POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT ELEKTROTECHNIKI TEORETYCZNEJ
I SYSTEMÓW INFORMACYJNO-POMIAROWYCH

PRACA DYPLOMOWA MAGISTERSKA
na kierunku INFORMATYKA

Marcin Jasion
Nr ind. 230338

Rok akad.: 2015/2016
Warszawa, 1 października 2015

Porównanie wydajności serwisów RESTful w językach Java i Go

Zakres pracy:

1. Przegląd istniejących metod tworzenia aplikacji internetowych
2. Projekt i implementacja systemu testowego
3. Opis testów wydajnościowych
4. Analiza wyników

(Podpis i pieczątka Kierownika Zakładu Dydaktycznego)

Kierujący pracą: prof. nzw. dr hab. inż. Krzysztof Siwek

Termin wykonania: 16 marca 2016

Streszczenie pracy magisterskiej

Porównanie wydajności serwisów RESTful w językach Java i Go

Niniejsza praca przedstawia porównanie wydajności serwisów RESTful w dwóch różnych językach programowania.

Na początku autor zajmuje się omówieniem istniejących metod tworzenia aplikacji internetowych. Opisuje architekturę REST i sposób jej wykorzystania do budowania systemów informatycznych z zastosowaniem mikroserwisów. Omawia języki Java i Go, które wykorzystuje w niniejszej pracy. Opisuje ich historię oraz różnice między nimi podczas tworzenia aplikacji internetowych.

W kolejnych rozdziałach autor przedstawia opis aplikacji stworzonych do przeprowadzenia testów. Prezentuje ich architekturę oraz sposób i przypadki użycia. Omawia też narzędzia wykorzystywane podczas przeprowadzania testów.

Główną część pracy stanowią testy wydajnościowe. Autor przedstawia opis poszczególnych testów z uwzględnieniem podziału na grupy oraz sposób ich przeprowadzenia. W dalszej części prezentuje wyniki testów w formie wykresów, przeprowadza ich wnikliwą analizę i porównuje wydajności testowanych aplikacji.

W ostatnim rozdziale autor dokonuje podsumowania całej pracy. Porównuje języki Java i Go pod względem wydajności oraz łatwości tworzenia w nich serwisów RESTful. Autor kończy pracę opinią o roli testów wydajnościowych podczas tworzenia systemów informatycznych.

The summary of master's thesis

Performance comparison of RESTful services in Java and Go languages

The present study compares the efficiency of RESTful services in two different programming languages.

At the beginning, the Author discusses the existing methods of creating online applications. He describes the REST architecture and its application to building information systems using microservices. He discusses the Java and Go languages used in this study. He describes their history and their differences in the process of creating online applications.

In the subsequent chapters, the Author describes the applications created for performing the tests. He presents their architecture, directions for use and use cases. He also discusses the tools used during the tests.

Performance tests are the main part of the study. The author describes each test, dividing them into groups, and the testing methods used. In further chapters, he presents the tests results as graphs, thoroughly analyses them and compares the efficiencies of the tested applications.

In the final chapter, the Author summarises the entire study. He compares the Java and Go languages in terms of the efficiency and ease of creating RESTful services with them. The Author concludes the study with his opinion on the role of performance tests in the process of creating information systems.

Warszawa, dnia 16 marca 2016r.

Politechnika Warszawska
Wydział Elektryczny

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa magisterska pt. Porównanie wydajności serwisów RESTful w językach Java i Go:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

.....

Spis treści

Składam serdeczne podziękowanie

Panu prof. nzw. dr hab. inż. K. Siwkowi
za umożliwienie wykonania pracy
oraz wydatną pomoc i opiekę
w czasie jej wykonania.

Marcin Jasion

Rozdział 1

Wstęp

W dobie coraz dynamicznej rozwijającej się sieci Internet serwisy internetowe stają się częścią naszego życia. Serwisy informacyjne, sklepy internetowe czy bankowość elektroniczna pomagają w codziennych czynnościach, zaoszczędzając w ten sposób nasz cenny czas i wysiłek. Jednak, aby serwisy internetowe spełniały swoje funkcje muszą mieć odpowiednią wydajność.

Dla użytkownika korzystającego z portali społecznościowych, system z którego korzysta wydaje się być prosty. W rzeczywistości jednak taki system jest bardzo skomplikowanym oprogramowaniem, które często działa na wielu serwerach.

W celu zapewnienia sprawnego funkcjonowania systemu, jego niezawodności oraz umożliwienia jego ciągłego rozwoju zostało wypracowanych wiele metod inżynierii oprogramowania. Dziedzina ta wytworzyła zasady postępowania podczas tworzenia i rozwijania oprogramowania, które obejmują sposoby gromadzenia wymagań o systemie, jego implementacji, testowania jak również wdrażania. Dobrze zaprojektowany i wykonany system może przynieść ogromne zyski czasowe i finansowe.

Wydajność jest jednym z aspektów powodzenia funkcjonowania systemu. Zaplanowanie i wykonanie odpowiednich testów wydajnościowych pozwala zbadać jak system zachowa się podczas dużego obciążenia, jak dużo użytkowników może równocześnie z niego korzystać oraz czy spełnia założone wymagania. Dzięki testom można też znaleźć i wyeliminować najsłabsze punkty systemu.

Celem niniejszej pracy jest porównanie wydajności serwisów *RESTful* w wybranych platformach oprogramowania. Na potrzeby badań stworzone zostaną aplikacje typu *RESTful* zaimplementowane w dwóch różnych językach programowania: *Java* i *Go*. Do analizy wydajności serwisów zostaną opracowane i wykonane testy wydajnościowe, na podstawie których zostanie przeprowadzona analiza.

Rozdział 2

Przegląd literatury

2.1 REST

REST (ang. *REpresentational State Transfer*) jest stylem architektonicznym oprogramowania dla rozproszonych systemów hipermedialnych. W 2000 roku opisał go Roy Thomas Fielding w swojej pracy doktorskiej zatytułowanej *"Architectural Styles and the Design of Network-based Software Architectures"* [1].

Roy Fielding badając szybki rozwój sieci *Internet* przedstawił różne style architektoniczne, które przyczyniły się do szybkiego rozwoju m.in. sieci *WWW* (ang. *World Wide Web*). Za źródło sukcesu uważał rozproszenie systemu oraz to, jak poszczególne zasoby sieci łączyły się ze sobą. Był on również współautorem protokołu *HTTP/1.0*. W swojej pracy zaproponował nowy styl architektury nadając mu nazwę *REpresentational State Transfer* skrót *REST*.

Styl *REST* jest zgodny z architekturą sieci *WWWW*. Opisuje on sieć jako rozproszony zbiór aplikacji wykorzystujących *hypermedia*, czyli dane będące powiązaniem grafiki, dźwięku, wideo, tekstu oraz *hyperlinków*.

2.1.1 Architektura REST

REST jest hybrydą kilku stylów architektonicznych. Przestrzeganie zasad, które *REST* wprowadza gwarantuje lepszą wydajność, skalowalność oraz niezawodność usług internetowych tworzonych w tej architekturze. Swoją architekturą łączy kilka wzorców tworzenia aplikacji sieciowych.

Pierwszym jest wzorzec *klient-serwer*. Rozdziela on rolę serwera i klienta. Rolą serwera jest zapewnienie danych, natomiast rolą klienta jest zgłaszanie się po dane. Klient ma możliwość tworzenia aplikacji wykorzystujących ten sam interfejs, które można uruchomić na różnych systemach. Serwerom daje to możliwość rozwijać swoje interfejsy niezależnie od klientów.

Kolejnym wzorcem jest bezstanowość (ang. *stateless*). Opiera się on na zasadzie, że każde żądanie klienta do serwera musi zawierać informacje potrzebne do zrozumienia tego żądania oraz wygenerowania odpowiedzi. W praktyce oznacza to, że serwer nie przechowuje danych szczególnych klienta w swoim kontekście. Wymusza to na kliencie trzymanie stanu sesji. Dzięki temu pamięć serwera nie jest ograniczana, a jednocześnie pozwala na zwiększenie liczby klientów, z którymi może się równocześnie komunikować. Dodatkowo zwiększa to możliwość skalowalności usługi. Zachowanie zasady bezstanowości może powodować, że czasem część danych jest przesyłana nadmiarowo.

Dla poprawy wydajności wykorzystywana jest pamięć podręczna (ang. *cache*). Klient może zaimplementować pamięć podręczną w taki sposób, by przechowywać wybrane zasoby w pamięci otrzymane z serwera. Wymaga to klasyfikacji zasobów na takie, które nie muszą być często odświeżane, dzięki czemu nie są generowane dwa takie same żądania. Dodatkowo serwer może poinformować, czy otrzymany zasób może być przechowywany w pamięci, czy też nie oraz na jak długo.

Kolejną zasadą, którą *REST* wykorzystuje jest architektura warstwowa. Dzięki temu podejściu możliwe jest zhierarchizowanie komponentów systemu, tak by nie musiały wiedzieć o systemie więcej, niż to do czego zostały stworzone. W ten sposób system staje się prostszy do zrozumienia, a poszczególne komponenty mogą być wykorzystywane wielokrotnie.

REST wymaga zachowania jednolitego interfejsu w komunikacji między komponentami systemu. Każdy zasób powinien być jednoznacznie identyfikowany przy pomocy *URI* (ang. *Uniform Resource Identifier*). W *REST* możliwe jest modyfikowanie zasobów przez ich reprezentacje - taki sam zasób może zostać zwrócony w różnych formach np. *XML*, *JSON* czy zwykłego tekstu. Zwrócona forma determinuje sposób przetwarzania danych przez klienta. Wykorzystanie *hypermediów* jako silnika stanów aplikacji (ang. *Hypermedia as the Engine of Application State*, w skrócie *HATEOAS*) pozwala klientowi

na przechodzenie między stanami zasobu. Informacje o zasobach i operacjach na nich przesyłane są bezpośrednio do klienta. Dzięki temu klient nie musi mieć wiedzy o wszystkich operacjach na zasobach, żeby z nich korzystać. Jednolity interfejs sprawia, że architektura systemu jest uproszona, a współpraca między komponentami bardziej przejrzysta.

2.1.2 Usługi REST i RESTful

Usługi *REST* nie są ściśle powiązane z jakimkolwiek protokołem transportowym, jednak najczęściej funkcjonują w oparciu o protokół *HTTP* (ang. *Hypertext Transfer Protocol*).

Usługi *REST*, jak i ich zasoby są dostępne przy użyciu unikalnego adresu *URL* (ang. *Uniform Resource Locator*), który jest szczególnym przypadkiem adresu *URI*. Poza identyfikacją zasobu, pozwala on na zidentyfikowanie sposobu dostępu do niego. Ogólny adres *URL* ma następujący format:
schemat://nazwa_użytkownika:hasło@host:port/ścieżka?zapytanie.

W usługach *REST* część zasobów, w zależności od potrzeb może przyjmować różną postać tzn. jeden zasób dostępny pod tym samym adresem *URL* może zostać zaprezentowany w formacie np. *XML*, *JSON* lub innym, w zależności od potrzeb i jego zastosowania.

Usługi nie przechowują stanu pomiędzy kolejnymi wywołaniami. Oznacza to, że wszystkie żądania są od siebie niezależne.

Szczególnym przypadkiem usług *REST* są usługi *RESTful*. Pozwalają one wykonywać operacje *CRUD* (ang. *Create, Read, Update, Delete*) na zasobach. Zastosowanie metod *HTTP* umożliwia odwoływanie się do tych samych zasobów (mających ten sam identyfikator) w różnym celu. *RESTful* bazuje na następujących metodach protokołu *HTTP*:

- *GET* - do pobierania zasobów,
- *POST* - do tworzenia zasobów,
- *PUT* - do aktualizowania zasobów,
- *DELETE* - do usuwania zasobów.

CRUD w programowaniu opisuje szczególną funkcję aplikacji, która pozwala

tworzyć, odczytywać, modyfikować oraz usuwać dane. Najczęściej aplikacje takie integrują się z bazą danych.

2.2 Mikroserwisy

Mikroserwisy są małymi, niezależnymi serwisami, które ze sobą współpracują [2]. Są one odpowiedzią na rosnącą popularność trendu w tworzeniu oprogramowania ukierunkowanego na domenę (ang. *Domain-Driven Design*, skrót *DDD*) zapewniającego ciągłość dostaw zmian (ang. *continous delivery*), tworzone przez małe, niezależne zespoły oraz pozwalające na ich łatwe skalowanie.

Tworzenie oprogramowania ukierunkowanego na domenę zyskało popularność po 2003 roku. Eric Evans w książce *Domain-Driven Design: Tackling Complexity in the Heart of Software* przedstawił sposób tworzenia takiego oprogramowania [3]. Należy podkreślić, że DDD nie jest metodyką tworzenia oprogramowania. Jest sposobem na zrozumienie działania rozległych systemów, co pozwala na ustalenie priorytetów podczas ich tworzenia. Autor przedstawił, jak ważna jest reprezentacja prawdziwego świata w tworzonym oprogramowaniu.

Od 2003 roku podejście do tworzenia oprogramowania przy użyciu mikroserwisów podlega nieustannemu rozwojowi koncepcyjnemu i technologicznemu. Idzie ono w parze z *DDD*. Ponieważ mikroserwisy są małe, ich liczba rośnie wraz ze zmianami dodawanymi do systemu. W przypadku dużych, monolitycznych systemów dodanie kolejnej zmiany może skutkować błędami w innej części systemu, których nie dało się przewidzieć. Mikroserwisy pozwalają być skupione tylko na zadaniu, do którego zostały stworzone. Komunikacja między mikroserwisami odbywa się przez sieć, więc bez problemu można je uruchomić na odizolowanym środowisku lub platformie, przez co mikroserwisy stają się niezależne.

Uruchamianie mikroserwisów na różnych systemach pozwala na stworzenie systemu bardziej odpornego na awarie. Jeśli jeden z mikroserwisów przestanie działać np. przez awarię sprzętu, spowoduje to wyłączenie pojedynczej funkcjonalności. Sam system będzie wciąż działał, a niedziałającą usługę szybko można przenieść i uruchomić w innym miejscu. Przy monolitycznym

systemie, by zabezpieczyć się przed awariami musimy uruchomić ten sam system na kilku maszynach, marnując przez to zasoby.

Mikroserwisy pozwalają na łatwe skalowanie. Wystarczy, że uruchomimy ten sam serwis w kilku miejscach, a wydajność danej części systemu powinna wzrosnąć. W monolitycznych systemach jesteśmy zmuszeni do skalowania całych systemów, również i funkcjonalności, których w systemie rzadziej się używa.

Niezależność mikroserwisów jest zaletą przy wyborze technologii. Można stworzyć te same mikroserwisy w różnych językach, o ile każdy będzie działać tak samo. Przydaje się to również, gdy chcemy poprawić wydajność danej usługi lub gdy technologia, której używamy nie jest przystosowana pod konkretne zadanie.

Kolejną zaletą tworzenia systemów opartych o mikroserwisy jest wygoda ich wdrażania na środowiska produkcyjne niezależnie od pozostałych części systemu. Jeśli dodamy nową zmianę w danym serwisie, nie ma potrzeby restartowania całego systemu. Również w przypadku błędu pozwala to na szybkie jego znalezienie, poprawienie i wdrożenie.

2.3 Java

2.3.1 Historia i ewolucja języka Java

Początki języka Java miały miejsce w 1991 roku. Wtedy to Patrick Naughton i James Gosling, inżynierowie firmy Sun, wpadli na pomysł zaprojektowania języka, który można uruchomić na urządzeniach takich jak dekodery telewizyjne [4]. Ponieważ urządzenia takie pochodziły od różnych producentów i każdy mógł stosować różne procesory, język nie mógł być oparty na jednej architekturze. W ten sposób zespół pracujący na rozwiązaniem wskrzesił model uruchamiania oprogramowania na maszynach wirtualnych, gdzie programy były kompilowane do kodu pośredniego. Początkowo język nazwano Oak. Ponieważ taki język już istniał, postanowiono zmienić nazwę na Java.

Do roku 1994 roku zespół projektowy bez skutku próbował wykorzystać język w urządzeniach kablowych. Wtedy postanowili wykorzystać język do stworzenia przeglądarki internetowej niezależnej od architektury. W 1995 za-

prezentowali efekt swojej pracy - przeglądarkę *HotJava*. Przez wbudowaną przenośność język został uznany, że jest wart rozwijania i w 1996 roku wydano wersję 1.0. Krótko po tym wyszła wersja 1.1 rozszerzona m.in o możliwość drukowania, wprowadzającą model zdarzeń dla programowania GUI oraz poprawiająca refleksję.

Język miał wciąż spore braki i w 1998 roku wydano wersję 1.2. Wersja ta była ogromną zmianą w porównaniu do poprzedniczek, dlatego zmieniła swoją marketingową nazwę na "Java 2 Standard Edition Software Development Kit Version 1.2". Wraz z nią pojawiło się wiele nowych elementów. Do najważniejszych z nich należą biblioteka Collection oraz Swing. Biblioteka Swing pozwalała na tworzenie aplikacji z graficznymi interfejsami użytkownika (w tym apletów w przeglądarce), której elementy w łatwy sposób skalowały się do ekranów urządzeń, na których była uruchamiana. Równolegle zostały opublikowane dwie wersje języka: Micro Edition przeznaczona do urządzeń przenośnych np. telefonów komórkowych oraz Enterprise Edition do pisania aplikacji serwerowych. Dwie kolejne wersje (1.3 i 1.4) wprowadzały ulepszenia w samym języku oraz rozszerzały standardową bibliotekę. Z biegiem czasu popularność aplikacji uruchamianych po stronie klienta gasła, jednak rosła popularność pisania aplikacji uruchamianych po stronie serwerów.

W 2004 pojawiła się nowa wersja języka oznaczona numerem 5.0. Wprowadziła ona typy sparametryzowane (ang. *generic types*), co rozszerzyło możliwości pisania oprogramowania bardziej odpornego na błędy programisty. Doszły również rzeczy zaczerpnięte z języka C#: pętla *for each* do poruszania się po tabliach i kolekcjach oraz *autoboxing* pozwalający konwertować typy proste na obiekty i odwrotnie. Dodane zmiany były zaimplementowane tak, by nie zmieniać nic w maszynie wirtualnej.

Wersja Java SE 6 nie wnosiła nowych funkcji, ale rozszerzała bibliotekę standardową oraz poprawiła wydajność i bezpieczeństwo.

Z powodu problemów finansowych firmy Sun, w 2010 koncern Oracle wykupił firmę i w ten sposób stał się właścicielem praw do języka Java. Mocno opóźniona kolejna wersja wyszła w 2011. Wprowadzała ona sporo zmian. Do najważniejszych z nich należało dodanie klas *Fork/Join* upraszających programowanie równoległe, pozwalając na tworzenie procesów oraz poprawę

wydajności na systemach z procesorami wielordzeniowymi. Kolejną zmianą była nowa biblioteka obsługi wejścia/wyjścia nazwana *NIO*. Java 7 wnosiła też mechanizm *invokedynamic*, który pozwalał na nowy sposób wywołania metod. Wymusiło to zmiany w *bytecode* języka. Zmiana ta stała się fundamentem do tego co przyniesie Java 8.

2.3.2 Java 8

Java 8 jest najnowszą, obecnie istniejącą wersją języka. Została opublikowana w 2014 roku i przyniosła długo oczekiwane zmiany.

Jedną z nich i najważniejszą są wyrażenia *Lambda*, które pozwalają na czyste i zwięzłe wyrażenie pojedynczej metody. Są one często wykorzystywane przy operowaniu na kolekcjach, między innymi do sortowania czy filtrowania kolekcji. W przykładzie 2.1 zaprezentowano sortowanie listy osób, alfabetycznie po imieniu, przy użyciu klas anonimowych. Przykład 2.2 prezentuje tą samą funkcjonalność z wykorzystaniem wyrażeń *Lambda*.

Przykład 2.1: Sortowanie kolekcji przy użyciu klas anonimowych

```
1 Collections.sort(persons, new Comparator<Person>(){
2   public int compare(Person p1, Person p2){
3     return p1.firstName.compareTo(p2.firstName);
4   }
5 });
```

Przykład 2.2: Sortowanie kolekcji przy użyciu wyrażeń *Lambda*

```
1 Collections.sort(persons, (p1, p2) ->
    p1.firstName.compareTo(p2.firstName));
```

Przy użyciu wyrażeń *Lambda* kod jest wyraźnie prostszy i krótszy. Wyrażenia te w *bytecode* używają mechanizmu *invokedynamic* wprowadzonego w Javie 7. Mechanizm ten pozwala na opóźnienie przetłumaczenia wyrażenia w *bytecode* do momentu jego wywołania. Wynikiem wyrażenia *Lambda* jest

metoda statyczna, która jest tworzona w czasie wykonania *bytecode*.

