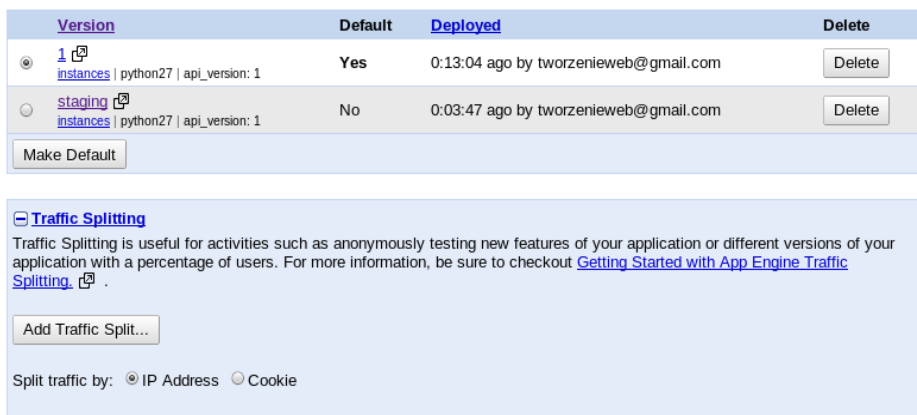


Rys. 5.1: Wygląd zaplecza administracyjnego stworzonej aplikacji



Rys. 5.2: Wygląd zaplecza administracyjnego stworzonej księgarni internetowej

Dodatkowo, w panelu administracyjnym możliwe jest ustawienie, która z wdrożonych aplikacji jest aplikacją domyślną (Rys. 5.3).



Rys. 5.3: Zarządzanie wersjami oprogramowania wdrożonego na platformę *Google Application Engine*

5.4 Framework Symfony

Front-end aplikacji został napisany przy wykorzystaniu frameworka Symfony w wersji 1.4. Powodem wyboru tej technologii jest możliwość szybkiego tworzenia oprogramowania również z wykorzystaniem generatorów i założeń *scaffoldingu*. Obecnie bardzo często daje się zaobserwować łączenie technologii w sposób podobny do przedstawionego, czyli przykładowo zaplecze aplikacji i interfejs bazodanowy tworzony jest w jednej technologii na przykład *Django* albo *Ruby on Rails*, natomiast prezentacja treści jest wykonana przy użyciu języka *PHP*. Dzieje się tak, ponieważ *PHP* jest doskonale dostosowane do tworzenia dokumentów końcowych w języku *HTML*.

Dla przyspieszenia tworzenia części przeglądowej aplikacji, utworzono wykorzystując *Doctrine ORM*, schemat bazy danych tożsamy z tym który znajduje się w *back-endzie*. Tak naprawdę jest on potrzebny wyłącznie do wygenerowania formularzy danych, ponieważ jak zostało wspomniane, operacje bazodanowe są wykonywane wykorzystując webserwisy.

Istotnym udogodnieniem w interfejsie użytkownika jest wykorzystanie wtyczki *jQuery Validation*. Wykorzystując tę bibliotekę, możliwe jest wykonanie sprawdzania poprawności danych, wprowadzonych do poszczególnych pól. Całość jest zintegrowana z klasami formularzy *Symfony*, więc w momencie ustalenia przez aplikację rodzaju formularza, tworzony jest automatyczny plik reguł walidacji (Listing 5.4), który jest następnie dołączany do źródła strony. Jeśli kod *JavaScript* jest włączony w przeglądarce, to skrypt nie pozwoli na wysłanie formularza i będzie pokazywał odpowiednie błędy na stronie.

Listing 5.4: Przykładowy zbiór reguł sprawdzania danych formularzy po stronie klienta

```
1 processForm = function(form) {
2     form.submit();
3 }
4
5 jQuery(function($){
6
7     $('#id').parents('form').validate({
8         rules: {"first_name":{"maxlength":255},"last_name":{"
9             maxlength":255},"email_address":{"required":true,"
10             email":true},"username":{"required":true,"maxlength"
11             :128},"password":{"required":true,"maxlength":128},"
12             password_again":{"maxlength":128}},
13         messages: {"first_name":{"maxlength":function(a, elem){
14             return '\\\\' + $(elem).val() + '\\\\' is too long
15             (255 characters max).';}}, "last_name":{"maxlength":
16             function(a, elem){ return '\\\\' + $(elem).val() + '
17             \\\\' is too long (255 characters max).';}}, "
18             email_address":{"required":"Required.", "email":"You
19             should provide valid email address"},"username":{"
20             required":"Required.", "maxlength":function(a, elem){
21                 return '\\\\' + $(elem).val() + '\\\\' is too long
22                 (128 characters max).';}}, "password":{"required":"
23                 Required.", "maxlength":function(a, elem){ return '
24                 \\\\' + $(elem).val() + '\\\\' is too long (128
```

```
        characters max).';}}, "password_again": {"maxlength":
        function(a, elem){ return '\\\\"' + $(elem).val() + '
        \\\\" is too long (128 characters max).';}}},
10    onkeyup: false,
11    wrapper: 'span class="help-inline"',
12    errorElement: 'span',
13    errorPlacement: function(error, element)
14    {
15
16        element.parent().parent().addClass('error');
17
18        if(element.parents('.radio_list').is('*') || element.
            parents('.checkbox_list').is('*'))
19        {
20            error.appendTo( element.parent().parent().parent() )
                ;
21        }
22        else
23        {
24            error.appendTo( element.parent() );
25        }
26
27    },
28    submitHandler: function(form) {
29
30        processForm(form);
31
32    }
33
34    });
35
36    $('#password_again').rules('add', {"equalTo": "#password",
        "messages": {"equalTo": "The two passwords must be the
        same."}});
37
38    });
39
40    /* for some reason the jQuery Validate plugin does not
```

```
    include a generic regex method */
41 jQuery.validator.addMethod(
42     "regex",
43     function(value, element, regexp) {
44         if (regexp.constructor != RegExp)
45             regexp = new RegExp(regexp);
46         else if (regexp.global)
47             regexp.lastIndex = 0;
48         return this.optional(element) || regexp.test(value);
49     },
50     "Invalid."
51 );
```

Rysunek 5.4 pokazuje rezultat działania walidacji formularza. Dopiero kiedy wszystkie błędy zostaną poprawione, skrypt pozwoli na wysłanie żądania typu POST do docelowej usługi. Takie podejście w dużym stopniu ogranicza ruch sieciowy w aplikacji. Oczywiście oprócz walidacji danych po stronie klienta, należy również bezwzględnie sprawdzać dane po stronie serwera, ponieważ może to prowadzić do znacznych uchybień w bezpieczeństwie aplikacji.

Innym ważnym udogodnieniem jest modyfikacja istniejącej wtyczki, odpowiedzialnej za ochronę dostępu do aplikacji. Wtyczka *sfDoctrineGuardPlugin* z założenia działała z bazą danych, więc w punktach odpowiedzialnych za logowanie do aplikacji, pobieranie obiektu użytkownika lub tworzenie nowego profilu, należało zmodyfikować kod. W ten sposób zamiast wykonywać zapytania do bazy, serwer lub obiekt *XMLHttpRequest* wysyła żądanie do aplikacji na platformie GAE i przetwarza zwrócony rezultat. W wypadku uwierzytelnienia konieczne jest zapisanie danych użytkownika do sesji, więc od tej pory aplikacja zapamięta skojarzony obiekt użytkownika.

Register to have full access to application

First name

Last name

Email address
 Required.

Username
 Required.

Password
 Required.

Password again
 The two passwords must be the same.

Rys. 5.4: Wynik działania wtyczki *jQuery validation*

5.5 Test wydajności *webserwisów*

W celu sprawdzenia faktycznej wydajności usług, wykonano kilka testów sprawdzających zarówno działanie operacji odczytu, jak również zapisu. Jak widać na listingu 5.5, test przeszedł bardzo sprawnie, osiągając liczbę prawie 60 żądań na sekundę. Test był przeprowadzany dla **100** użytkowników jednocześnie. Następnie została sprawdzona usługa komentarzy (Listing 5.6). W wypadku komentarzy test wypadł trochę gorzej, ale związane jest to z większym skomplikowaniem tej usługi i większą ilością danych do pobrania. Wynik **37.6** żądania na sekundę jest bardzo wysoką wartością i z pewnością gwarantuje szybkie działanie aplikacji.