Kolejną zmianą jest możliwość tworzenia metod domyślnych (ang. *default methods*) i statycznych (ang. *static methods*). Pozwola to m.in. na dostarczenie implementacji metody bez potrzeby tworzenia klas abstrakcyjnych.

Dużą nowością są też nowe klasy do obsługi dat i godzin. Pozwalają one na operacje takie jak dodawanie i odejmowanie w sposób uporządkowany i bardziej naturalny do zrozumienia.

Mniej popularną nowością jest możliwość wywoływania kodu *JavaScript* na wirtualnej maszynie Java (ang. *Java Virtual Machine*, w skrócie *JVM*).

2.3.3 Spring

Spring jest szkieletem aplikacji (ang. *framework*) zapoczątkowanym przez Roda Johnsona. W 2001 roku w książce *"Expert One-on-One: J2EE Design and Development"* zaprezentował on bibliotekę, która pozwalała tworzyć oprogramowanie biznesowe składające się się prostych komponentów *JavaBean* [5]. Biblioteka ta upubliczniona została w 2002 roku [6]. Z czasem uzupełniana o nowe funkcje, stała się jedną z najpopularniejszych bibliotek do tworzenia aplikacji. Złożenie biblioteki z modułów pozwoliło na dobieranie ich do potrzeb architektury aplikacji. Do najważniejszych i najbardziej popularnych modułów należą:

- *IoC* - do wstrzykiwania zależności,
- *MVC* - do tworzenia aplikacji webowych,
- *Data* oraz *JDBC* - jako warstwa dostępu do baz danych,
- *Security* - autoryzacja i zabezpieczanie aplikacji.

Przez lata tworzenie aplikacji wykorzystujących *Spring* było skomplikowane. Wiązało się z tworzeniem wielu plików konfiguracyjnych oraz dogłębnego zapoznania się z dokumentacją. W 2013 roku wraz z wyjściem wersji 4 biblioteki powstał projekt *Spring Boot*.

Spring Boot jest biblioteką, która pozwala na tworzenie oprogramowania z zachowaniem zasady *convention over configuration*, której główną cechą jest dostarczenie domyślnej konfiguracji, pozwalającej na uruchomienie aplikacji

bez potrzeby zbędnego konfigurowania. W *Spring Boot* zasada ta stała się na tyle rozpowszechniona, że od rozpoczęcia tworzenia aplikacji do jej pierwszego uruchomienia wystarczy kilka minut. Dodanie kolejnych bibliotek również nie wymaga żmudnych konfiguracji. Tworzenie aplikacji, które można uruchomić w ciągu kilku minut od rozpoczęcia programowania było celem, któremu twórcy *Spring* musieli stawić czoła w kontekście mikroserwisów.

Spring Boot obecnie pozwala na użycie wszystkich modułów jakich *Spring* dostarcza (m.in. *MVC*, *Security*, *Data*) oraz dodatkowo dodaje kilka kolejnych, niedostępnych w innych bibliotekach (m.in. Jetty, Tomcat, metryki, shell).

Najnowsza wersja biblioteki *Spring* oznaczona numerem 4.2 została wydana w czerwcu 2015, a *Spring Boot* w wersji 1.3 w listopadzie 2015.

2.3.4 Serwery Tomcat 8 i Jetty 9

Serwery *Tomcat* i *Jetty* są serwerami *HTTP* napisanymi w języku *Java*. Pełnią one również funkcję kontenerów aplikacji. Na serwerach można uruchomić aplikacje, które implementują interfejs *javax.servlet.Servlet*, służący do obsługi żądań *HTTP* tworzonych w języku *Java*. Do bibliotek wykorzystujących ten interfejs należy *Java Server Pages* czy *Spring MVC*. Aplikacje, które implementują klasę *javax.servlet.Servlet* buduje się do pliku *WAR*, a następnie wgrywa na kontener, gdzie jest rozpakowywana i uruchamiana pod wybranym kontekstem. *Tomcat* i *Jetty* implementują najnowszą wersje *Servlet 3.1*, która wykorzystuje nieblokującą kolejkę wejść-wyjść oraz pozwala na zastosowanie technologii *WebSocket*. Ponieważ serwery są napisane w języku *Java*, są dostępne w formie bibliotek, które można dołączyć do aplikacji, zbudować do pliku *JAR* i uruchomić.

2.4 Język Go

Język *Go*, potocznie nazywany *golang*, powstał w 2007 roku w firmie Google. Autorami jego byli Ken Thompson, Robert Griesemer oraz Rob Pike [7]. Ponieważ każdy z nich miał swój wkład w rozwój języka *C*, chcieli stworzyć język programowania, który będzie w momencie tworzenia tym, czym *C* był

w latach 80. Autorzy języka chcieli, by nowy język pozwalał na tworzenie dużych systemów, które łatwo będą się skalować, kompilacja ich będzie bardzo krótka, będzie statycznie typowany, a zarządzanie pamięcią nie będzie zmorą programistów. Przykład 2.3 przedstawia program, który wypisuje tekst *Hello World*.

Przykład 2.3: Przykład programu w języku Go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello World")
7 }
```

Go był tworzony z myślą o programowaniu równoległym. Projektując, autorzy wzorowali się językami *Erlang* oraz *C*. W *Erlang* tworzenie programów wielowątkowych było proste i pozwalało na wykorzystanie wszystkich rdzeni systemu równomiernie. Wadą *Erlang* była wydajność programów na systemach jednowątkowych. Język *C* był w tym względzie znacznie wydajniejszy, natomiast tworzenie programów operujących na kilku wątkach wymaga zastosowania bibliotek takich jak *POSIX* oraz *OpenMP*. W języku *C* pisanie skalowalnych aplikacji wymagało dodatkowej pracy.

Go łączy najlepsze cechy z obu języków, a do tworzenia systemów wielowątkowych wykorzystuje *go rutyny* (ang. *gorutine*) [8]. Są to funkcje o równoległym wykonywaniu. Wywoływana funkcja musi być poprzedzona słowem kluczowym *go*. Z wykorzystaniem kanałów programista jest w stanie zaimplementować komunikację między uruchomionymi funkcjami. Jednak wywołanie funkcji w *goroutine* nie oznacza stworzenia nowego wątku.

W przykładzie 2.4 przedstawiony jest program uruchamiający funkcję wewnątrz *go rutyny*

Przykład 2.4: Przykładowy program w Go wykorzystujący goroutine

```
1  package main
2
3  import "fmt"
4
5  func say(s string) {
6      fmt.Println(s)
7  }
8
9  func main() {
10     go say("Hello World")
11 }
```

Kolejną rzeczą, którą autorzy języka postawili sobie na celu była jego szybka kompilacja. Programy w językach *C/C++* kompilowały się długo. Szybkość kompilacji *Go* pozwala nawet na tworzenie skryptów zastępujących języki skryptowe takie jak *Bash*, *Perl*, *Python*

Język *Go*, podobnie jak *Java*, jest językiem statycznie typowanym. To znaczy, że typy zmiennych są znane w momencie kompilacji. Dzięki temu można uniknąć wielu błędów już w momencie kompilacji. Jednak język ma też wbudowane mechanizmy programowania dynamicznego.

Dostępnie typy zmiennych są w większości podobne do tych, które występują w języku *C*. Występują tu również struktury i interfejsy. Jednak język pozwala na tworzenie metod (prywatnych i publicznych), które będą przypisane do struktury. Nie można powiedzieć, że język ten należy do grupy języków obiektowych ponieważ niemożliwe jest np. zaprogramowanie dziedziczenia. Kolejną różnicą w stosunku do języka *Java* jest brak typów generycznych, które tak bardzo pomogły w tworzonych systemach.

Składnia języka *Go* jest bardzo czytelna i łatwa do zrozumienia. Autorzy stworzyli zbiór standardów dotyczący m.in. wcięć czy wywoływania metod, które mają na celu uspójnienie pisanych programów. Kompilator języka ma wbudowany analizator, który w momencie odstępstw przerywa dalszą kompilację. Takie podejście pozwala, by programista mógł wdrożyć się w istniejący

system skupiając się na jego działaniu, a nie dodatkowo na stylu składni jaki panuje w zespole

Język *Go* jest coraz częściej wykorzystywany np. w systemach *Docker*, *Kubernetes*. Znalazł również zastosowanie w usługach sieciowych takich jak serwery DNS czy bazy danych.

Rozdział 3

Projekt aplikacji

3.1 Opis aplikacji

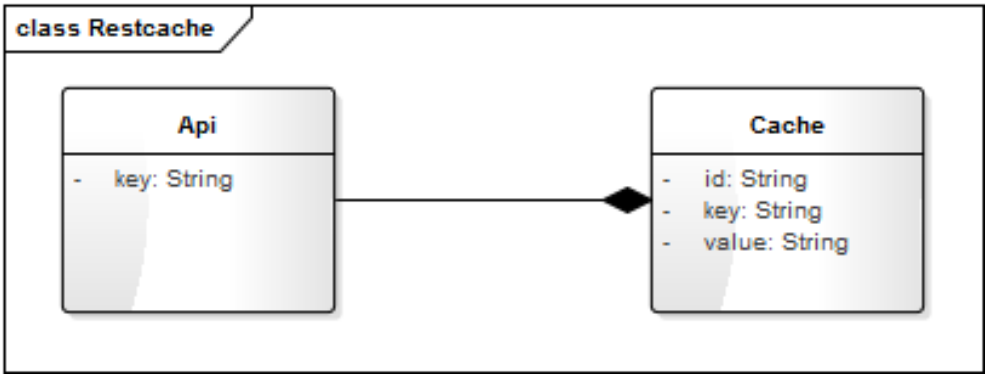
Aplikacja stworzona na potrzeby przeprowadzenia badań jest serwisem typu *RESTful*. Jej celem jest przechowywanie danych w bazie w postaci *klucz-wartość*. Aby móc skorzystać z aplikacji klient musi się autoryzować unikalnym kluczem (*API key*). Klient posiadający klucz może zapisywać i odczytywać dane.

Wykorzystywany w aplikacji model składa się z dwóch klas jednakowo odwzorowanych na kolekcjach bazy *MongoDB*. Pierwsza klasa, *Api*, służy do przechowywania w bazie danych informacji o zarejestrowanych kluczach *API* (*API key*). Indeksem głównym w kolekcji jest pole *key*.

Druga klasa, *Cache*, jest obiektem służącym do przechowywania danych *klucz-wartość*, które klient chce przechować w bazie. Kolekcja *Cache* posiada trzy indeksy:

- na pole *_id*,
- na pole *api*, by wyszukiwanie wszystkich obiektów zapisanych z danym kluczem *API key* było najszybsze,
- na parę pół (*api*, *key*), by wyszukiwanie pojedynczych obiektów klienta było jak najszybsze.

Diagram klas aplikacji przedstawiono na rysunku 3.1.



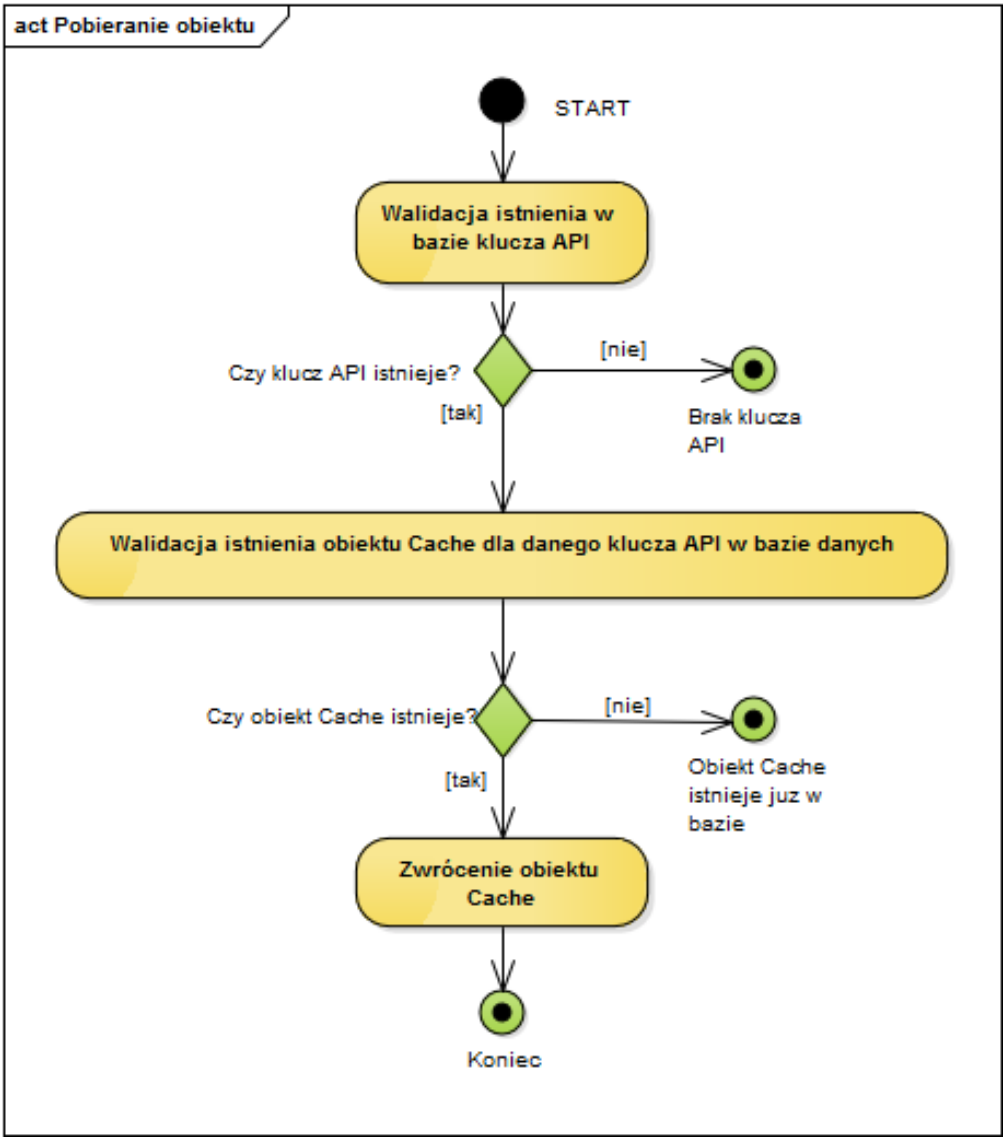
Rysunek 3.1: Diagram klas aplikacji wykorzystywanej do przeprowadzenia testów

Aplikacje udostępniają *API*, które przedstawia się w poniżej opisany sposób. Aby otrzymać klucz należy wykonać żądanie *GET /api/*. W odpowiedzi aplikacja zwróci unikalny, losowy klucz identyfikujący danego klienta. Do generowania kluczy użyty został mechanizm *UUID* (ang. *universally unique identifier*) generujący losowy, 128 bitowy ciąg znaków. Przykładowy identyfikator wygląda następująco: *f0778bae-2902-4ff6-93fa-9776403ecb0f*. W tym momencie klient posiadający swój klucz może tworzyć, aktualizować, usuwać i pobierać obiekty, które są zapisane w bazie danych. Każde żądanie musi w adresie zawierać otrzymany klucz. Początek żądania ma formę */api/{api_key}/*. W przypadku podania błędnego klucza lub jego braku aplikacja zwróci błąd autoryzacji.

Metoda *GET /api/{api_key}/{key}* pozwala na pobranie zapisanej wartości w bazie danych. Jeśli nie istnieje wartość o podanym kluczu zwracany jest błąd *HTTP 404 - Not Found*. Rysunek 3.2 prezentuje diagram aktywności dla pobierania obiektu w aplikacji.

Metoda *POST /api/{api_key}/{key}* pozwala na utworzenie obiektu *Cache* w bazie danych. Jeśli w bazie danych istnieje obiekt *Cache* o podanym kluczu zwracany jest błąd *HTTP 409 - Conflict*. Natomiast jeśli w przesyłanym żądaniu pole *value* nie zostanie zdefiniowane lub wartość będzie pusta (*null*), aplikacja zwróci błąd *HTTP 400 - Bad request*. Rysunek 3.3 prezentuje diagram aktywności dla tworzenia obiektu w aplikacji.

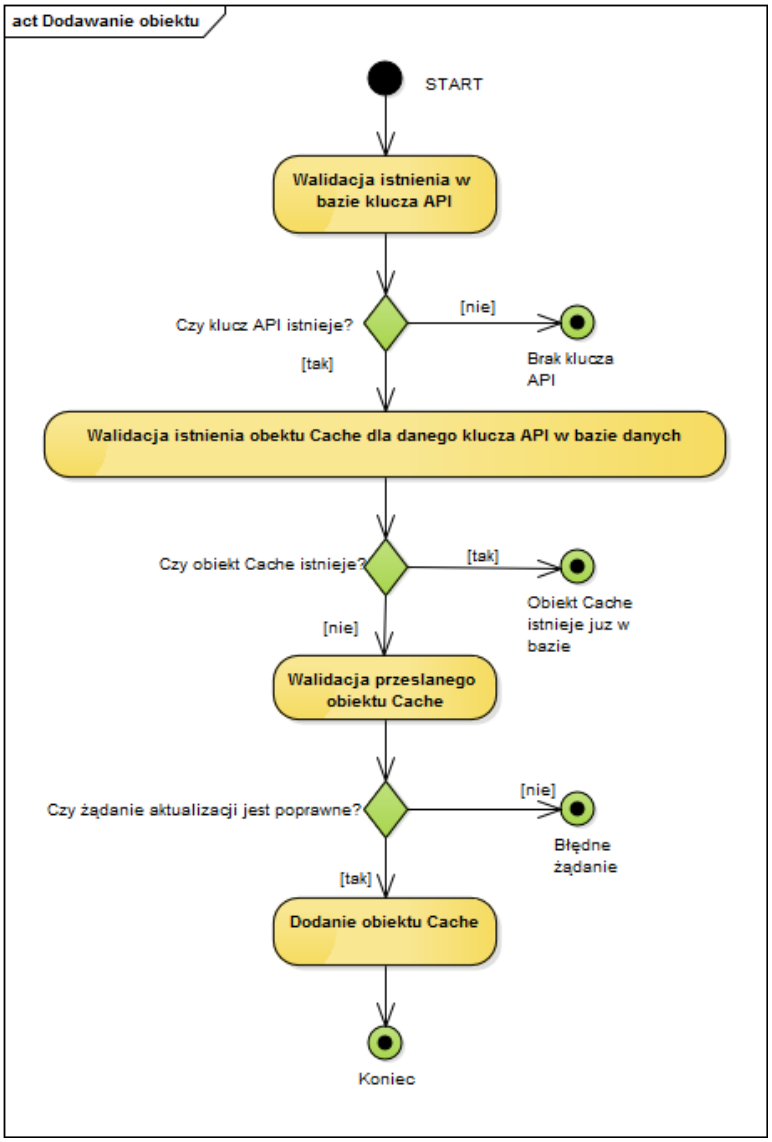
Metoda *PUT /api/{api_key}/{key}* pozwala na aktualizacje istniejącego



Rysunek 3.2: Diagram aktywności pobierania obiektu w aplikacji

obiektu *Cache* w bazie danych. Jeśli w bazie danych nie istnieje obiekt *Cache* o podanym kluczu zwracany jest błąd *HTTP 404 - Not Found*, a jeśli w przesyłanym żądaniu pole *value* nie zostanie zdefiniowane lub wartość będzie pusta (*null*), aplikacja zwróci błąd *HTTP 400 - Bad request*. Rysunek 3.4 prezentuje diagram aktywności dla aktualizowania obiektu w aplikacji.

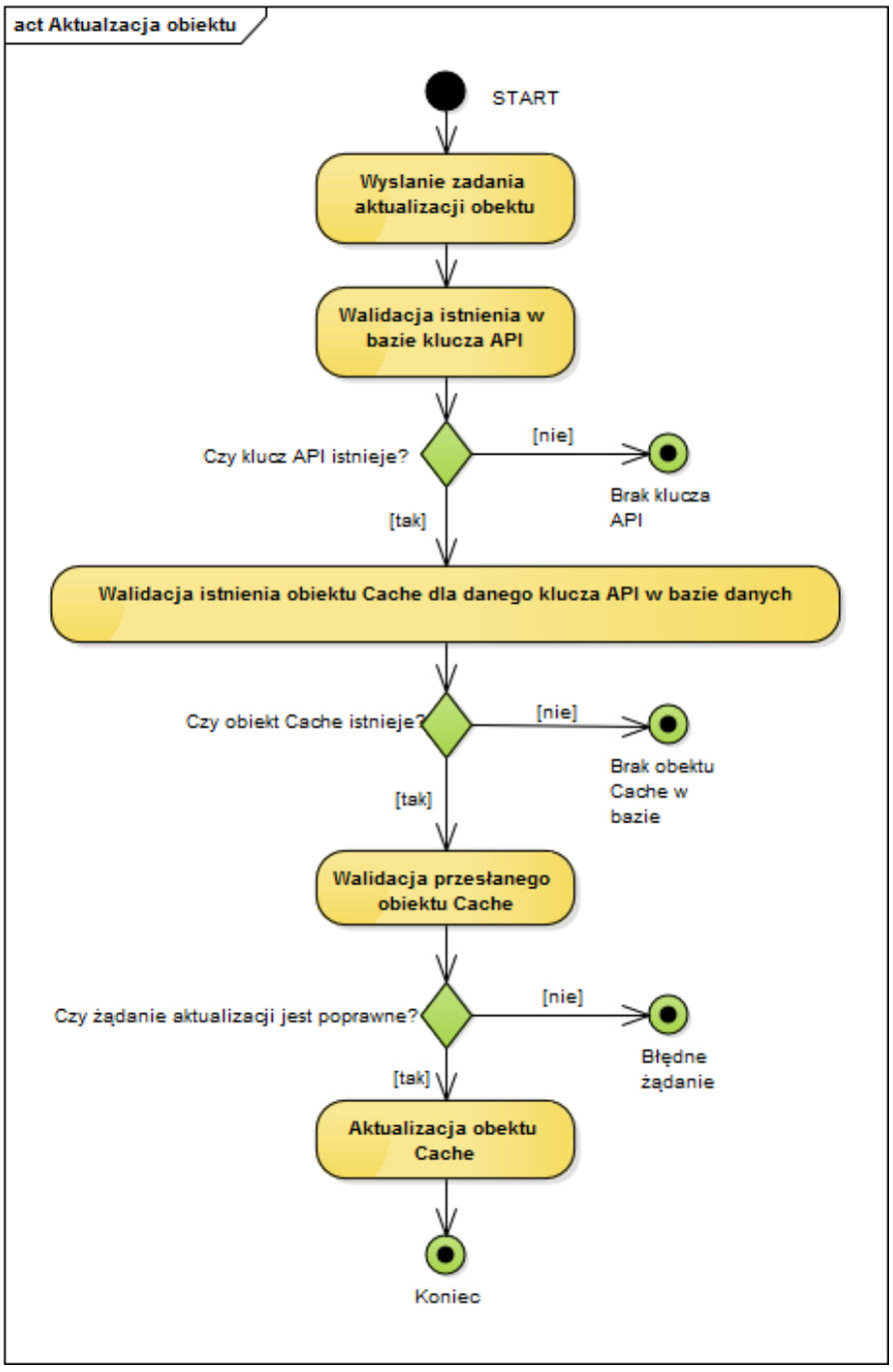
Metoda *DELETE /api/{api_key}/{key}* pozwala na usunięcie istniejącego obiektu *Cache* z bazy danych. Jeśli w bazie danych nie istnieje obiekt



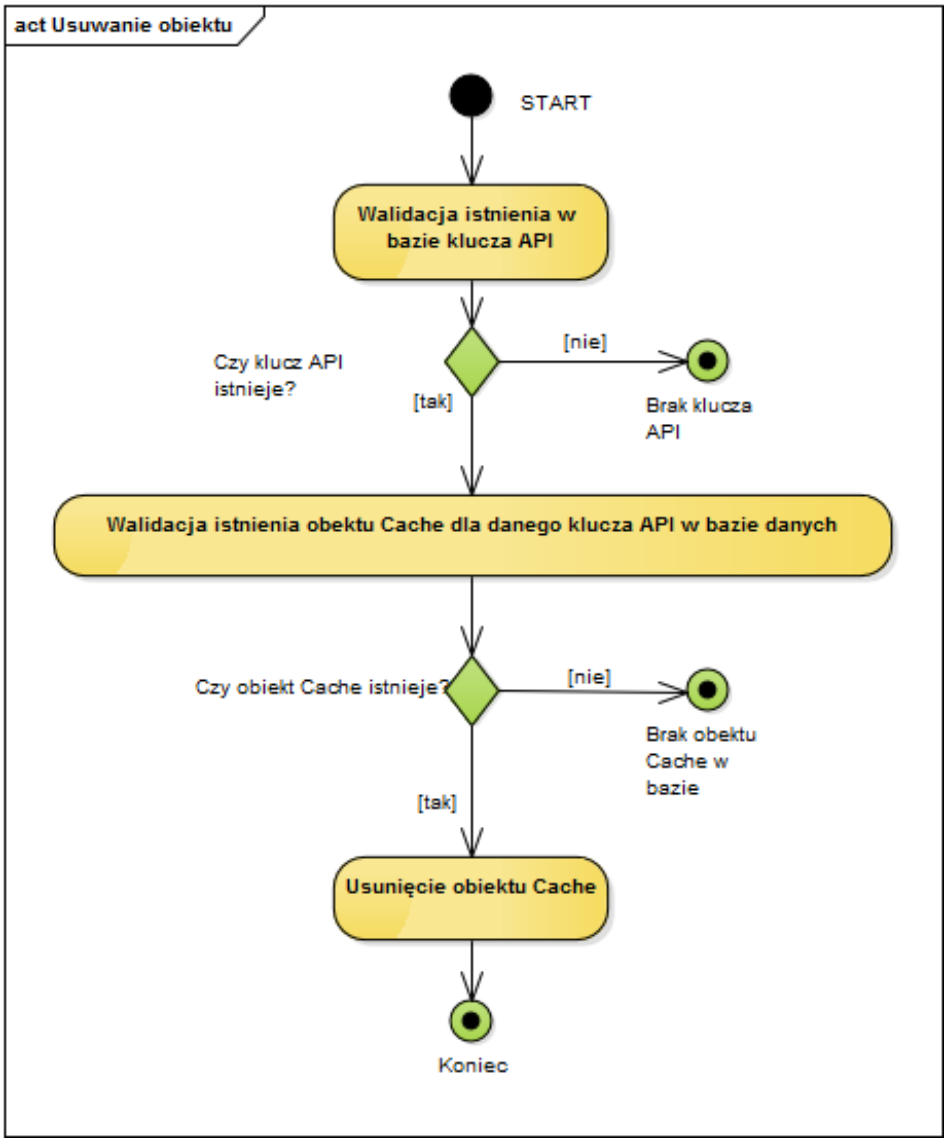
Rysunek 3.3: Diagram aktywności tworzenia obiektu w aplikacji

Cache o podanym kluczu zwracany jest błąd *HTTP 404 - Not Found*. Rysunek 3.5 prezentuje diagram aktywności dla usuwanie obiektu w aplikacji.

Ostatnią dostępną metodą jest żądanie *GET /api/{api_key}*, które pozwala na pobranie listy wszystkich obiektów, które klient o danym *api_key* zapisał.



Rysunek 3.4: Diagram aktywności aktualizowania obiektu w aplikacji



Rysunek 3.5: Diagram aktywności usuwanie obiektu w aplikacji

3.2 Testy akceptacyjne

By potwierdzić, że stworzone *API* aplikacji w językach *Java* i *Go* zachowuje się tak samo zostały przygotowane 24 testy akceptacyjne przy użyciu biblioteki *Spock*, która została opisana w rozdziale 4.5.

W dodatku A przedstawione są wyniki testów aplikacji uruchomionych na serwerze *Tomcat* (rysunek A.1), na serwerze *Jetty* (rysunek A.2) oraz w języku *Go* (rysunek A.3).

Rozdział 4

Narzędzia wykorzystane do przeprowadzenia testów

4.1 Docker

Docker [9] jest platformą dla programistów i administratorów systemów do tworzenia, dostarczania i uruchamiania aplikacji. Pozwala on zbudować aplikację z zależnościami, która będzie się zachowywać tak samo na środowisku produkcyjnym jak i programistycznym. Dzieje się tak, ponieważ budując aplikację tworzymy obrazy, które po zbudowaniu przenosimy na docelowe środowisko.

Docker opiera się na kontenerach linuxowych (LXC - Linux Containers). Kontenery w Linuxie są wirtualizacją na poziome systemu operacyjnego, która pozwala na separacje aplikacji od systemu i fizycznej infrastruktury wykorzystywanej m.in. do połączeń sieciowych czy plików. Każdy z kontenerów może uruchomić swój proces, który może mieć własnych użytkowników. W jednym systemie może być uruchomiona nieograniczona ilość kontenerów. Konteneryzacja, w przeciwieństwie do wirtualizacji, oferuje niewielki narzut na zasoby systemu. Uruchomienie pojedynczego kontenera ogranicza się do wykonania kilku standardowych poleceń systemowych.

Do testów w pracy użyto Docker w wersji 1.9.0.

4.2 MongoDB

MongoDB [10] jest nierelacyjną bazą danych (ang. *NoSQL database*). Główną cechą tej bazy jest brak ściśle zdefiniowanej struktury. Dane w bazie przechowywane są w postaci dokumentów. Dokument jest niczym innym jak obiektem typu *JSON*. Sama baza przechowuje jednak dokumenty w formie binarnej, w skrócie nazywanej *BSON* (*Binary JSON*). Dokument sam w sobie jest strukturą złożoną z par klucz-wartość. Wartości w dokumencie mogą być cyframi, napisami, wartościami logicznymi (*boolean*), tablicami, jak również mogą zagnieżdżać inne dokumenty czy tablice dokumentów. Pojedyncze dokumenty przechowywane są w kolekcji. Kolekcja natomiast musi należeć do określonej bazy.

Każdy dokument w kolekcji musi mieć swój unikalny identyfikator. Kluczem w dokumencie oznaczającym identyfikator jest *_id*. To po identyfikatorze można zmienić pojedynczy dokument. Jeśli zapisywany dokument nie będzie zawierał identyfikatora baza danych sama wypełni go swoim własnym. Identyfikatory są indeksowane, co pozwala przyspieszyć operacje na dokumentach. Projektanci bazy ograniczyli rozmiar pojedynczego dokumentu do *16MB*. Do przechowywania większych plików trzeba zastosować inne rozwiązanie.

Przykład 4.1 prezentuje przykładowy dokument, który zapisany jest w bazie *MongoDB*

Przykład 4.1: Przykład dokumentu zapisanego w bazie MongoDB

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: {
        first: "Jan",
        last: "Kowalski"
    },
    birth: new Date('Jan 01, 1970'),
    email: "jan@kowalski.pl"
}
```

To, że kolekcje w *MongoDB* nie posiadają zdefiniowanej struktury oznacza, że w kolekcji można zapisać dwa dokumenty, gdzie jedynym wspólnym kluczem będzie *_id*. Również w kolekcji mogą się znaleźć dokumenty, które temu samemu polu przypiszą wartości dwóch różnych typów np. boolean i string. Indeksy na kolekcji są bardzo zbliżone do tych znanych z baz relacyjnych. Administrator może założyć indeks unikalny i nieunikalny na jedną lub więcej kolumn. Dzięki założonemu indeksowi czas wyszukiwania dokumentów znacząco skróci się.

Dokumenty w *MongoDB*, w porównaniu do baz relacyjnych, nie mogą posiadać relacji innych niż zagnieżdżenia. Kolejną różnicą jest brak transakcji. W *MongoDB* nie można wykonać zbioru poleceń. Również zasada atomowości różni się od tej z baz relacyjnych. Jeśli przeprowadzana jest pojedyncza zmiana wielu dokumentów, to zmiana może się udać w całości lub wcale na pojedynczym dokumencie, a nie całym zbiorze, których dotyczy. Brak tych elementów pozwolił na zachowanie wysokiej wydajności, gdy baza składa się z wielu węzłów.

MongoDB znalazł zastosowanie wśród projektów operujących na dużej ilości danych (*Big Data*). Dzięki prostej konfiguracji można uruchomić instancje bazy na wielu węzłach, by zabezpieczyć dane przed utratą. Baza posiada wiele funkcji agregujących, pozwalających na analizę zapisanych danych na wielu węzłach równocześnie.

Do testów w pracy użyto bazę *MongoDB* w wersji *3.0*.

4.3 Apache JMeter

Apache JMeter [11] jest programem służącym do wykonywania testów aplikacji w celu zmierzenia ich wydajności. Początkowo wykorzystywany był do tworzenia testów serwisów internetowych. Z czasem został rozszerzony o dodatkowe funkcje. *Apache JMeter* można użyć do symulowania wysokiego obciążenia aplikacji na serwerze, sieci lub innych testowanych obiektach. Obecnie *Apache JMeter* można zastosować do testowania serwerów i protokołów:

- HTTP,

- HTTPS,
- FTP,
- SOAP oraz REST,
- relacyjne bazy danych - przy użyciu sterownika JDBC,
- nierelacyjne bazy danych np. MongoDB,
- usług pocztowych wykorzystujących protokoły: SMTP, POP3 oraz IMAP,
- TCP.

Apache JMeter można rozszerzać o własne pluginy, więc lista usług dostępnych do testowania jest nieograniczona.

Apache JMeter jest wielowątkowym narzędziem, przez co można wykonywać ten sam test równolegle symulując w ten sposób wielu użytkowników.

Pierwsza stabilna wersja Apache JMeter została wydana 15 grudnia 1998r. Do testów w pracy użyto wersji 2.13.

4.4 Digitalocean

Digitalocean [12] jest usługą pozwalającą na zakup wirtualnych serwerów w tzw. chmurze. Zastosowany model chmury to *Infrastructure as a Service* (skrót: *IAAS*, z ang. infrastruktura jako usługa), dzięki której klient może kupić zasoby sprzętowe (dysk, procesor, łącze, pamięć RAM) w zależności od potrzeb i budżetu. Dla klientów zaletą takiego modelu jest możliwość zakupu tylko wtedy, gdy dana usługa jest potrzeba. Kolejną zaletą jest możliwość dostosowania potrzebnych zasobów w zależności od obciążenia aplikacji.

Digitalocean pozwala na wybór lokalizacji, w której kupowany serwer wirtualny będzie uruchomiony. Wśród obecnie dostępnych lokalizacji są Amsterdam, Frankfurt, Londyn, Nowy Jork, Singapur, San Francisco oraz Toronto. Taka możliwość pozwala tworzyć i uruchamiać aplikację w serwerowniach jak najbliżej docelowych użytkowników.

4.5 Groovy i Spock

Groovy jest językiem obiektowym podobnym do języka *Java*. Język ten kompilowany jest do kodu bajtowego *Java* dzięki czemu może być uruchomiony na jego maszynie wirtualnej.

Groovy wprowadza kilka funkcji, które nie były dostępne we wcześniejszych wersjach języka *Java*. Pierwszą z nich jest możliwość stosowania domknięć (ang. *closure*), które pojawiły się dopiero w 8 wersji języka *Java*, w postaci wyrażeń *lambda*. *Groovy* pozwala również na tworzenie klas na dwa sposoby: dynamicznie interpretowanych oraz statycznie kompilowanych. Ostatnią funkcją dostępną w *Groovy* jest możliwość tworzenia języków dziedzinowych (ang. *domain-specific language*). Funkcja ta jest wykorzystywana w bibliotece *Spock* służącej do testowania tworzonego oprogramowania.

Biblioteka *Spock* służy do tworzenia testów jednostkowych i integracyjnych w zachowaniu praktyki *BDD* (ang. *behavior-driven development*). Biblioteka wymaga języka *Groovy*, a do swojego działania wykorzystuje m.in. bibliotekę *JUnit* do uruchamiania testów oraz *Mockito* do tworzenia atrap obiektów (ang. *mock*).

W przeciwieństwie do innych bibliotek służących do testowania, w *Spock* tworzy się scenariusze, które dzielą się na trzy główne części:

- *given* - gdzie określane są założenia początkowe,
- *when* - gdzie określana jest funkcja, którą się testuje,
- *then* - gdzie sprawdzane jest, czy testowana funkcja zachowała się poprawnie.

Given-When-Then jest stylem tworzenia specyfikacji zachowań aplikacji, który został zapoczątkowany wraz z metodyką *BDD* (ang. *behavior-driven development*). *BDD* polega na tworzeniu oprogramowania poprzez opisywanie zachowania. Wspólnie z metodyką *TDD* (ang. *test-driven development*), która polega na tworzeniu oprogramowania sterowanego testami, należą do zbioru metodyk zwinnego wytwarzania oprogramowania *Agile*.

Rozdział 5

Testy wydajnościowe

5.1 Testy wydajnościowe oprogramowania

Testowanie oprogramowanie jest ważnym procesem związanym z jego wytwarzaniem. Celem testowania jest sprawdzenie czy oprogramowanie jest zgodne ze specyfikacją oraz czy system spełnia oczekiwania klienta. Wykorzystuje się do tego różnego rodzaju testy. Jednymi z nich są testy sprawdzające wydajność oprogramowania.

Testy wydajnościowe pozwalają sprawdzić jak oprogramowanie zachowa się w zależności od obciążenia serwera, bazy danych oraz samej aplikacji w oparciu o przygotowane scenariusze użycia oraz wygenerowanych wirtualnych użytkowników, którzy te scenariusze wykonują. Można zbadać, czy nie nastąpi przerwa w działaniu aplikacji, ile czasu będą zajmować poszczególne funkcje aplikacji, jak wykorzystane będą zasoby serwera oraz czy aplikacja pozwala na skalowanie. Testy wydajnościowe mogą przyjmować różną formę.

Pierwszą formą są testy obciążeniowe. W takim teście ustala się ilu użytkowników jednocześnie ma korzystać z aplikacji. Na podstawie otrzymanych wyników z takiego testu można zbadać jak aplikacja zachowa się podczas określonego obciążenia oraz otrzymać informacje o czasach odpowiedzi aplikacji.

Drugą formą są testy przeciążeniowe. Pozwalają one określić jak zachowa się aplikacja w warunkach skrajnego obciążenia. Przykładem takiego testu jest zbyt duża liczba użytkowników aplikacji korzystających w tym samym

czasie. Dzięki takim testom można zaobserwować między innymi, czy i w jakim momencie aplikacja wyłączy się.

Testy wydajnościowe w aplikacjach internetowych przeprowadza się, by zbadać jak dużo użytkowników może korzystać z systemu jednocześnie. Takie testy można przeprowadzić zarówno z jednego komputera, jak i z kilku.

W aplikacjach mogą być również badane czasy odpowiedzi aplikacji w zależności od liczby równoległych żądań. Zazwyczaj testy takie są stosowane, gdy aplikacja w swojej specyfikacji ma określony maksymalny czas odpowiedzi przy danej liczbie żądań.

Testy wydajnościowe powinny być prowadzone na środowisku testowym identycznym do produkcyjnego.

5.2 Scenariusze testów wydajnościowych wykorzystanych do badań

Do przeprowadzenia testów wydajnościowych przygotowano dwie aplikacje. Pierwszą z nich była aplikacja w języku *Java* uruchomiona na dwóch różnych serwerach: *Tomcat* i *Jetty*, które obecnie są jednymi z najpopularniejszych rozwiązań służących do uruchamiania aplikacji *Java*. Drugą była aplikacja w języku *Go*, która przy użyciu biblioteki *HttpRouter* [13] pozwala na tworzenie aplikacji działającej jako serwer *HTTP*.

Każda aplikacja była testowana przez przygotowany zbiór testów w aplikacji *Apache JMeter*. *Apache JMeter* pozwala na tworzenie i wykonywanie testów wydajnościowych, symulujących wielu klientów korzystających z aplikacji. Przygotowane testy zostały podzielone na 4 grupy:

- testy sprawdzające wydajność walidacji istnienia klucza *API*,
- testy sprawdzające wydajność walidacji obiektu *Cache*,
- testy sprawdzające wydajność operacji typu *CRUD*,
- testy sprawdzające wydajność powyższych scenariuszy równolegle.

Pierwsza grupa sprawdza wydajność aplikacji, gdy klient chciał wykonać operacje posiadając nieistniejący klucz *API*. Na tą grupę składało się pięć testów, wykonywanych w następującej kolejności:

1. pobieranie listy wszystkich obiektów *Cache*,
2. pobieranie pojedynczego obiektu *Cache*,
3. tworzenie obiektu *Cache*,
4. aktualizacja obiektu *Cache*,
5. usunięcie obiektu *Cache*.

Każde żądanie kończyło się sukcesem, gdy aplikacja zwracała błąd autoryzacji.

Druga grupa testów sprawdza wydajność, gdy klucz *API* istniał, jednak klient chciał przeprowadzić operacje pobierania, usuwania i aktualizacji nie istniejącego obiektu *Cache*. Scenariusz grupy wyglądał następująco:

1. pobierz nowy klucz *API*,
2. pobierz obiekt *Cache* autoryzując się otrzymanym kluczem *API*,
3. zaktualizuj obiekt *Cache* autoryzując się otrzymanym kluczem *API*,
4. usuń obiekt *Cache* autoryzując się otrzymanym kluczem *API*.

Każde żądanie kończyło się sukcesem, gdy aplikacja zwracała błąd podczas modyfikacji obiektu *Cache*.