Dodatkowo, w celu przyspieszenia operacji odczytu w encjach *kategorii* oraz *książek* przechowywane są informacje o średniej ocen pochodzące z komentarzy, jak również ilość książek przechowywana w danej kategorii. Oznacza to pewną nadmiarowość, ale jest ona konieczna w tego typu rozwiązaniach i powszechnie stosowana.

Jak widać na listingu 5.7, obciążenie *front-endu* testowane na lokalnym komputerze równie skutecznie radzi sobie z obsługiem skokowym 100 użytkowników. W przyszłości można przenieść kod *front-endu* również na jedno z rozwiązań opartych na chmurze dla języka *PHP*. Oferuje to na przykład hosting *pagoda box* (<https://pagodabox.com/>). Na lokalnym środowisku testowym wykorzystano serwer *Ngnix*, który cechuje się dużo mniejszym narzutem pamięci niż na przykład serwer *Http Apache*.

Listing 5.5: Test wydajności usługi pobierania kategorii w ramach stworzonej aplikacji

```

1 Server Software:      Google
2 Server Hostname:      tworzenieweb2.appspot.com
3
4 Document Path:        /api/categories
5 Document Length:      0 bytes
6
7 Concurrency Level:    100
8 Time taken for tests:  1.721 seconds
9 Complete requests:    100
10 Failed requests:      0
11 Write errors:         0
12 Requests per second:  58.11 [#/sec] (mean)
13 Time per request:     1720.744 [ms] (mean)
14 Time per request:     17.207 [ms] (mean, across all
    concurrent requests)
15 Transfer rate:        12.20 [Kbytes/sec] received
16
17 Connection Times (ms)
18          min    mean[+/-sd] median    max
```

```

19 Connect:      86   174   48.5    175    256
20 Processing:   294   765  297.3    796   1466
21 Waiting:     293   764  297.2    794   1465
22 Total:       379   939  328.0    931   1718

```

Listing 5.6: Test wydajności usługi komentarzy

```

1 Server Software:      Google
2 Server Hostname:      tworzenieweb2.appspot.com
3
4 Document Path:        /api/comments/6/
5
6
7 Concurrency Level:    100
8 Time taken for tests:  7.879 seconds
9 Complete requests:    100
10 Failed requests:      0
11 Write errors:         0
12 Total transferred:    243500 bytes
13 HTML transferred:     210200 bytes
14 Requests per second:  37.59 [#/sec] (mean)
15 Time per request:     27879.412 [ms] (mean)
16 Time per request:     278.794 [ms] (mean, across all
    concurrent requests)
17 Transfer rate:        8.53 [Kbytes/sec] received
18
19 Connection Times (ms)
20      min    mean[+/-sd] median    max
21 Connect:    64  121  60      68    270
22 Processing: 270   761  332.0    723   1647
23 Waiting:    268   760  332.0    722   1646
24 Total:      339  1014  3563.3    847  16173

```

Listing 5.7: Test *front-endu* aplikacji na lokalnym komputerze

```

1 Server Software:      nginx/1.1.19
2 Server Hostname:      thesis.web.dev
3 Server Port:          80

```

```
4
5 Document Path:      /
6 Document Length:    6621 bytes
7
8 Concurrency Level:   100
9 Time taken for tests: 2.747 seconds
10 Complete requests:  100
11 Failed requests:    0
12 Write errors:       0
13 Total transferred:  694500 bytes
14 HTML transferred:   662100 bytes
15 Requests per second: 36.40 [#/sec] (mean)
16 Time per request:    2747.185 [ms] (mean)
17 Time per request:    27.472 [ms] (mean, across all
    concurrent requests)
18 Transfer rate:       246.88 [Kbytes/sec] received
19
20 Connection Times (ms)
21      min    mean[+/-sd] median    max
22 Connect:      2      3    0.8      3      5
23 Processing:   77 1496 801.5   1509   2742
24 Waiting:      77 1496 801.5   1509   2742
25 Total:        82 1499 801.7   1512   2746
```

Potencjał aplikacji wynikający z uzyskanych wyników, pokazuje, że rozproszenie usług ma głęboki sens i przynosi wymierne rezultaty. Celem pracy nie było pokazanie wyłącznie korzyści związanych z wydajnością aplikacji. Dzięki dekompozycji aplikacji na warstwę usługową (webserwisy) i warstwę prezentacyjną zyskujemy dużą elastyczność. W każdej chwili można podmienić platformę *GAE* na zupełnie inne rozwiązanie, mając na uwadze zachowanie struktury obiektów zwracanych przez webserwisy. Nie wpłynie to w żadnym stopniu na działanie części prezentacyjnej. Ważne jest jednak by obiekty *JSON* pasowały do wcześniej ustalonej struktury.