Kolejną grupą są testy wydajności operacji *CRUD*. Do przeprowadzenia tej grupy testów został przygotowany zbiór 100 tysięcy losowych wartości w formie pliku *CSV*. Każda wartość składała się 3 części. Pierwszą był klucz obiektu *Cache*, natomiast dwie kolejne to wartości, które zostaną zapisane w aplikacji w polu *value* obiektu. *Apache JMeter* pozwala na przekazanie pliku *CSV*, którego wartości można użyć do przeprowadzenia testów. Scenariusz tej grupy wyglądał następująco:

1. pobierz nowy klucz *API*,
2. utwórz obiekt *Cache* o kluczu i wartości otrzymanym z parametru,
3. pobierz utworzony obiekt *Cache*,
4. zaktualizuj obiekt *Cache* ustawiając nową wartość pola *value*,
5. usuń obiekt *Cache*.

Każdy żądanie oznaczane było jako poprawne, gdy aplikacja nie zwracała błędu.

Ostatnią grupę stanowiły wszystkie testy, opisane w powyższych grupach.

Wszystkie grupy były wykonywane w 15 minutowych cyklach oddzielonych 60 sekundową przerwą.

Każda aplikacja testowana była w czterech przypadkach testowych, które różniły się od siebie liczbą klientów równolegle wykonujących żądania oraz stanem początkowym bazą danych. Były to:

- 100 klientów oraz pusta baza danych,
- 250 klientów oraz pusta baza danych,
- 100 klientów oraz baza danych wypełniona danymi,
- 250 klientów oraz baza danych wypełniona danymi.

W dwóch ostatnich przypadkach baza danych była wypełniona losowymi danymi: 4000000 obiektów w kolekcji *api*, 40000000 obiektów w kolekcji *cache*. Łącznie baza danych zajmowała 12 gigabajtów.

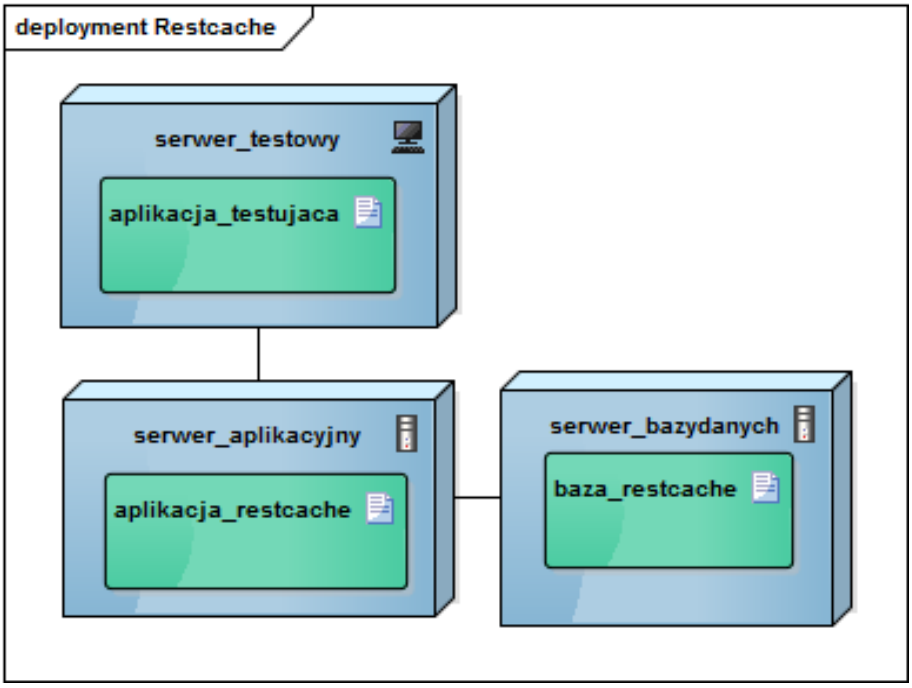
5.3 Środowisko testowe

Do przeprowadzenia testów wydajnościowych wykorzystano 3 serwery wirtualne o poniższych specyfikacjach technicznych:

- 8 rdzeni, 16 gigabajtów pamięci RAM, 160 gigabajtów dysku SSD dla serwera aplikacyjnego,
- 4 rdzenie, 8 gigabajtów pamięci RAM, 80 gigabajtów dysku SSD dla serwera bazy danych,
- 4 rdzenie, 8 gigabajtów pamięci RAM, 80 gigabajtów dysku SSD dla serwera, na którym uruchomiony był program *Apache JMeter*.

Serwery komunikowały się po sieci lokalnej (ang. *LAN*) bezpośrednio między sobą.

Na rysunku 5.1 zaprezentowany został diagram wdrożenia infrastruktury wykorzystanej do przeprowadzenia testów.



Rysunek 5.1: Diagram wdrożenia infrastruktury wykorzystywanej do przeprowadzenia testów

Rozdział 6

Wyniki testów

Wyniki testów każdej aplikacji uruchamianej w poszczególnych przypadkach testowych przedstawiono w formie wykresów: wykresu przedstawiającego liczbę żądań obsłużonych przez aplikację w ciągu sekundy i wykresu rozkładu czasów odpowiedzi aplikacji.

Wyniki testów podzielono na dwie grupy zależne od początkowego stanu bazy danych.

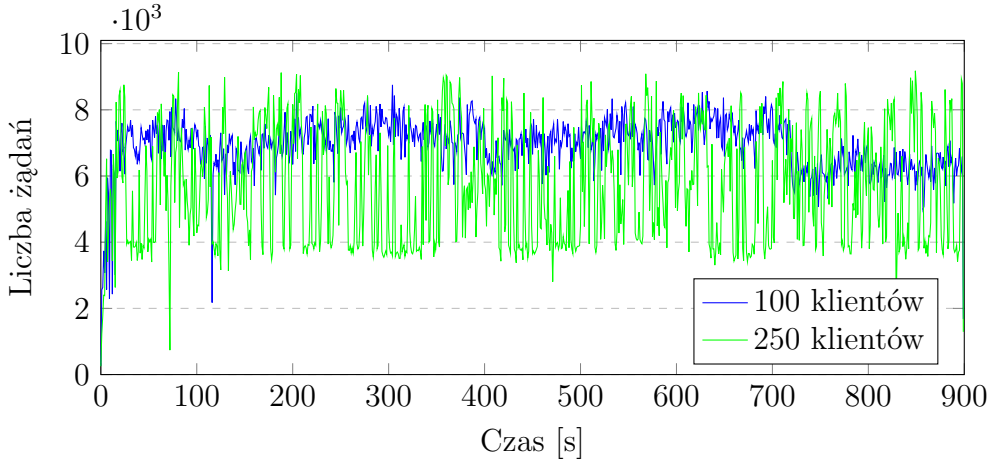
6.1 Testy z pustą bazą danych

6.1.1 Test wydajności walidacji API

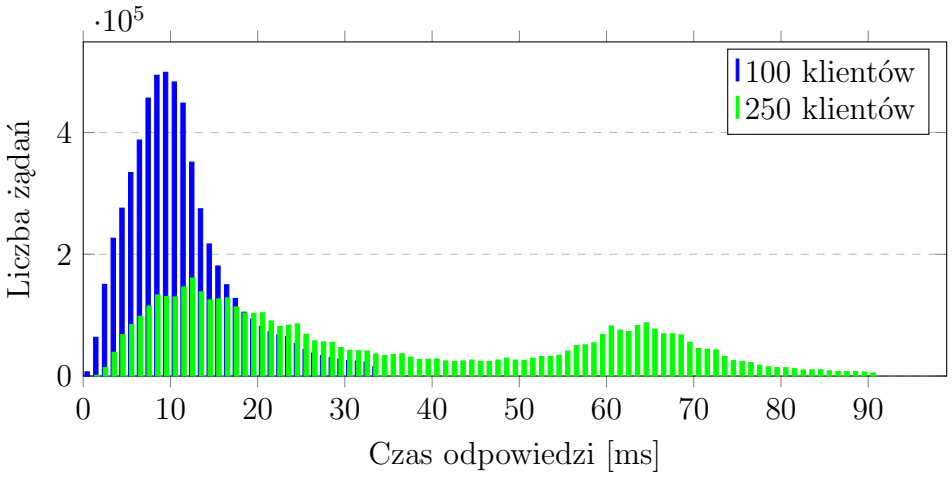
Wyniki testów wydajności walidujących istnienie klucza API przedstawiają wykresy zamieszczone na rysunkach 6.1 - 6.6.

Z wykresów prezentujących liczbę żądań obsłużonych w ciągu sekundy przez poszczególne aplikacje (rys. 6.1, 6.3, 6.5) wynika, że przy 100 klientach największą liczbę żądań obsłużyła aplikacja napisana w *Go* - od 9.0 do 11.0 tysięcy obsłużonych żądań. Wydajność serwera *Jetty* oscylowała w przedziale od 5.0 do 10.0 tysięcy obsłużonych żądań, przy znacznych wahaniach przepustowości. Przepustowość serwera *Tomcat* oscylowała w przedziale od 6.0 do 8.0 tysięcy obsłużonych żądań. Przy 250 klientach wydajność aplikacji w *Go* oscylowała w przedziale od 8.0 do 10.0 tysięcy obsłużonych żądań, a serwery *Jetty* i *Tomcat* obsłużyły od 4.0 do 8.0 tysięcy żądań przy dużych wahaniach przepustowości.

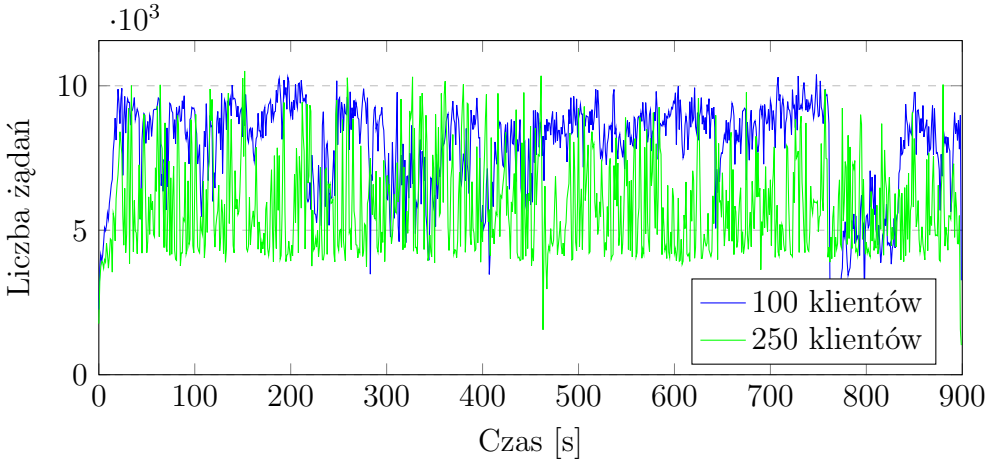
Z wykresów rozkładu czasów odpowiedzi aplikacji (rys. 6.2, 6.4, 6.6) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 13.70 milisekund, dla serwera *Jetty* 11.42 milisekund, a dla aplikacji w *Go* 9,44 milisekundy. Najdłużej trwające żądania trwały około 35 milisekund w przypadku serwerów *Tomcat* i *Jetty* oraz 18 milisekund przy aplikacji w *Go*. Przy 250 klientach średnie czasy odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* były prawie takie same i wynosiły odpowiednio 36.32 i 35.52 milisekund, a w aplikacji w *Go* 22.47 milisekundy. Najdłużej trwające żądania trwały ponad 90 milisekund w przypadku serwera *Tomcat*, poniżej 88 milisekund w przypadku serwera *Jetty* i około 55 milisekundy przy aplikacji w *Go*. Dodatkowo, przy 250 klientach, dla serwerów *Tomcat* i *Jetty* rozkłady czasów odpowiedzi były spłaszczone.



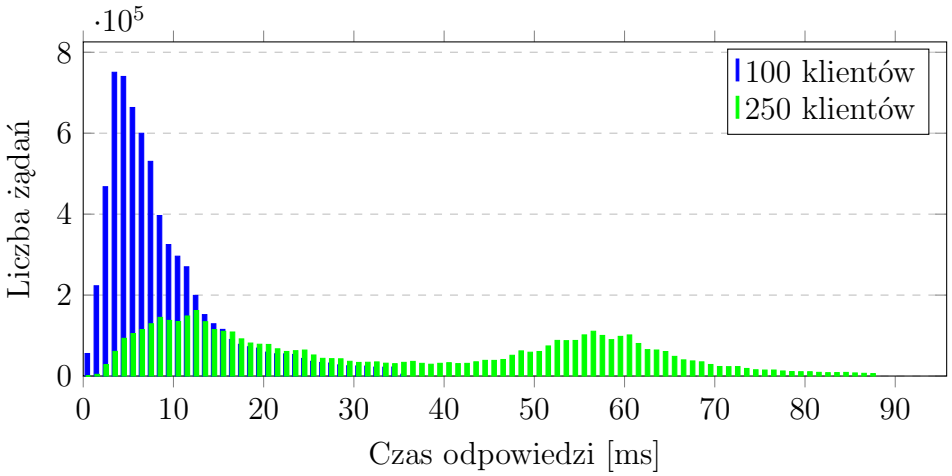
Rysunek 6.1: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia klucza API



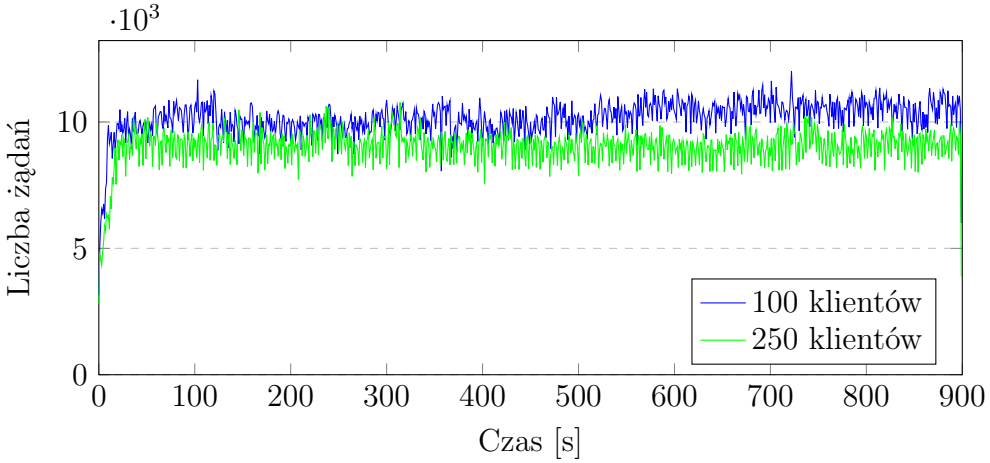
Rysunek 6.2: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia klucza API



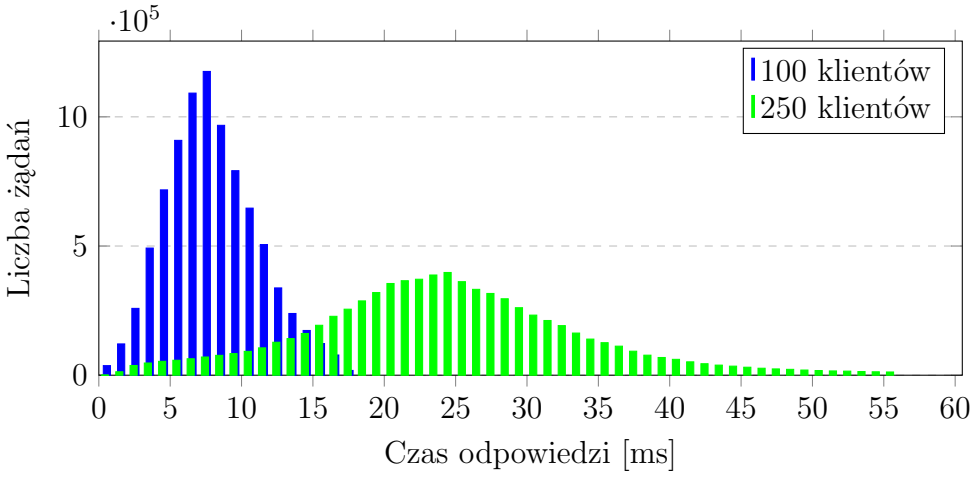
Rysunek 6.3: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia klucza API



Rysunek 6.4: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia klucza API



Rysunek 6.5: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia klucza API



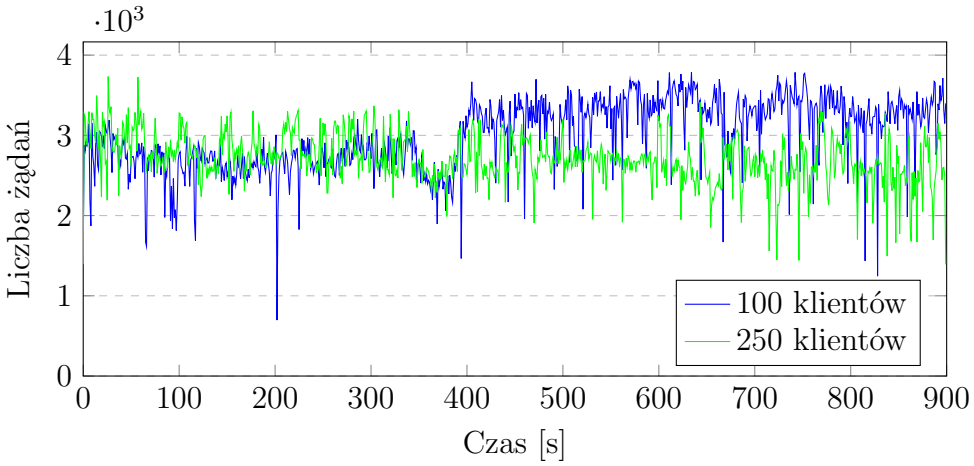
Rysunek 6.6: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia klucza API

6.1.2 Test wydajności walidacji istnienia obiektów Cache

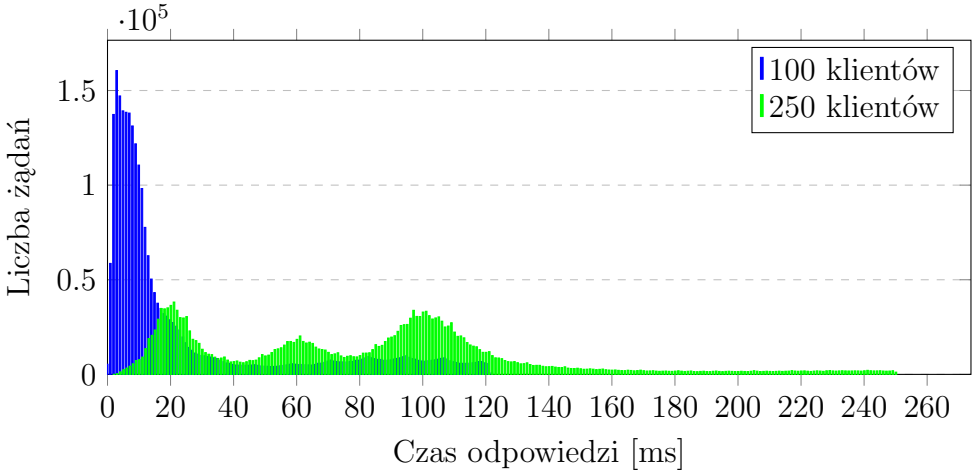
Wyniki testów wydajności walidacji istnienia obiektów Cache przedstawiają wykresy na rysunkach 6.7 - 6.12.

Z wykresów przedstawiających rozkład ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. 6.7, 6.9, 6.11) wynika, że przy 100 klientach największą liczbę żądań (około 7.0 tysięcy) obsłużyła aplikacja napisana w *Go*. Wydajność serwera *Jetty* oscylowała w przedziale od 3.5 do 4.5 tysiąca obsłużonych żądań, a serwera *Tomcat* w przedziale od 2.0 do 3.5 tysiąca obsłużonych żądań. Przy 250 klientach wydajność aplikacji w *Go* oscylowała w przedziale od 7.0 do 8.0 tysięcy obsłużonych żądań, serwer *Jetty* obsłużył około 3.0 tysięcy żądań, a serwer *Tomcat* od 2.0 do 3.5 tysiąca żądań.