Oprócz optymalizacji *back-endu* aplikacji, nie bez znaczenia jest również praca wykonana po stronie *front-endu*. Implementacja początkowej walidacji danych z wykorzystaniem *JavaScriptu*, pozwoli na szybszą detekcję błędów w

formularzach, co z pewnością docenią użytkownicy. Z drugiej strony moment wysłania żądania do serwera zostaje opóźniony do momentu kiedy jest już duża pewność prawidłowości danych.

Asynchroniczne pobieranie komentarzy to kolejny krok poprawiający jakość stworzonej aplikacji. W podejściu synchronicznym, na początku pobrano by z bazy dane o książce, następnie dane o komentarzach powiązanych ze stroną. Dzięki technologii *AJAX*, te dwa zadania mogą być wykonywane jednocześnie. Serwer WWW generuje tylko absolutną podstawę strony, pozostałe komponenty są dołączane w międzyczasie. Przekładając omówione zagadnienie na bardziej życiowe porównanie - o ile szybciej możliwe jest wybudowanie domu, jeśli jednocześnie kopane są fundamenty, stawiane mury domu i kładziony dach.

Podsumowanie

Właściwe zoptymalizowanie wydajności aplikacji internetowych jest jednym z ważniejszych czynników wpływających na ich przyszły sukces. Odpowiednia analiza i projekt aplikacji, idący w parze z dobrym zapleczem technicznym - sprzętowym powinny być stawiane jako nadrzędny cel przy realizacji projektów informatycznych w sieci. Faktyczne obsłużenie wielu użytkowników w niewielkich odstępach czasowych, zwiększenie popularności i dostępności, przy redukcji kosztów utrzymania to jedne z głównych korzyści, jakie można uzyskać poprzez rozsądne skalowanie wydajności aplikacji internetowych.

Optymalizacja wydajności serwisów internetowych ma istotne znaczenie w ujęciu globalnym, ponieważ wpływa ona na obciążenie całego Internetu. Analiza przeprowadzana w ramach poprzednich rozdziałów pokazała, że na odpowiednią wydajność aplikacji wpływa wiele czynników i żaden z nich nie powinien zostać zlekceważony. Dobór odpowiedniej platformy dostosowanej do potrzeb, to jedno z pierwszych zadań podczas projektowania aplikacji. Rozwiązania, takie jak *Google Application Engine* pokazują, że wydajność zależy w dużej mierze od sposobu składowania danych. Rezygnacja z rozwiązań charakterystycznych dla baz relacyjnych przynosi dużo lepszą skalowalność i wydajność aplikacji. Dzięki możliwości kontroli wydajności aplikacji za pomocą ilości pracujących jednocześnie instancji serwerów, można dostosować aplikację do bieżących potrzeb.

Optymalizacja to również wybór odpowiedniego języka programowania,

najlepszych wzorców projektowych i rozwiązań. Niekiedy wiąże się to ze zwiększeniem skomplikowania aplikacji. Należy mieć jednak na uwadze, że większość obecnych dużych serwisów internetowych, musiała przejść ogromny refaktoring, by móc sprostać rosnącym wymaganiom użytkowników i poziomowi ruchu sieciowego.

Optymalizacja *front-endu* pokazuje, że buforowanie jak największej ilości dostępnych zasobów, a także dobór odpowiedniego nośnika prezentowanych informacji wymiennie wpływa na czas "ładowania" strony. Dzięki rozwojowi języka *HTML5*, możliwe jest osiągnięcie wielu efektów graficznych, które dotychczas były możliwe do uzyskania wyłącznie przy pomocy edytorów graficznych i ciężkich grafik, koniecznych do pobrania.

Podczas optymalizacji należy jednak mieć na uwadze, by nie starać się na początku przyspieszać wszystkiego, ponieważ może to wyrządzić więcej problemów niż korzyści. Dobrze jest monitorować na bieżąco wydajność aplikacji, przez co możliwe jest znalezienie punktów newralgicznych systemu i właśnie wtedy zastosowanie optymalizacji. Należy więc zapamiętać słowa Donalda Knutha mówiące, że "niedojrzała optymalizacja jest źródłem wszelkiego zła".

Zaimplementowane w ramach pracy rozwiązanie pokazuje, że łącząc kilka dostępnych rozwiązań, jak na przykład framework *Django* z frameworkiem *symfony*, można osiągnąć naprawdę zadowalające efekty. Dzięki wyspecjalizowanym webserwisom możliwe jest ich wykorzystanie w więcej niż jednej aplikacji, natomiast technologia AJAX przybliża aplikacje webowe jeszcze bardziej do aplikacji desktopowych. Użytkownicy mają więc do dyspozycji coraz lepsze narzędzia, dostępne wprost z interfejsu przeglądarki. Sukces takich narzędzi, jak *Google Docs* nie byłby możliwy bez rozwoju technologii *JavaScript*. Dlatego coraz częściej warto odchodzić od synchronicznych stron internetowych, opartych wyłącznie na biernych relacjach klient-serwer.

Wykorzystane optymalizacje

Wyodrębnienie usług

Mianem usługi określa się tu każdy element oprogramowania, mogący działać niezależnie od innych oraz posiadający zdefiniowany interfejs, za pomocą którego udostępnia realizowane funkcje. Sposób działania każdej usługi jest w całości zdefiniowany przez interfejs ukrywający szczegóły implementacyjne – niewidoczne i nieistotne z punktu widzenia klientów. Dodatkowo, istnieje wspólne, dostępne dla wszystkich usług medium komunikacyjne, umożliwiające swobodny przepływ danych pomiędzy elementami platformy.

Takie podejście zapewniło łatwy dostęp i testowanie usług. Z drugiej strony jest to potężne narzędzie ponieważ usługi mogą ze sobą wchodzić w interakcje tworząc określone przypadki użycia.

Wykorzystanie technologii *PaaS*

Jak można przeczytać *PaaS* jest szczególnie polecany dla programistów, którzy chcą się po prostu skupić na wykonaniu własnego zadania, czyli napisaniu aplikacji. Programista nie musi zajmować się rzeczami takimi jak konfiguracja serwera, replikacja bazy danych czy problem wirtualizacji serwerów. Wszystko to zamknięte jest w prostym interfejsie udostępnianym przez *Google Application Engine*. Użyta platforma zapewnia skalowalność i olbrzymie możliwości przetwarzania danych, przez co idealnie sprawdziła się w pracy.

Technologia *NoSQL*

W celu zapewnienia skalowalności operacji bazodanowych wykorzystano silnik *Google Datastore*. W ten sposób aplikacja jest w stanie szybko replikować dane na więcej niż jedną instancję bazy, zmniejszając czas oczekiwania na dostęp do danych. W odróżnieniu od większości relacyjnych baz danych, czas potrzebny na przetwarzanie 1000 czy milionów rekordów jest w zasadzie stały.

JavaScript i AJAX

Redukując ilość operacji wykonywanych przez serwer WWW na rzecz prze-

głównie internetowej zyskano szybszy czas renderowania strony. Dodatkowo optymalizacja części *front-endowej* polegająca na przeniesieniu wszystkich definicji *JavaScript* zaraz przed zamykającym znacznikiem `</body>` wyraźnie poprawiła czas parsowania strony przez parser *DOM*.

Buforowanie zasobów

W celu ograniczenia ilości żądań wykonywanych przez serwer, wykorzystano również buforowanie, zapisujące tymczasowo wynik dla kategorii lub stron. Oczywiście taki bufor ma określoną "żywołność", w wypadku przekroczenia czasu życia zasobu, pobierana jest nowa wersja - w tym celu wykonywane jest już żądanie do webserwisu.

Perspektywy na przyszłość

Chociaż efekty osiągnięte dzięki optymalizacji są widoczne, jest to dopiero początek drogi w celu zapewnienia maksymalnej wydajności dla tworzonego oprogramowania. Jak donoszą statystyki architektura takich gigantów jak facebook zamyka się w około 180 tysięcy serwerów. Ilość ta jest przeznaczona do obsługi, szacowanej na 900 milionów bazy użytkowników. Dlatego też firmy tego formatu zawsze będą kilka kroków do przodu w porównaniu z jakimkolwiek rozwiązaniem na rynku.