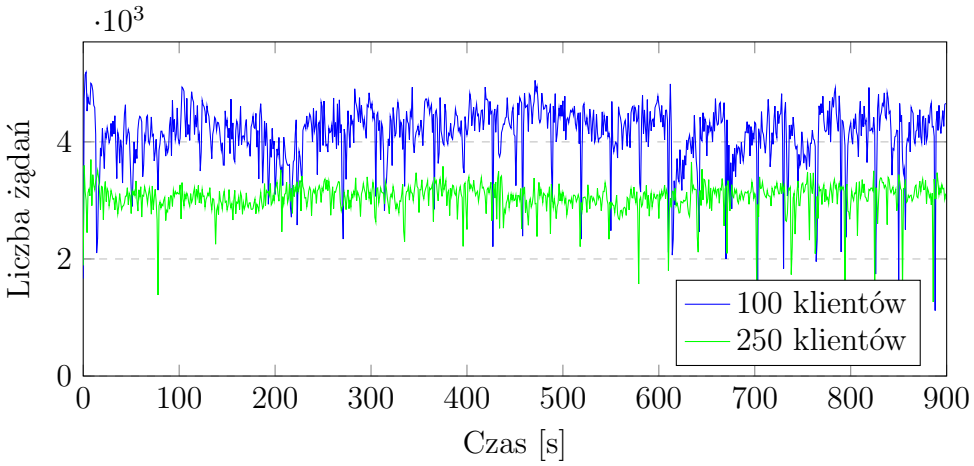
Z wykresów rozkładu czasów odpowiedzi (rys. 6.8, 6.10, 6.12) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 30.32 milisekund, dla serwera *Jetty* 23.32 milisekund, a dla aplikacji w *Go* 13.79 milisekund. Najdłużej trwające żądania trwały około 120 milisekund w przypadku serwera *Tomcat*, 45 milisekund przy *Jetty* i tylko 25 milisekund w aplikacji w *Go*. Przy 250 klientach średnie czasy odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły odpowiednio 69.97 i 76.92 milisekundy, a w aplikacji w *Go* tylko 30.75 milisekundy. Najdłużej trwające żądania trwały 250 milisekund w przypadku serwera *Tomcat*, około 120 milisekund w przypadku serwera *Jetty* i tylko 65 milisekundy przy aplikacji w *Go*. Czasy odpowiedzi serwera *Tomcat* w sporej części były dużo dłuższe niż pokazuje to średnia.



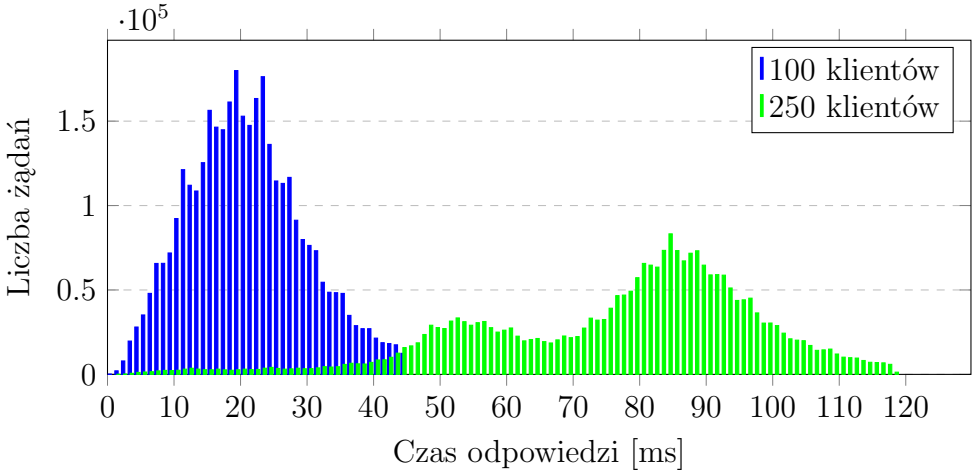
Rysunek 6.7: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia obiektu Cache



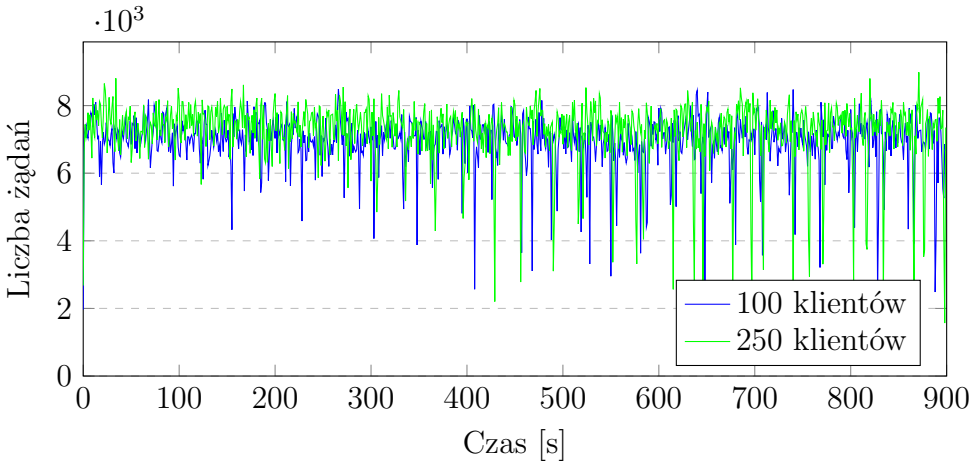
Rysunek 6.8: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia obiektu Cache



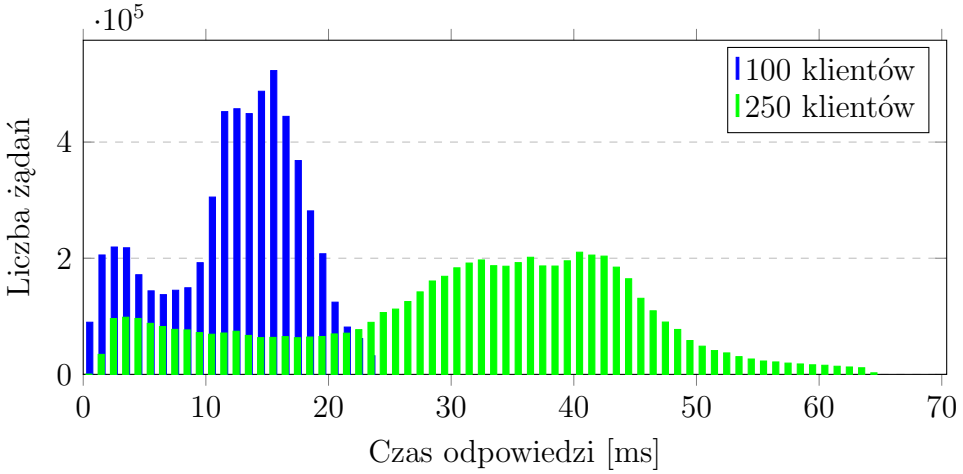
Rysunek 6.9: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia obiektu Cache



Rysunek 6.10: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia obiektu Cache



Rysunek 6.11: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia obiektu Cache



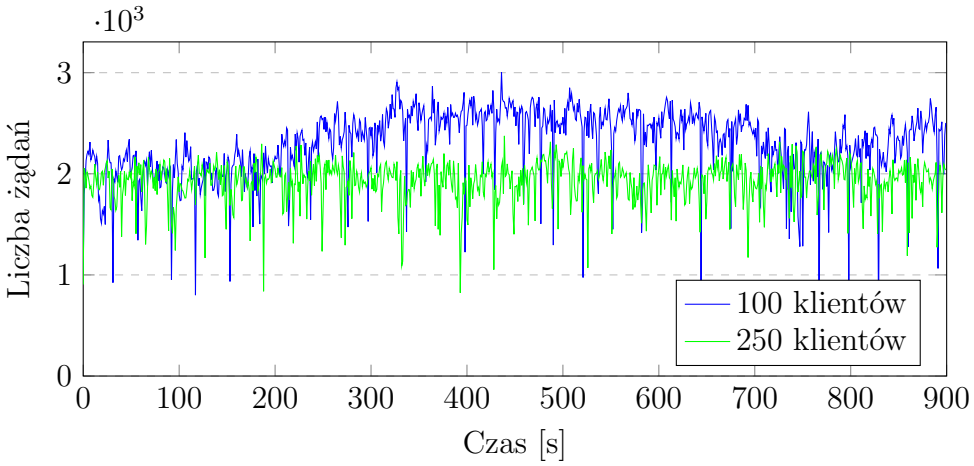
Rysunek 6.12: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia obiektu Cache

6.1.3 Test wydajności operacji CRUD

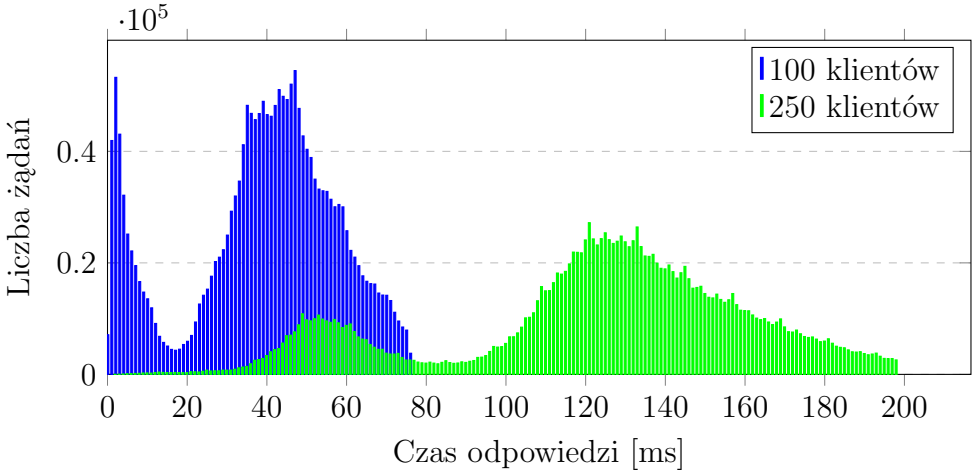
Wyniki testów wydajności operacji CRUD przedstawiają wykresy na rysunkach 6.13 - 6.18.

Z wykresów przedstawiających rozkład ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. 6.13, 6.15, 6.17) wynika, że przy przy 100 klientach największą liczbę żądań obsłużyła aplikacja napisana w *Go* - od 4.0 do 5.0 tysięcy. Serwer *Jetty* obsługiwał ponad 2.2 tysiące żądań, a przepustowość serwera *Tomcat* oscylowała w przedziale od 2.0 do 2.5 tysiąca obsłużonych żądań. Przy 250 klientach aplikacja w *Go* obsłużyła od 4.5 do 6.0 tysięcy żądań, serwer *Jetty* około 2.0 tysięcy żądań podobnie jak serwer *Tomcat*.

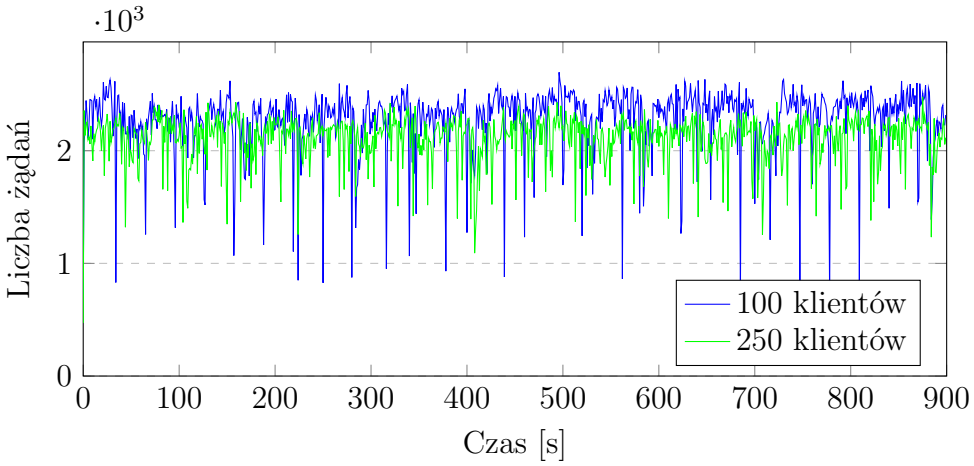
Z wykresów rozkładu czasów odpowiedzi (rys. 6.14, 6.16, 6.18) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 42.15 milisekund, dla serwera *Jetty* 42.41 milisekund i 20.90 milisekund dla aplikacji w *Go*. Najdłużej trwające żądania trwały poniżej 80 milisekund w przypadku serwerów *Tomcat* i *Jetty* - jednak *Jetty* miał większą liczbę obsłużonych żądań. Dla aplikacji w *Go* najdłużej trwające żądania trwały tylko 35 milisekund. Przy 250 klientach średnie czas odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły odpowiednio 107.56 i 92.32 milisekund, a w aplikacji w *Go* tylko 35.68 milisekundy. Najdłużej trwające żądania trwały 200 milisekund w przypadku serwera *Tomcat*, 180 milisekund w przypadku serwera *Jetty* i 90 milisekund przy aplikacji w *Go*.



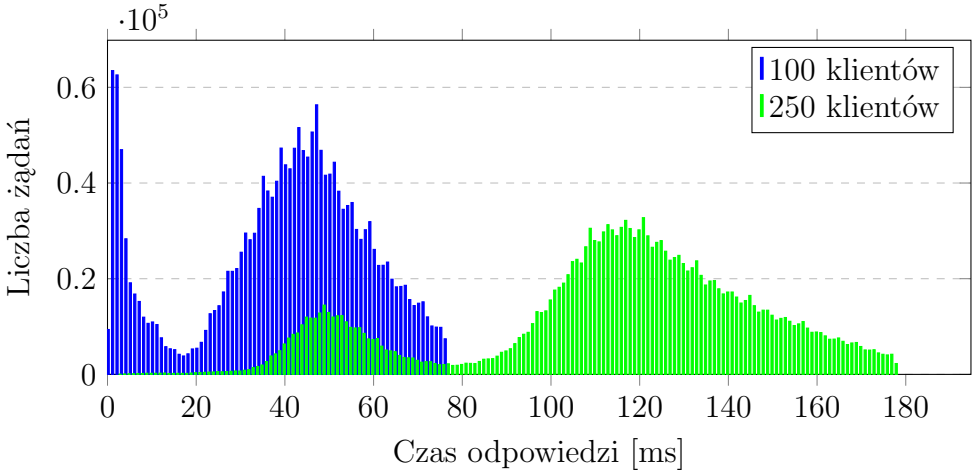
Rysunek 6.13: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu operacji CRUD



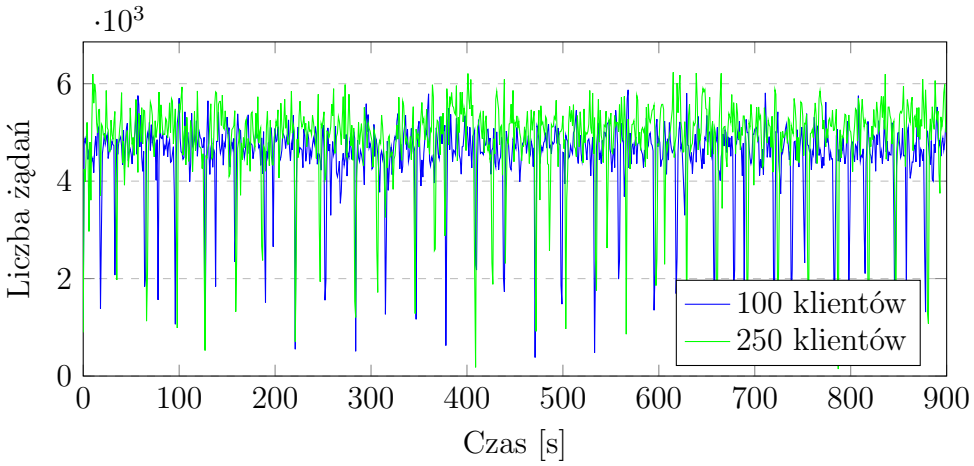
Rysunek 6.14: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu operacji CRUD



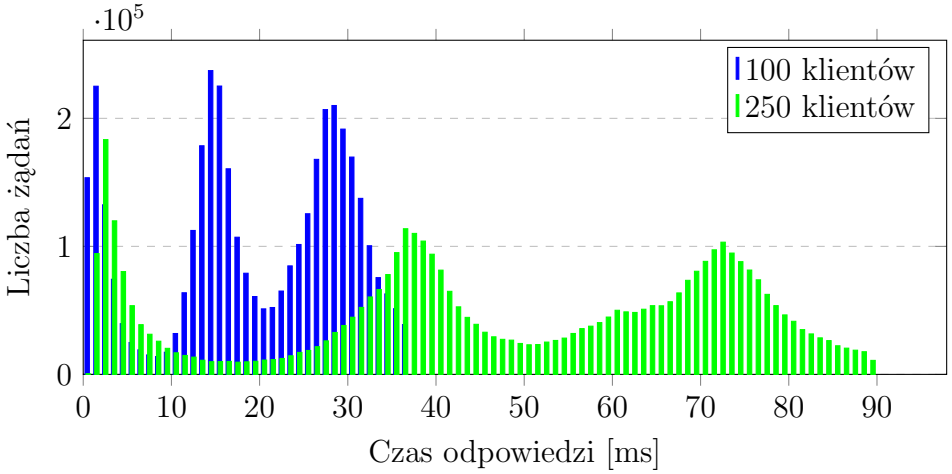
Rysunek 6.15: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu operacji CRUD



Rysunek 6.16: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu operacji CRUD



Rysunek 6.17: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu operacji CRUD



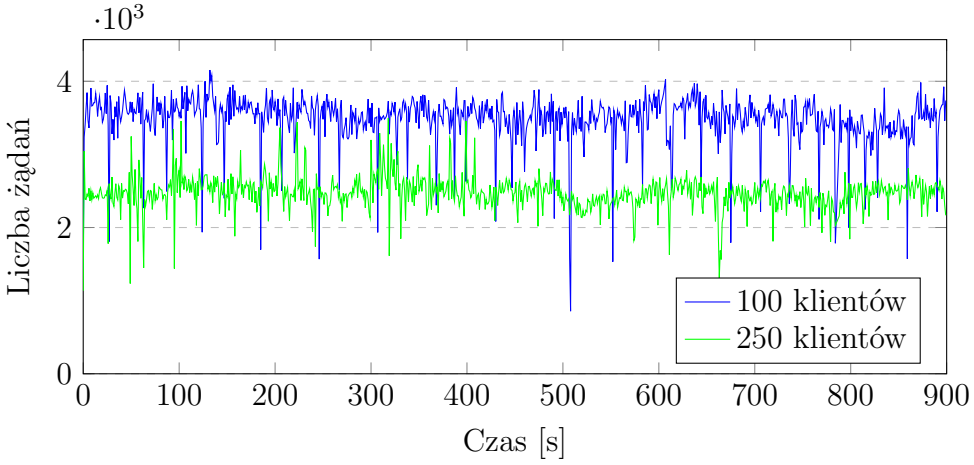
Rysunek 6.18: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu operacji CRUD

6.1.4 Test wydajności walidacji API, obiektów Cache oraz operacji CRUD równolegle

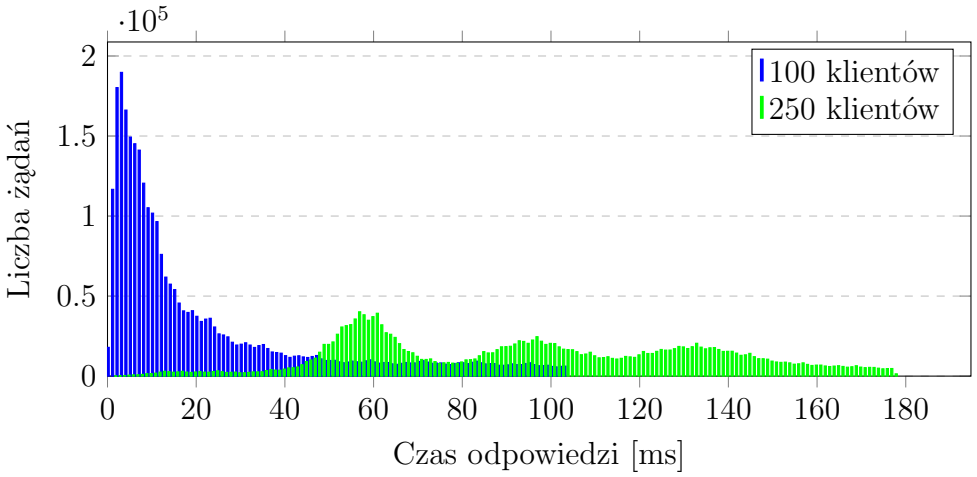
Diagramy zawierające wyniki wydajności walidacji API, obiektów Cache oraz operacji CRUD równolegle zamieszczono na rys. 6.19 - 6.24.

Z rozkładów ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. 6.19, 6.21, 6.23) wynika, że przy 100 klientach największą liczbę żądań obsłużyła aplikacja napisana w *Go* - od 7.0 do 8.0 tysięcy, a serwery *Jetty* i *Tomcat* obsługiwały od 3.0 do 4.0 tysięcy żądań. Przy 250 klientach aplikacja w *Go* obsłużyła również od 7.0 do 8.0 tysięcy żądań, serwer *Jetty* 3.0 tysiące, a wydajność serwera *Tomcat* oscylowała na poziomie 2.5 tysiąca żądań.

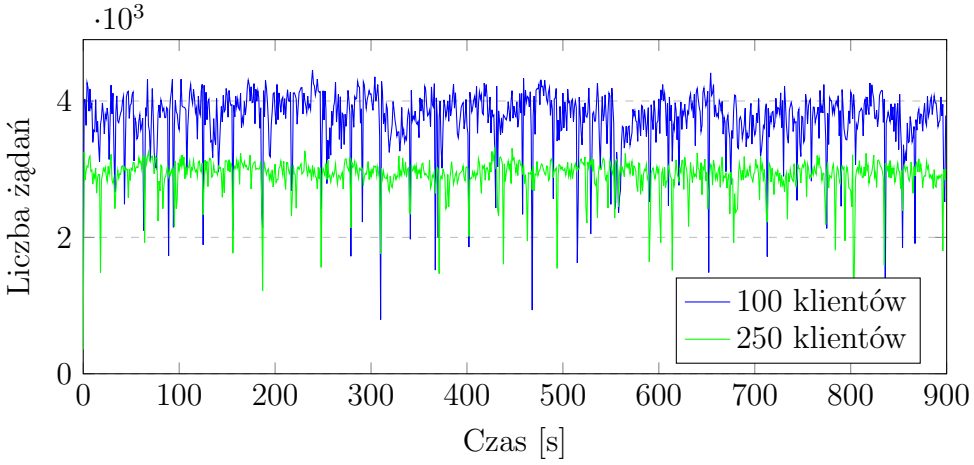
Z wykresów rozkładu czasów odpowiedzi (rys. 6.20, 6.22, 6.24) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 27.30 milisekund, dla serwera *Jetty* 25.63 milisekund i 13.94 milisekund dla aplikacji w *Go*. Najdłużej trwające żądania trwały poniżej 110 milisekund w przypadku serwera *Tomcat*, powyżej 60 milisekund przy *Jetty* i około 35 milisekund w aplikacji w *Go*. Przy 250 klientach średnie czas odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły 86.85 i 74.48 milisekund, a w aplikacji w *Go* tylko 28.06 milisekundy. Najdłużej trwające żądania trwały poniżej 180 milisekund w przypadku serwera *Tomcat*, 150 milisekund w przypadku serwera *Jetty* i 85 milisekund przy aplikacji w *Go*.



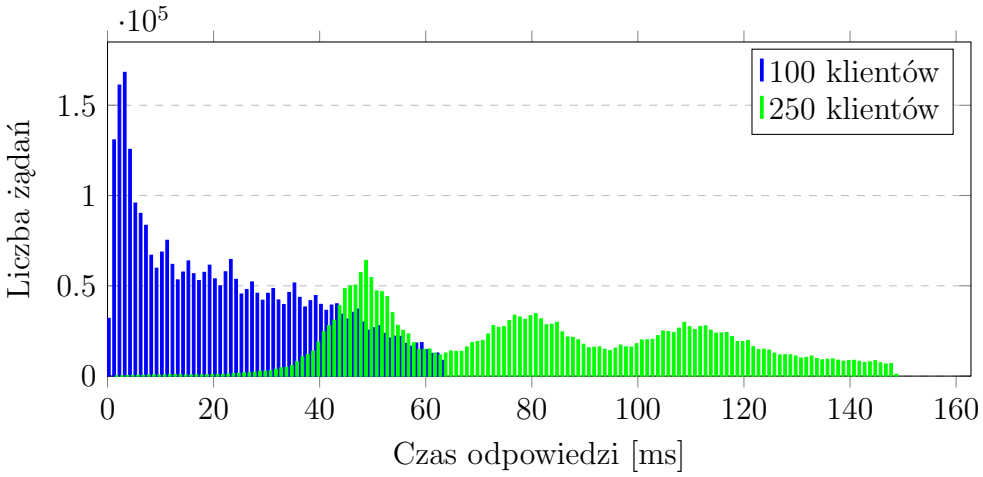
Rysunek 6.19: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



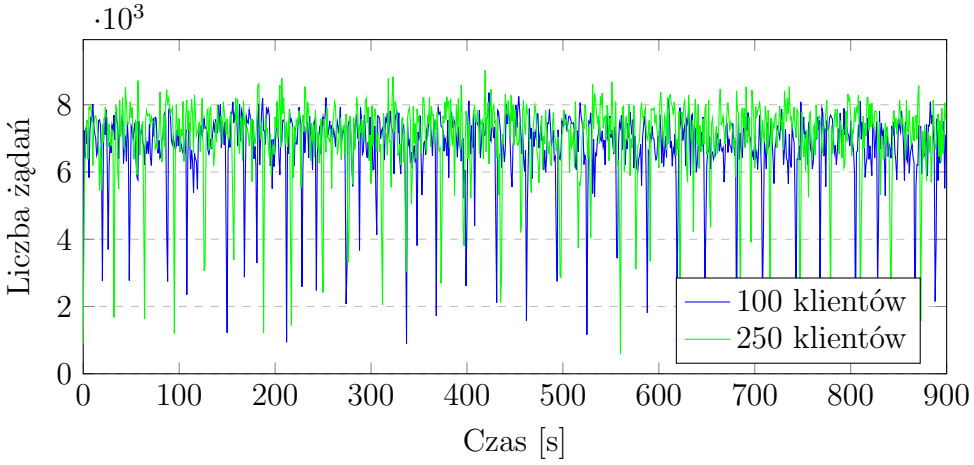
Rysunek 6.20: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



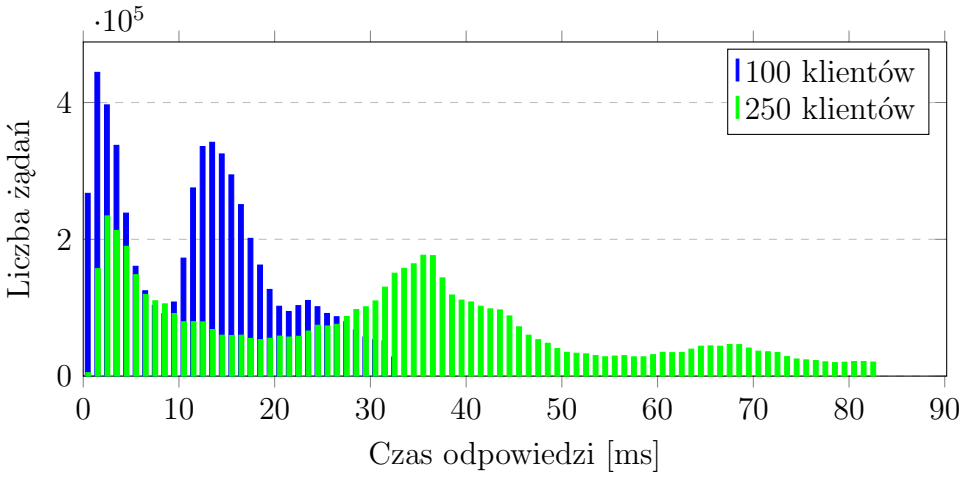
Rysunek 6.21: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



Rysunek 6.22: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



Rysunek 6.23: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



Rysunek 6.24: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle

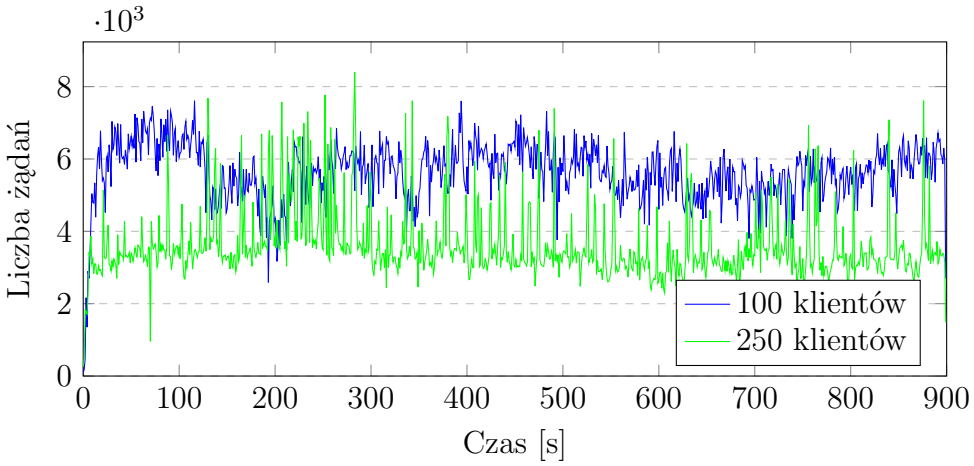
6.2 Testy z bazą wypełnioną danymi początkowymi

6.2.1 Testy wydajności walidacji API

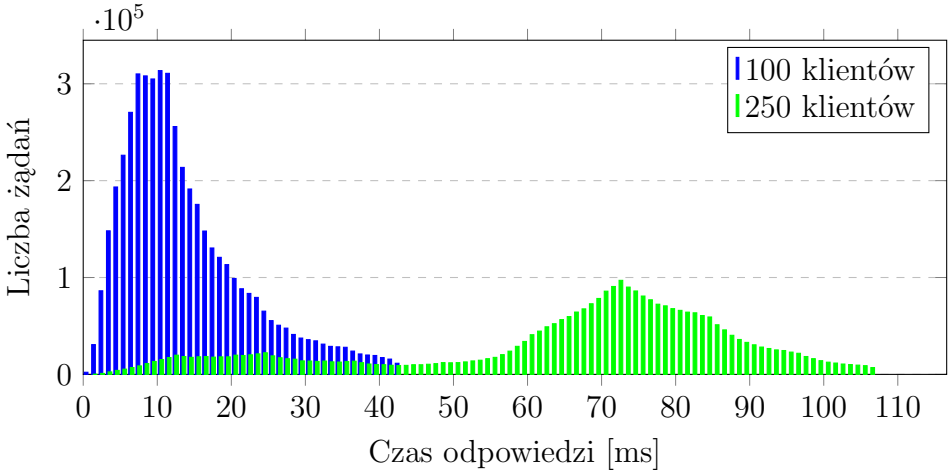
Diagramy zawierające wyniki wydajności walidacji API zamieszczono na rysunkach 6.25 - 6.30

Z rozkładów ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. rys. 6.25, 6.27, 6.29) wynika, że przy 100 klientach aplikacja w *Go* obsługiwała od 9.0 do 10.0 tysięcy żądań. Wydajność serwera *Jetty* oscylowała w przedziale od 8.0 do 10.0 tysięcy żądań. Serwer *Tomcat* obsługiwał od 5.0 do 7.0 tysięcy żądań. Przy 250 klientach aplikacja w *Go* obsługiwała również od 8.0 do 10.0 tysięcy żądań, serwer *Jetty* od 4.0 do 8.5 tysiąca żądań przy bardzo dużych wahaniach przepustowości, a wydajność serwera *Tomcat* oscylowała w przedziale od 3.0 do 6.0 tysiąca obsłużonych żądań.

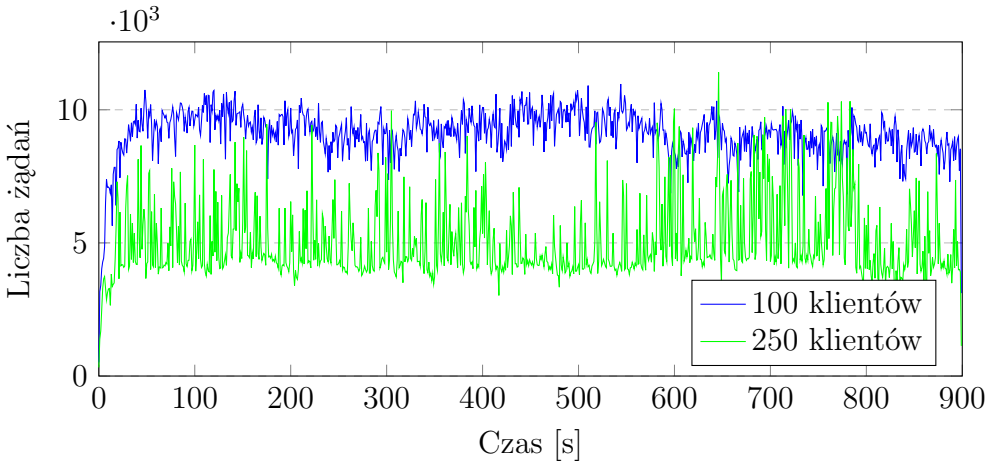
Z wykresów rozkładu czasów odpowiedzi (rys. 6.26, 6.28, 6.30) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 16.72 milisekund, dla serwera *Jetty* 10.25 milisekund i 10.24 milisekund dla aplikacji w *Go*. Najdłużej trwające żądania trwały poniżej 45 milisekund przy serwerze *Tomcat*, 30 milisekund przy *Jetty* i 18 milisekund w aplikacji w *Go*. Przy 250 klientach średnie czasy odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły odpowiednio 67.28 i 47.00 milisekundy, a w aplikacji w *Go* 25.33 milisekundy. Najdłużej trwające żądania trwały poniżej 110 milisekund w przypadku serwera *Tomcat*, 83 milisekundy w przypadku serwera *Jetty* i 55 milisekund przy aplikacji w *Go*.



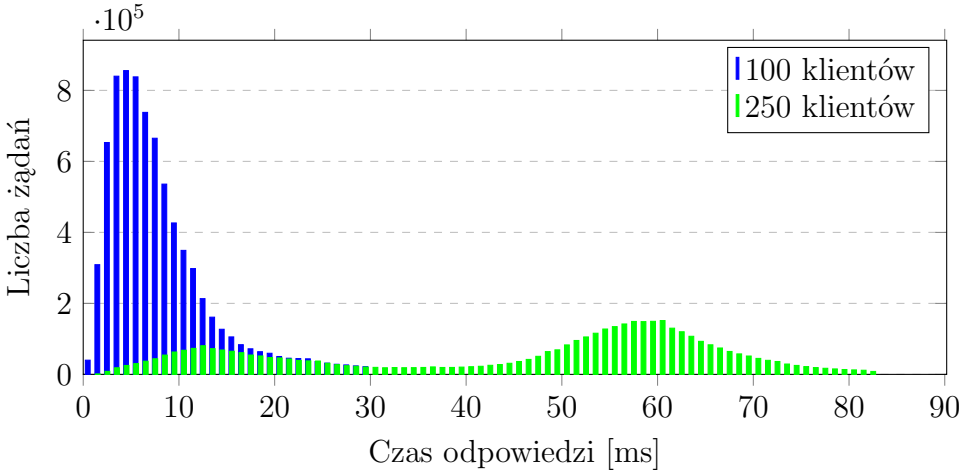
Rysunek 6.25: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia klucza API



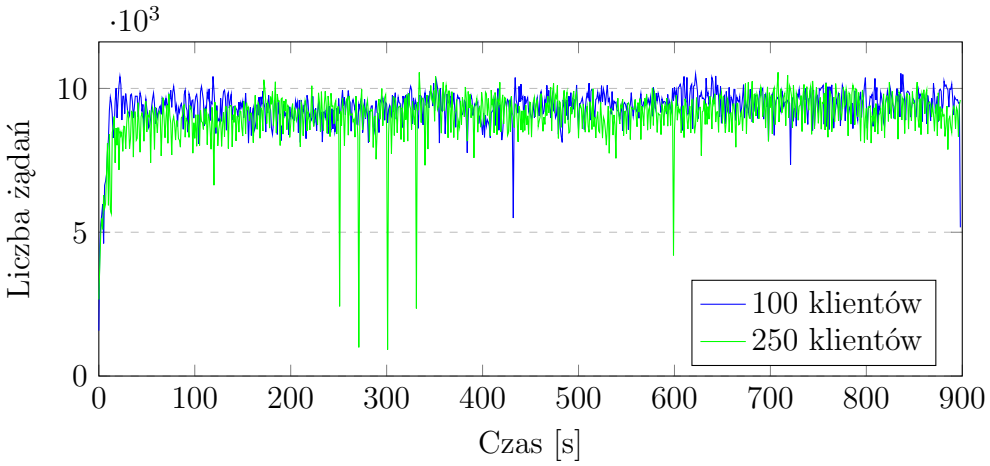
Rysunek 6.26: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia klucza API



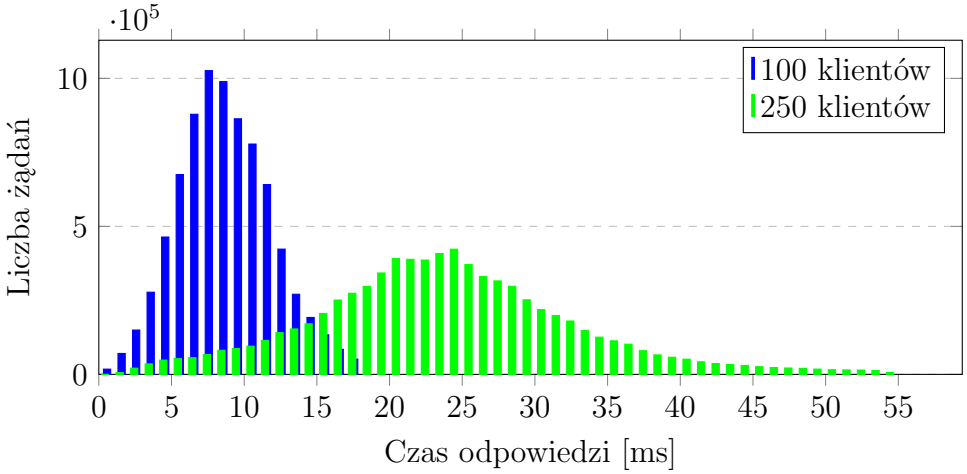
Rysunek 6.27: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia klucza API



Rysunek 6.28: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia klucza API



Rysunek 6.29: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia klucza API



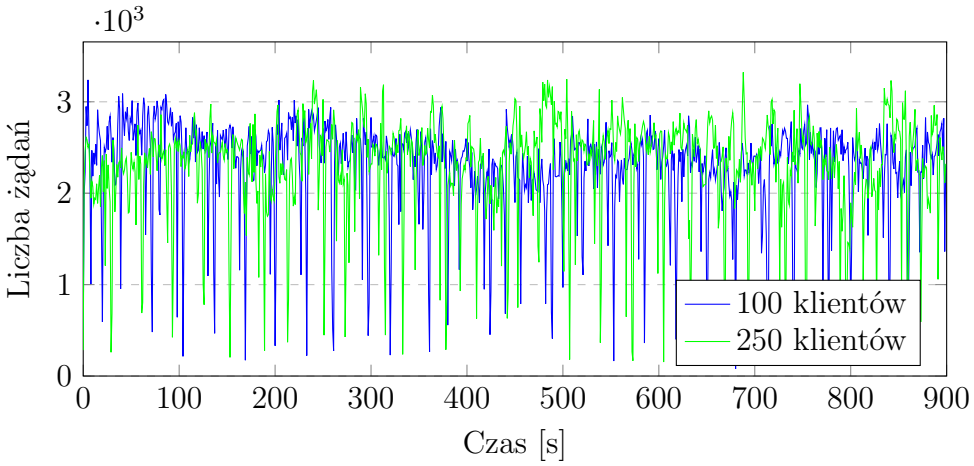
Rysunek 6.30: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia klucza API

6.2.2 Test wydajności walidacji istnienia obiektów Cache

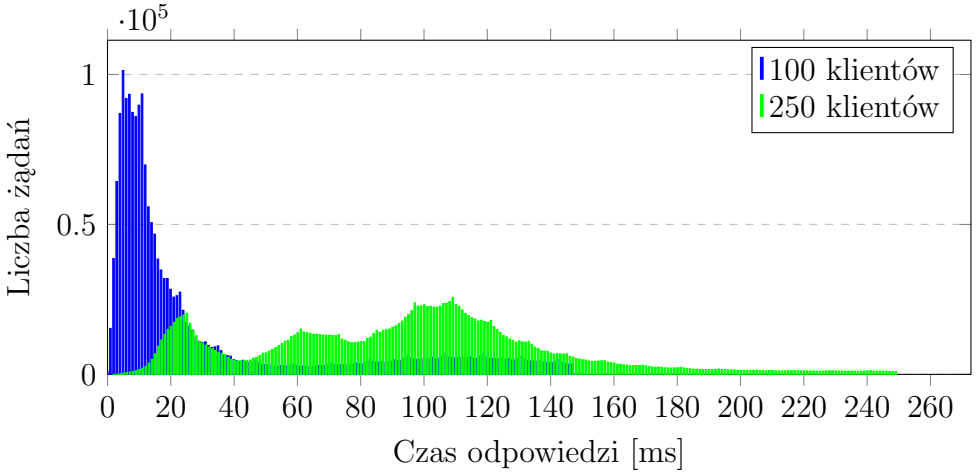
Wyniki testów wydajności walidacji istnienia obiektów Cache przedstawiają wykresy na rysunkach 6.31 - 6.36.