Kolejnym krokiem w celu optymalizacji strony byłoby, więc dalsze rozproszenie usług na serwery brzegowe. Każdy z nich zlokalizowany byłby w różnych rejonach geograficznych. Część serwerów musiałaby istnieć w Ameryce Północnej i Południowej, część w Europie, Azji, tak by zniwelować dysproporcje odległości użytkowników od serwera.

Przydatnym rozwiązaniem byłoby również utworzenie lub skorzystanie z istniejących usług *Content Delivery Network*, dla przechowywania danych statycznych. W ten sposób serwer WWW nie poświęcałby pamięci, ani mocy obliczeniowej na pobieranie i wyświetlanie grafiki, plików *CSS* czy skryptów *JavaScript*.

W przyszłości, cenną optymalizacją byłoby również stworzenie oddziel-

nych usług działających w tle przeznaczonych do zliczania elementów w kategoriach, obliczania średnich dla komentarzy. Warto nadmienić, że Google stworzyło oprogramowanie przeznaczone do takich właśnie zastosowań. Mowa tutaj o rozwiązaniu *MapReduce* [4]. Całość opiera się na rozbiciu problemu na dwa kroki. Pierwszy krok "map" - główny program (*master node*) pobiera dane z wejścia i dzieli je na mniejsze podproblemy. Następnie każdy z nich jest przesyłany do robotników (ang. *worker nodes*). Każdy z nich może albo dokonać kolejnego podziału na podproblemy, albo przetworzyć problem i zwrócić odpowiedź do głównego programu. Krok "reduce" polega na pobraniu odpowiedzi na wszystkie pod-problemy i złączeniu w jeden wynik - odpowiedź na główny problem. Główną zaletą *MapReduce* jest umożliwienie łatwego rozproszenia operacji. Zakładając, że każda z operacji "map" jest niezależna od pozostałych, może być realizowana na osobnym serwerze.

Spis rysunków

1.1	Analiza czasu wykonywania strony <i>http://ftims.edu.p.lodz.pl//</i> wykonana w przeglądarce Google Chrome	13
1.2	Cykl życia żądania w typowej aplikacji webowej	15
1.3	Analiza ruchu sieciowego na stronie <i>http://ftims.p.lodz.pl</i> . . .	23
1.4	Schemat architektury MySQL	25
1.5	Schemat testowej bazy danych	26
2.1	Przypadki użycia stworzonej aplikacji	42
2.2	Diagram budowy aplikacji księgarni internetowej	46
2.3	Usługi udostępnione na platformie <i>GAE (Google Application Engine)</i>	49
2.4	Projekt bazy danych aplikacji księgarni internetowej	50
3.1	Zestawienie rodzajów chmury ze względu na udostępniane zasoby	52
3.2	Komponenty wchodzące w skład architektury w chmurze . . .	54
3.3	Wygląd panelu statystyk w ramach usługi <i>GAE</i>	56
3.4	Kontrola wydajności aplikacji w <i>GAE</i>	57
3.5	Nadzór i zarządzanie instancjami serwerów w <i>GAE</i>	58
4.1	Ekran przedstawiający interfejs użytkownika stworzonej księgarni internetowej	70
4.2	Analiza ilości żądań aplikacji księgarni elektronicznej bez włączonych optymalizacji	76

4.3	Analiza żądań aplikacji księgarni elektronicznej z włączoną optymalizacją	77
5.1	Wygląd zaplecza administracyjnego stworzonej aplikacji	84
5.2	Wygląd zaplecza administracyjnego stworzonej księgarni internetowej	84
5.3	Zarządzanie wersjami oprogramowania wdrożonego na platformę <i>Google Application Engine</i>	85
5.4	Wynik działania wtyczki <i>jQuery validation</i>	89