Z wykresów przedstawiających rozkład ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. 6.31, 6.33, 6.35) wynika, że przy 100 klientach przepustowość aplikacji w *Go* kształtowała się w przedziale od 5.0 do 6.0 tysięcy żądań. Należy zaznaczyć, że przepustowość ta została osiągniętą po upływie 3 minut od rozpoczęcia testu. Wydajność serwera *Jetty* oscylowała w przedziale od 4.0 do 5.0 tysięcy obsłużonych żądań, a serwera *Tomcat* od 2.0 do 3.0 tysięcy obsłużonych żądań. Przy 250 klientach aplikacja w *Go* obsłużyła od 5.0 do 6.0 tysięcy żądań również po upływie 3 minut od rozpoczęcia testu, serwer *Jetty* około 3.0 tysięcy żądań, a serwer *Tomcat* od 2.0 do 3.0 tysiąca żądań.

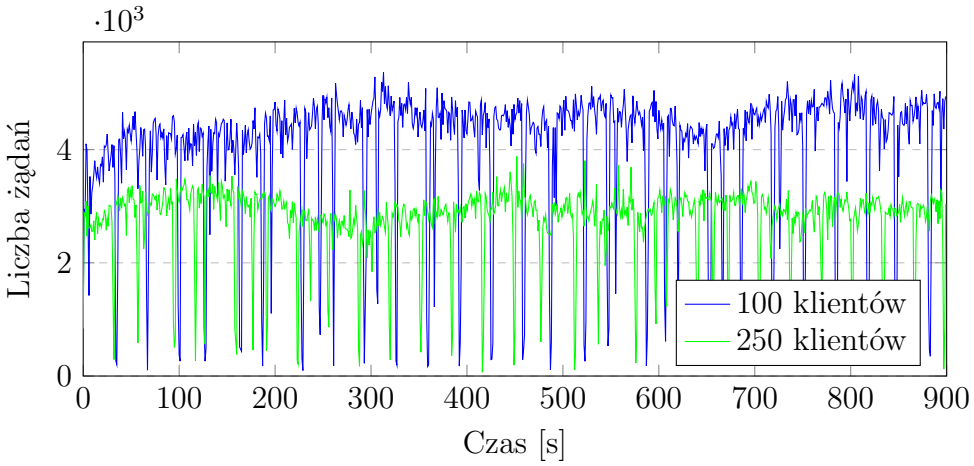
Z wykresów rozkładu czasów odpowiedzi (rys. 6.32, 6.34, 6.36) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 38.00 milisekund, dla serwera *Jetty* 22.15 milisekund i 19.75 milisekund dla aplikacji w *Go*. Najdłużej trwające żądania trwały 150 milisekund przy serwerze *Tomcat*, około 40 milisekund na serwerze *Jetty* i tyle samo w aplikacji w *Go*. Przy 250 klientach średnie czasy odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły 86.73 i 82.37 milisekundy, a w aplikacji w *Go* 40.66 milisekundy. Najdłużej trwające żądania trwały 250 milisekund w przypadku serwera *Tomcat*, 120 milisekund w przypadku serwera *Jetty* i poniżej 100 milisekund przy aplikacji w *Go*.



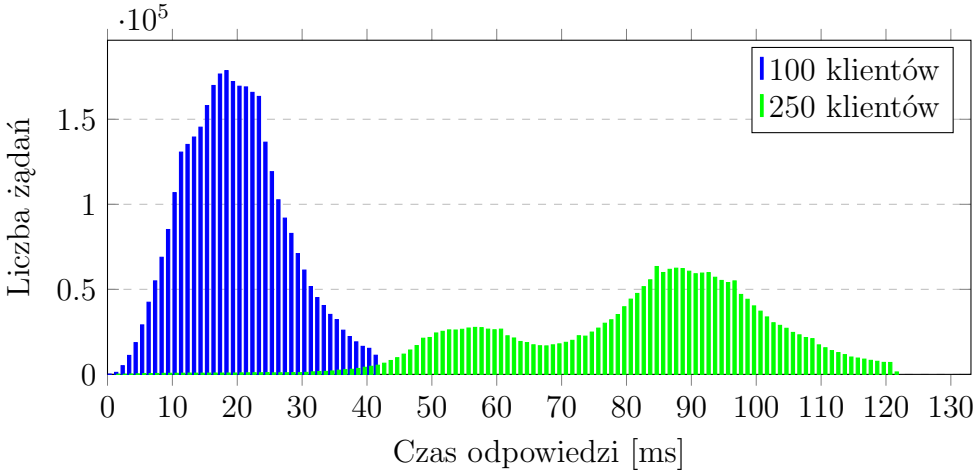
Rysunek 6.31: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia obiektu Cache



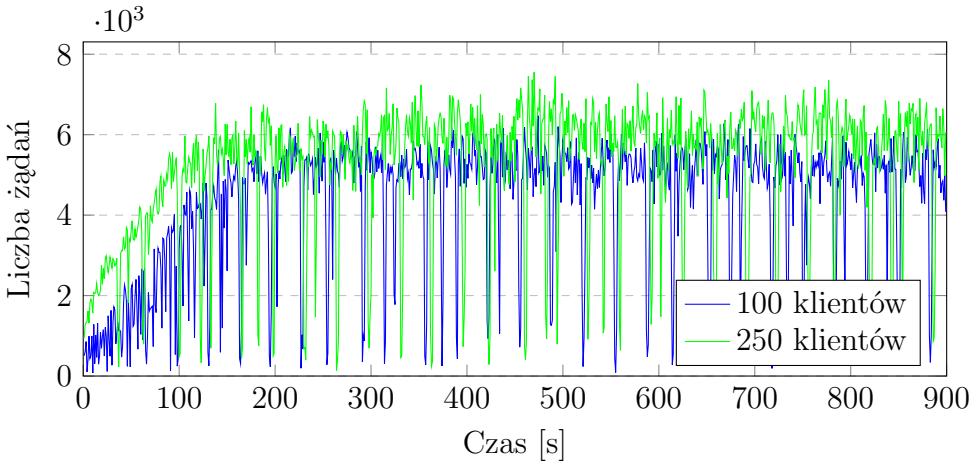
Rysunek 6.32: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia obiektu Cache



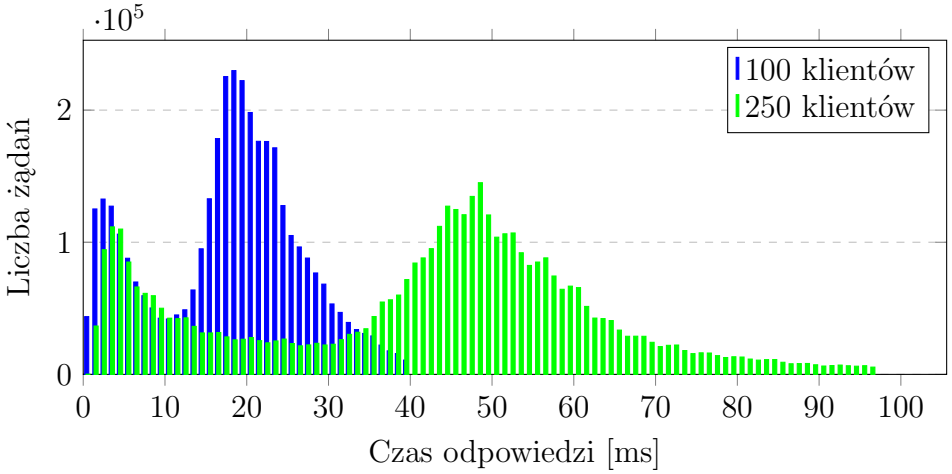
Rysunek 6.33: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia obiektu Cache



Rysunek 6.34: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia obiektu Cache



Rysunek 6.35: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu walidacji istnienia obiektu Cache



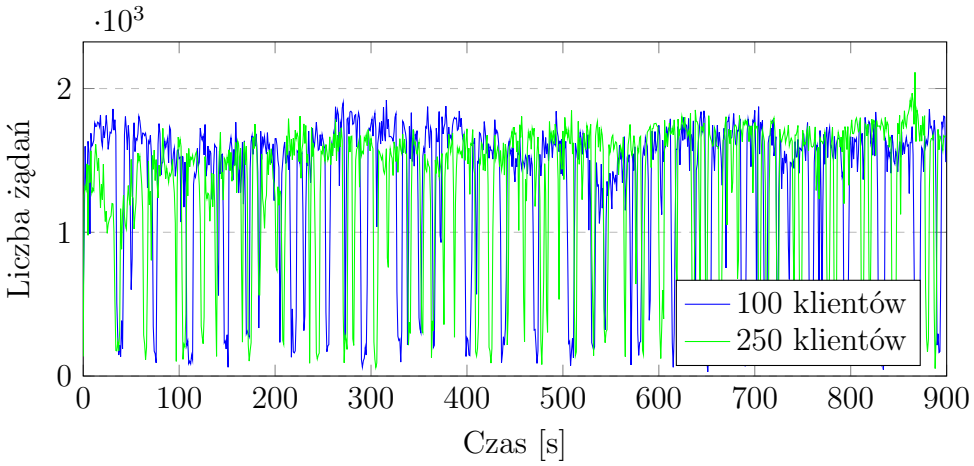
Rysunek 6.36: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu walidacji istnienia obiektu Cache

6.2.3 Test wydajności operacji CRUD

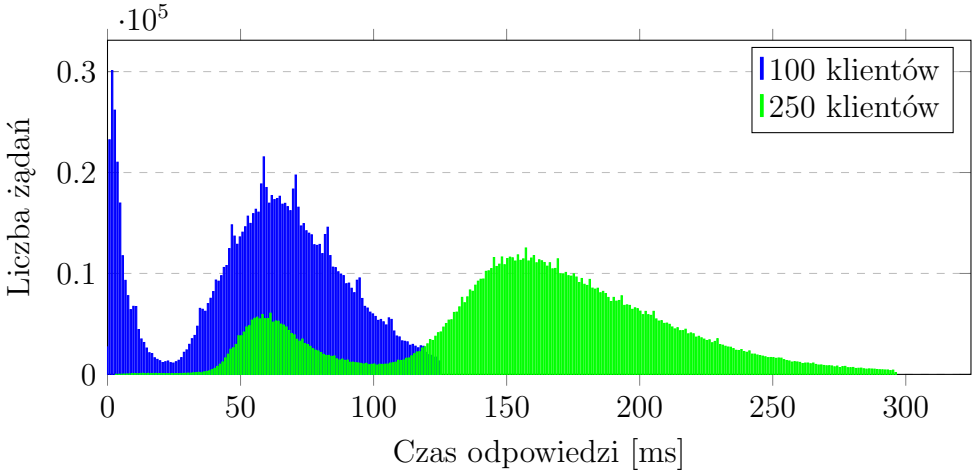
Wyniki testów wydajności walidacji istnienia operacji CRUD przedstawiają wykresy na rysunkach 6.37 - 6.42.

Z wykresów przedstawiających rozkład ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. 6.37, 6.39, 6.41) wynika, że przy 100 i 250 klientach poszczególne aplikacje zachowywały się porównywalnie. Przepustowość aplikacji w *Go* wahała się na od 0.5 do 3.0 tysięcy obsłużonych żądań, serwer *Jetty* obsługiwał od 0.5 do ponad 2.0 tysięcy żądań, a serwer *Tomcat* od 0.5 do około 1.5 tysiąca żądań. Dla każdej z aplikacji występowały spadki przepustowości nawet poniżej 500 żądań w ciągu sekundy.

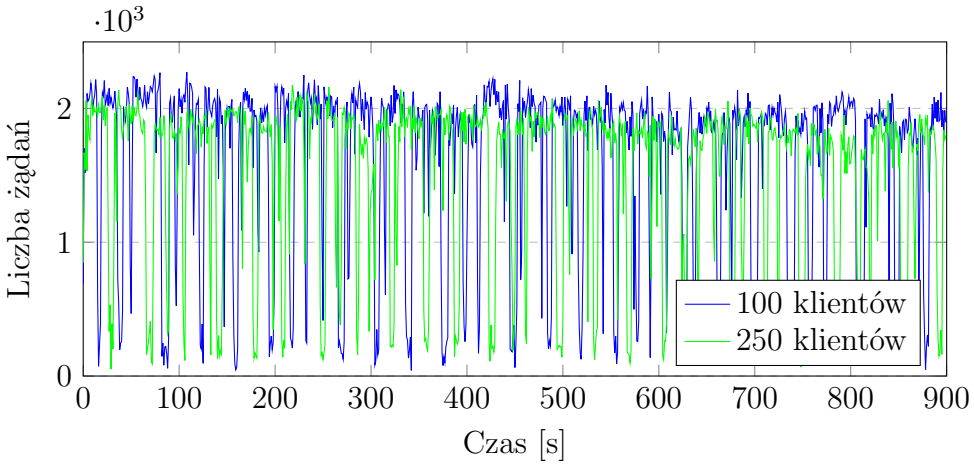
Z wykresów rozkładu czasów odpowiedzi (rys. 6.38, 6.40, 6.42) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 61.57 milisekund, dla serwera *Jetty* 50.04 milisekundy i 32.26 milisekund dla aplikacji w *Go*. Najdłużej trwające żądania trwały poniżej 130 milisekund przy serwerze *Tomcat*, około 100 milisekund przy serwerze *Jetty* i poniżej 70 milisekund w aplikacji w *Go*. Przy 250 klientach średnie czasy odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły odpowiednio 128.07 i 101.60 milisekund, a w aplikacji w *Go* tylko 48.96 milisekundy. Najdłużej trwające żądania trwały około 300 milisekund w przypadku serwera *Tomcat*, 240 milisekund w przypadku serwera *Jetty* i poniżej 170 milisekund przy aplikacji w *Go*.



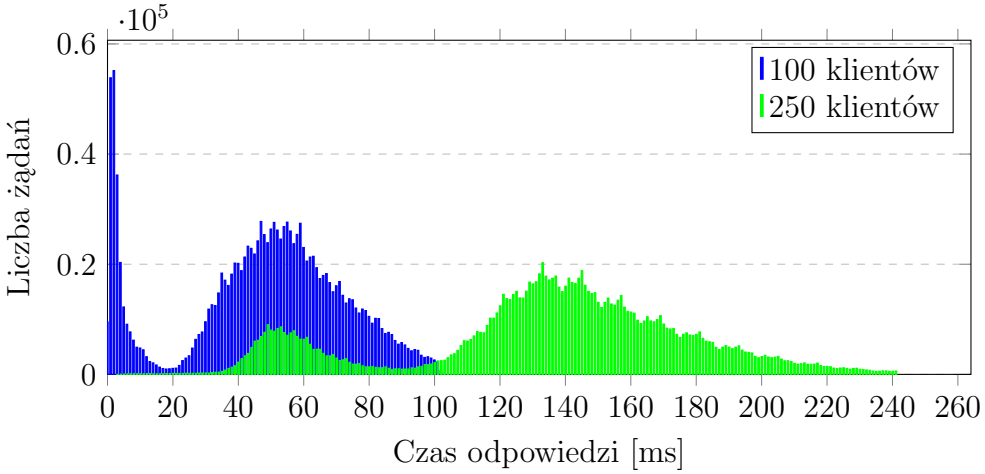
Rysunek 6.37: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu operacji CRUD



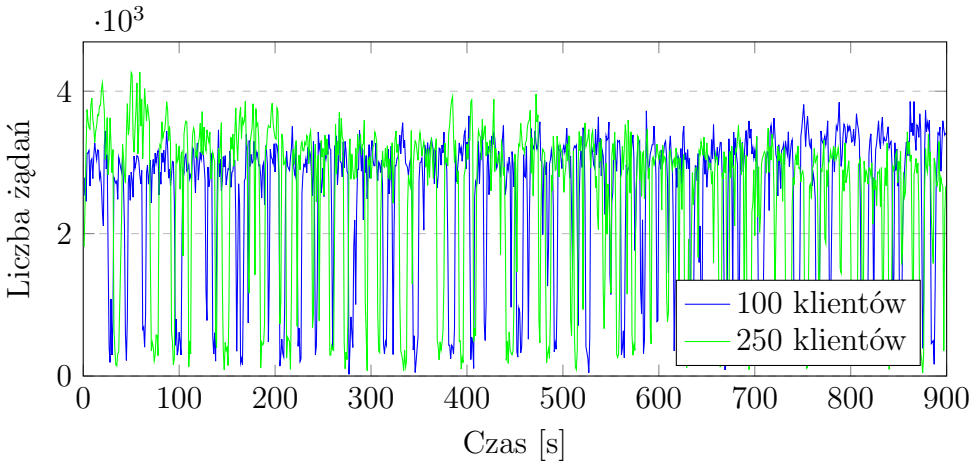
Rysunek 6.38: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu operacji CRUD



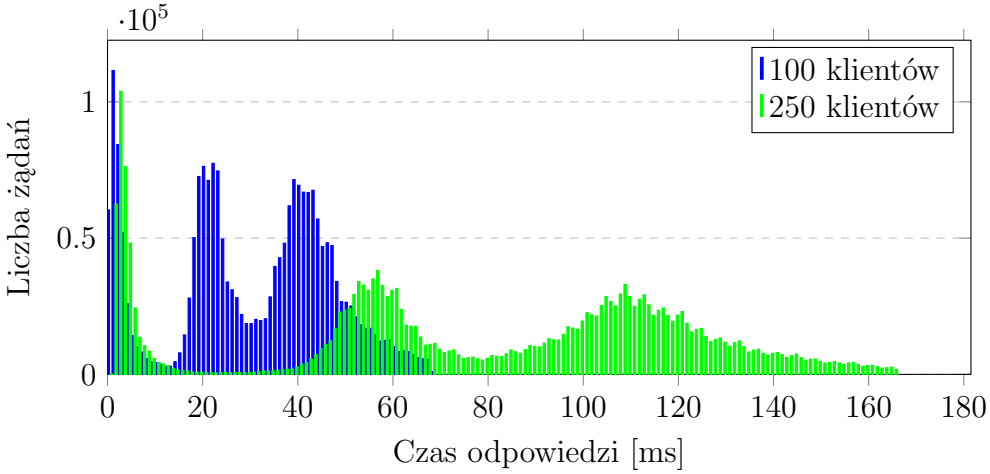
Rysunek 6.39: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu operacji CRUD



Rysunek 6.40: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu operacji CRUD



Rysunek 6.41: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu operacji CRUD



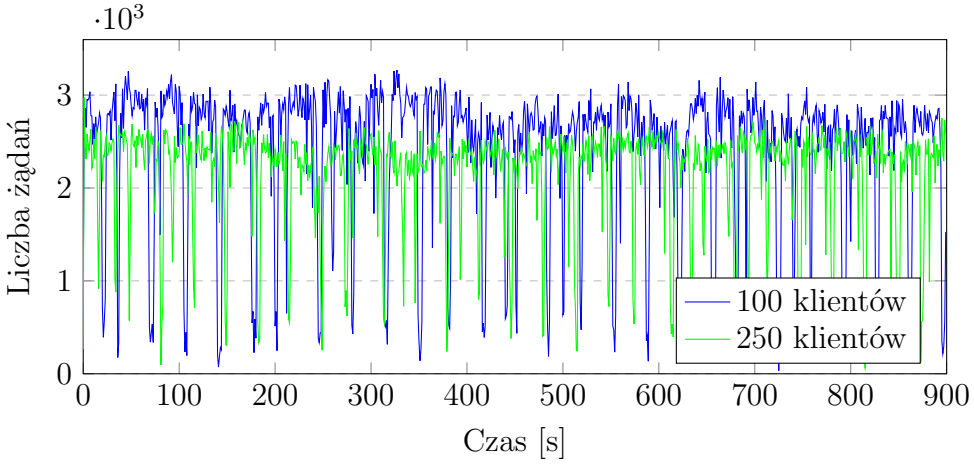
Rysunek 6.42: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu operacji CRUD

6.2.4 Test wydajności walidacji API, obiektów Cache oraz operacji CRUD równolegle

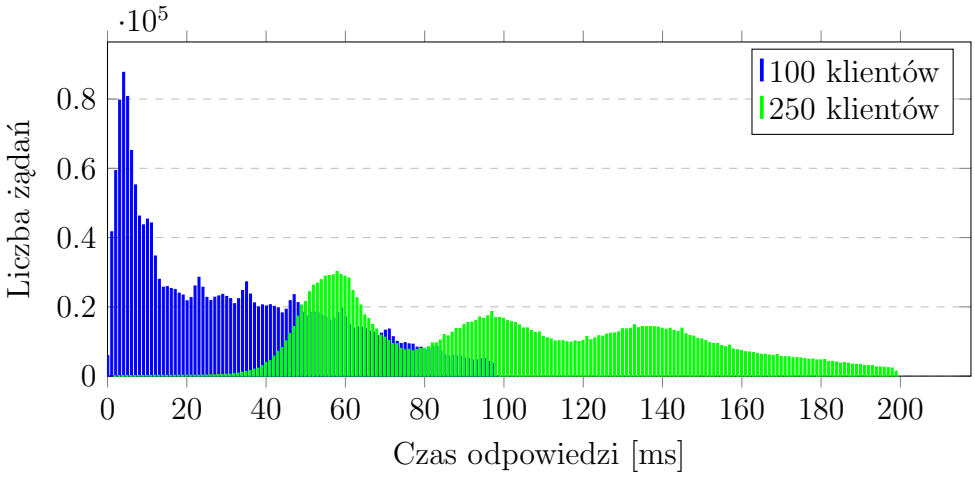
Wyniki testów wydajności walidacji API, obiektów Cache oraz operacji CRUD równolegle przedstawiają wykresy na rysunkach 6.43 - 6.48.