Spis listingów

1.1	Analiza strony z wykorzystaniem narzędzia ab	14
1.2	Test obciążenia czasowego	17
1.3	Zapytanie do wyświetlenia menu książki adresowej uczniów . .	26
1.4	Wynik zapytania z listingu 1.3	27
1.5	Utworzenie indeksu na polu lastname dla tabeli student . . .	28
1.6	Wynik zapytania z listingu 1.3 po optymalizacji indeksu . . .	29
1.7	Kilka możliwych do wykorzystania <i>złączeń</i> między tabelą profile a student	30
1.8	Bardziej rozbudowane zapytanie SQL	32
2.1	Przeciążenie metody zapisu dla klas reprezentujących encje book oraz comment	47
3.1	Realizacja <i>CORS</i> w praktyce	65
3.2	Kod JavaScript realizujący JSONP	65
3.3	Logika po stronie zewnętrznego serwera	65
4.1	Kod odpowiedzialny za realizację dodawania komentarzy . . .	71
4.2	Kod odpowiedzialny za wyświetlanie kategorii	73
5.1	Implementacja usługi odpowiedzialnej za komentarze	81
5.2	Wystawienie usługi komentarzy do publicznego dostępu	82
5.3	Rezultat zwrócony przez usługę kategorii	82
5.4	Przykładowy zbiór reguł sprawdzania danych formularzy po stronie klienta	86
5.5	Test wydajności usługi pobierania kategorii w ramach stwo- rzonej aplikacji	90

5.6	Test wydajności usługi komentarzy	91
5.7	Test <i>front-endu</i> aplikacji na lokalnym komputerze	91

Bibliografia

- [1] *Django 1.4 Dokumentacja*. <https://docs.djangoproject.com/en/dev/>.
- [2] *Django-norel Dokumentacja*. <http://docs.django-nonrel.org/en/latest/index.html>.
- [3] Historical trends in the usage of server-side programming languages for websites. Dostępny w internecie. http://w3techs.com/technologies/history_overview/programming_language.
- [4] *Introduction to Parallel Programming and MapReduce*. <http://code.google.com/intl/pl/edu/parallel/mapreduce-tutorial.html>.
- [5] *MySQL 5.1 Reference Manual*. http://dev.mysql.com/doc/refman/5.1/en/explain-output.html#explain_id.
- [6] N. Antonopoulos, L. Gillam. *Cloud Computing Principles, Systems and Applications*. Springer, 2010.
- [7] M. Lutz. *Learning Python*. O'Reily, 2009.
- [8] P. MacIntyrem, B. Danchilla, M. Gogal, T. Myer. *Pro PHP Programming*. Apress, 2011.
- [9] J. Noehr. *Piston Dokumentacja*. <https://bitbucket.org/jespern/django-piston/wiki/Documentation>.
- [10] A. Padilla, T. Hawkins. *Pro PHP Application Performance Tuning PHP Web Projects for Maximum Performance*. Apress, 2010.

- [11] G. Reese. *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud*. O'Reily, 2009.
- [12] D. Sanderson. *Programming Google App Engine*. O'Reilly, 2009.
- [13] B. Schwartz, P. Zaitsev, V. Tkachenko, J. D. Zawodny. *High performance MySQL, Second Edition*. O'Reily, 2008.
- [14] S. Souders. *High Performance Web Sites*. O'Reily, 2007.

Abstract

Nowadays Internet has been used as a wide range marketing and business tool. Most people use Internet in various fields of life. For this reason, sites and web applications becomes crowded and overloaded. Some times they can even stop working which will result in loss of money. That is why creating of high performance and stable applications is so important today. Also giving possibility to access page by huge amount of users can be cost-effective, when we put some commercials.

Last but not least web application architecture needs to be considered. It is very common that companies use expensive hosting solutions, that don't actually fit their needs. On the another hand, when periodically massive traffic comes, their servers cannot handle this. That is why cloud service providers are considered to be a good alternative in this situations because companies pay only for resources actually used and nothing more.

This master thesis will cover the topic of efficiency optimization of web applications. It will be splitted into three main areas of optimizations. Firstly the optimization of application architecture will be considered and then other issues like frontend optimization as well as database and webserver tuning will be discussed. Also, this thesis needs to provide some useful tools and techniques that need to be used for mesuring of actual application performance and availability. Without this knowledge we cannot even start the main part of optimization because we do not find the typical bottlenecks.

Płyta CD

Wraz z treścią pracy dyplomowej dołączono również płytę CD z kompletnym kodem źródłowym aplikacji. Dodatkowo kod można pobrać z poniższego repozytorium GIT: <https://github.com/tworzenieweb/Master-Thesis.git>