Z wykresów przedstawiających rozkład ilości żądań obsłużonych przez poszczególne aplikacje w ciągu sekundy (rys. 6.43, 6.45, 6.47) wynika, że przy 100 klientach przepustowość aplikacja w *Go* kształtowała się w przedziale od 4.0 do 6.0 tysięcy obsłużonych żądań, wydajność serwera *Jetty* oscylowała w przedziale od 3.0 do 3.5 tysiąca żądań, a serwera *Tomcat* obsłużył od 2.5 do 3.0 tysięcy żądań. Przy 250 klientach aplikacja w *Go* obsłużyła również od 4.0 do 6.0 tysięcy żądań, a przepustowość serwerów *Jetty* i *Tomcat* oscylowała w przedziale od 2.0 do 3.0 tysięcy żądań. Dla każdej z aplikacji występowały spadki przepustowości nawet poniżej 500 żądań w ciągu sekundy

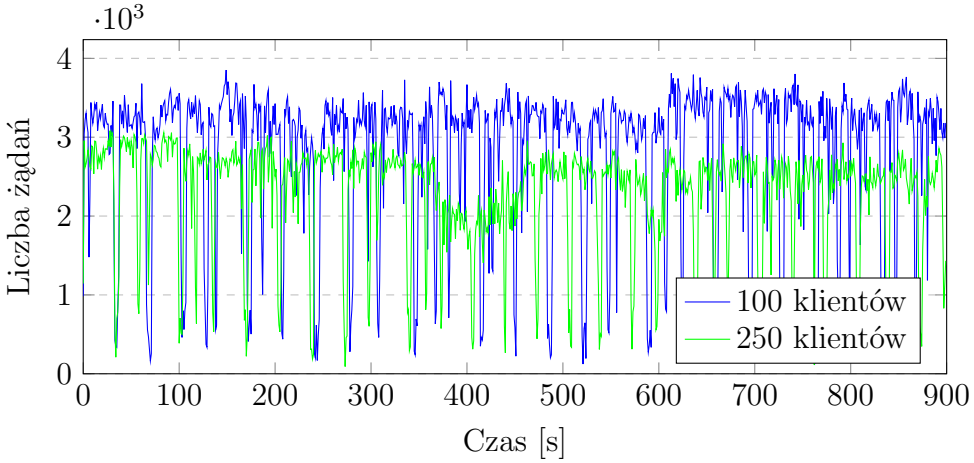
Z wykresów rozkładu czasów odpowiedzi (rys. 6.44, 6.46, 6.48) wynika, że przy 100 klientach średnie czasy odpowiedzi wynosiły: dla serwera *Tomcat* 35.82 milisekundy, dla serwera *Jetty* 30.10 milisekund i 19.40 milisekund dla aplikacji w *Go*. Najdłużej trwające żądania trwały poniżej 100 milisekund przy serwerze *Tomcat*, 80 milisekund przy serwerze *Jetty* i poniżej 50 milisekund w aplikacji w *Go*. Przy 250 klientach średnie czasy odpowiedzi aplikacji uruchamianych na serwerach *Tomcat* i *Jetty* wynosiły odpowiednio 96.96 i 82.34 milisekundy, a w aplikacji w *Go* tylko 38.33 milisekundy. Najdłużej trwające żądania trwały poniżej 200 milisekund w przypadku serwerów *Tomcat* i *Jetty* oraz 130 milisekund w teście aplikacji w *Go*.



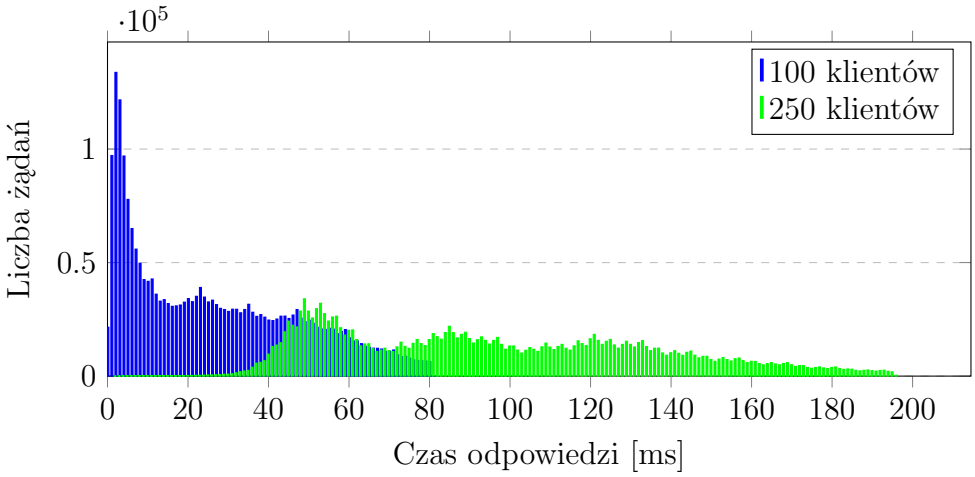
Rysunek 6.43: Tomcat 8 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



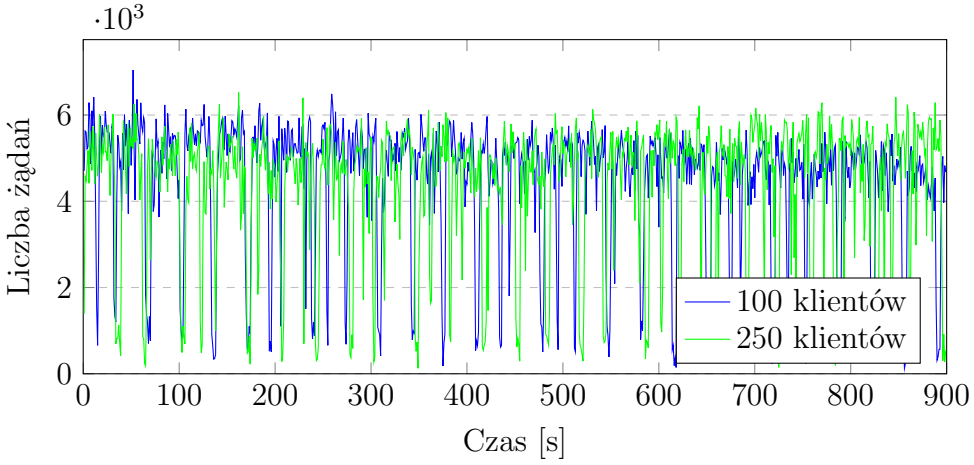
Rysunek 6.44: Tomcat 8 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



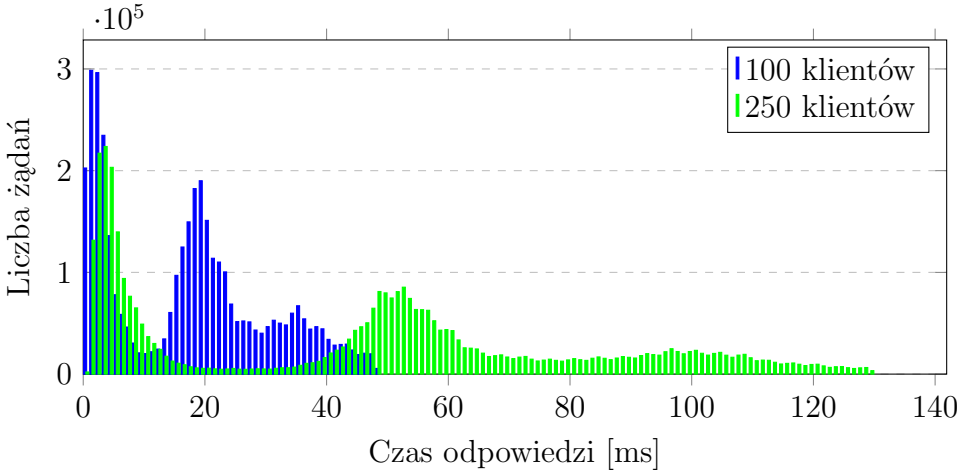
Rysunek 6.45: Jetty 9 - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



Rysunek 6.46: Jetty 9 - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



Rysunek 6.47: Go - liczba żądań obsłużonych przez aplikację w ciągu sekundy podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle



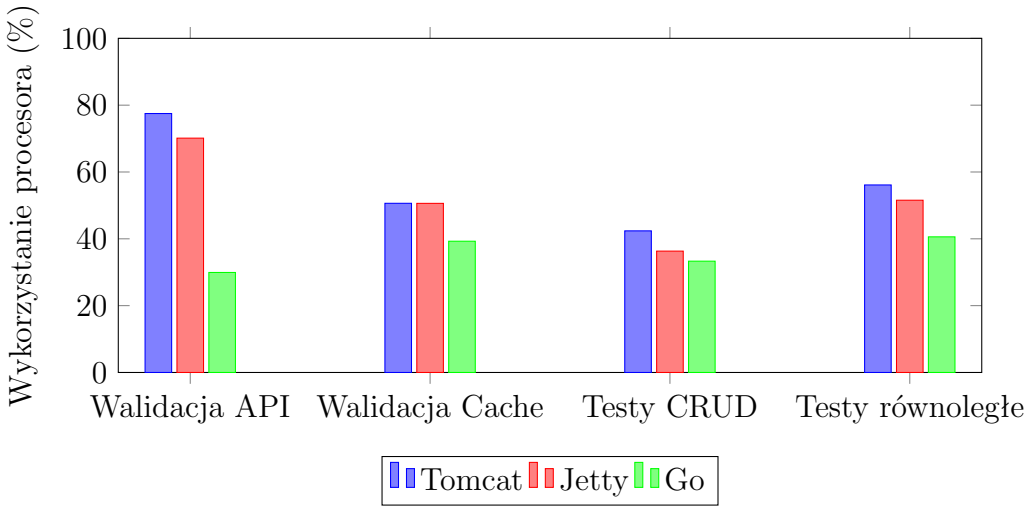
Rysunek 6.48: Go - rozkład czasów odpowiedzi aplikacji (95% odpowiedzi) podczas testu: walidacji istnienia klucza API, walidacji istnienia, operacji CRUD równolegle

6.3 Obciążenie serwerów podczas testów - pusta baza danych

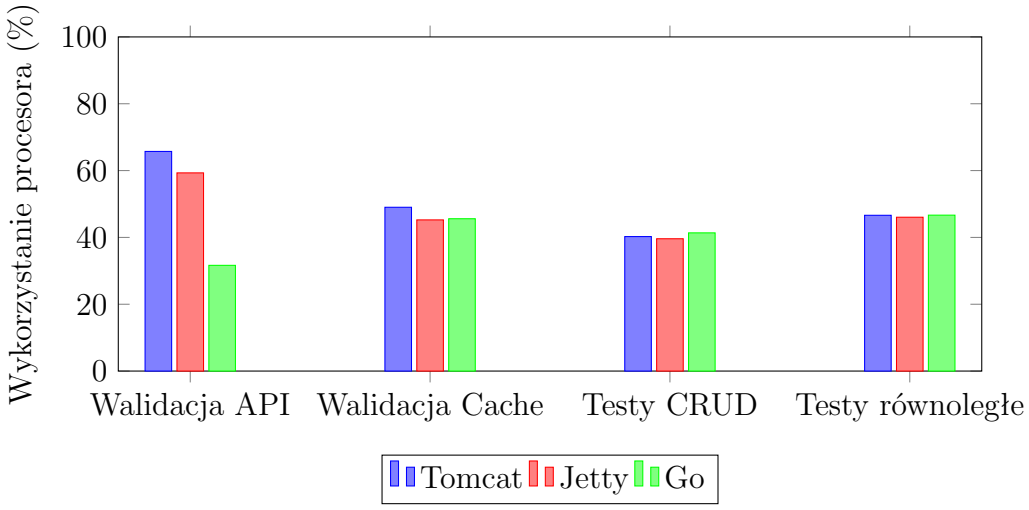
Średnie wykorzystanie procesora maszyny, gdzie uruchomiona była testowana aplikacja podczas testu z pustą baz danych przedstawiają diagramy na rysunkach 6.49 i 6.50. Dane zbierane były przy pomocy programu *dstat*.

Z diagramów wynika, że najmniej wykorzystywany był procesor podczas testów aplikacji w *Go*. Wykorzystywała ona procesor średnio od 29% do 47%. Średnie wykorzystanie procesora podczas testów na serwerach *Tomcat* i *Jetty* było znacznie wyższe. Dla serwera *Tomcat* wartość średnia wyniosła od 40% do 78%, a dla *Jetty* od 36% do 71%, w zależości od grupy testów. Podczas testów z 250 klientami średnie wykorzystanie procesora w większości przypadków było niższe. Największą różnicę zaobserwowano w grupie testów walidujących istnienie kluczy *API*

Wykorzystanie pamięci *RAM* na maszynie nie różniło się znacząco w poszczególnych grupach testów. Podczas testów na serwerze *Tomcat* maszyna testowa miała zajęte średnio 1950 MB podczas testów przy 100 klientach i 2000 MB przy 250 klientach. Podczas testów na serwerze *Jetty* maszyna testowa miała zajęte średnio 1900 MB przy 100 klientach i 1760 MB przy 250 klientach. Podczas testu aplikacji w *Go* wykorzystanie pamięci *RAM* na serwerze było porównywalne przy 100 i 250 klientach i wynosiło mniej niż 250 MB.



Rysunek 6.49: Wykorzystanie procesora przez aplikacje podczas testów z pustą bazą danych - 100 klientów



Rysunek 6.50: Wykorzystanie procesora przez aplikacje podczas testów z pustą bazą danych - 250 klientów

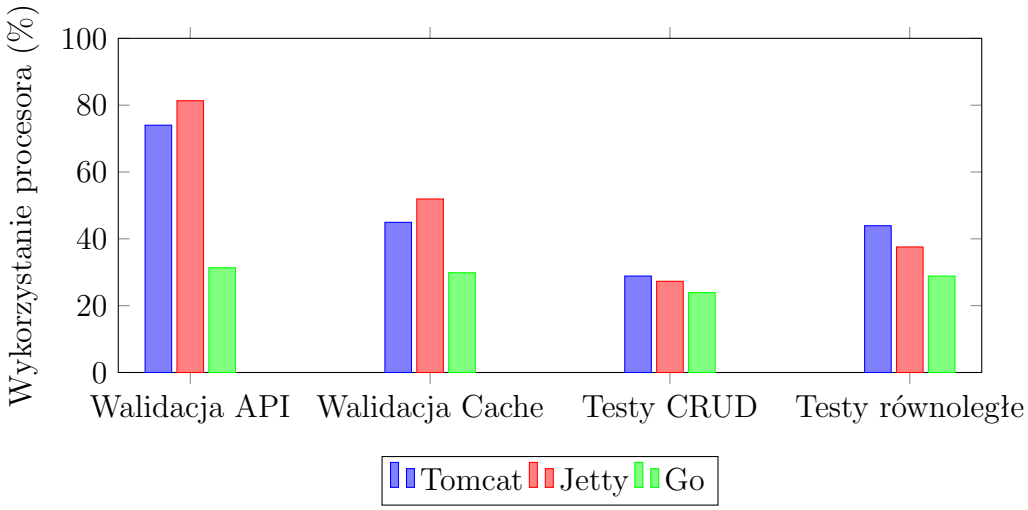
6.4 Obciążenie serwerów podczas testów - baza danych wypełniona danymi początkowymi

Średnie wykorzystanie procesora maszyny, gdzie uruchomiona była testowana aplikacja podczas testu z bazą danych wypełnioną danymi początko-

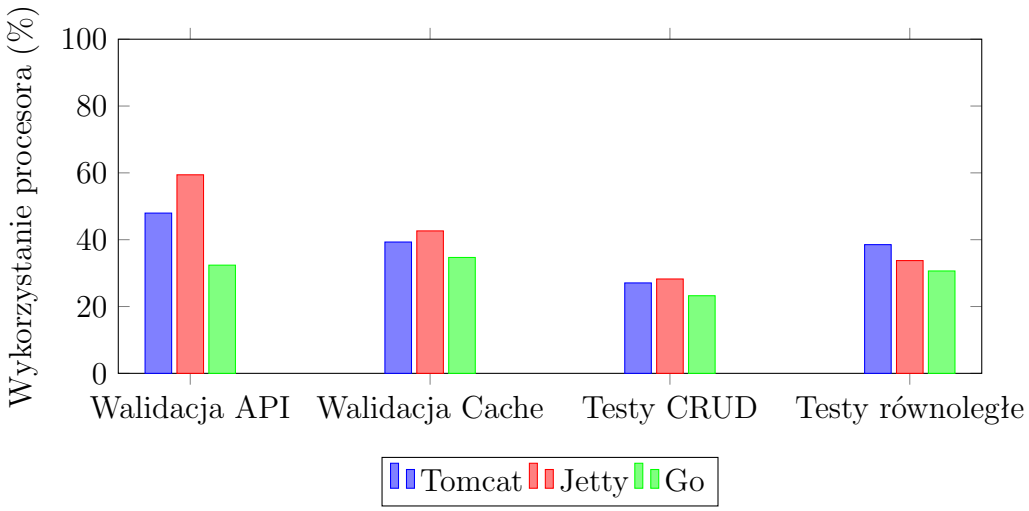
wymi przedstawiają diagramy na rysunkach 6.51 i 6.52.

Z diagramów wynika, że procesor był najmniej wykorzystywany podczas testów aplikacji w *Go*. Średnio aplikacja ta wykorzystywała od 23% do 34%. zasobów procesora. Znacznie wyższe średnie wykorzystanie procesora było podczas testów na serwerach *Tomcat* i *Jetty*. Serwer *Jetty* wykorzystywał procesor w przedziale od 28% do 81%, a *Tomcat* od 27% do 74%.

Wykorzystanie pamięci *RAM* na maszynie w poszczególnych grupach testów było porównywalne. Podczas testu aplikacji w języku *Go* maszyna miała zajęte średnio 250 MB. Podczas testów na serwerze *Tomcat* maszyna miała zajęte średnio 1830 MB przy 100 klientach i 2030 MB przy 250 klientach. Natomiast podczas testów na serwerze *Jetty* przy 100 i 250 klientach wykorzystanie pamięci *RAM* kształtowało się na poziomie 2010 MB.



Rysunek 6.51: Wykorzystanie procesora przez aplikacje podczas testów z bazą danych wypełnioną danymi początkowymi - 100 klientów



Rysunek 6.52: Wykorzystanie procesora przez aplikacje podczas testów z bazą danych wypełnioną danymi początkowymi - 250 klientów

6.5 Interpretacja wyników

Z przeprowadzonych testów wynika, że aplikacja w języku *Go* osiągała najlepsze wartości przepustowości i czasów odpowiedzi. Niezależnie od liczby klientów osiągane wyniki nie różniły się znacząco między sobą. Przy większej liczbie klientów liczba żądań obsługiwanych w ciągu sekundy nie była dużo niższa, mimo że czasy odpowiedzi żądań były wyraźnie dłuższe.

Aplikacja w języku *Java* osiągała wyraźnie niższe rezultaty przepustowości, szczególnie przy większej liczbie obsługiwanych klientów. Gdy z aplikacji korzystało 250 klientów obserwowano mniejszą liczbę przetworzonych żądań w ciągu sekundy. Wyraźne były też chwilowe spadki przepustowości. Czasy odpowiedzi były również dłuższe od aplikacji w *Go*. Analizując wyniki testowanych serwerów, na których była uruchomiona aplikacja *Java*, lepsze rezultaty zaobserwowano na serwerze *Jetty*. Zarówno wyniki przepustowości, jak i czasy odpowiedzi w większości przypadków testowych wykazywały przewagę nad serwerem *Tomcat*.

Analiza wykorzystania zasobów maszyny przez poszczególne aplikacje pokazuje, że aplikacja w *Go* wymagała znacznie mniejszych zasobów procesora i pamięci *RAM* od aplikacji w języku *Java*, a serwer *Jetty* wymagał nieznacznie większych zasobów od serwera *Tomcat*.

Grupą testów, w której zaobserwowano największe wahania wyników były testy operacji *CRUD*. Wykorzystywały one najbardziej komunikację z bazą danych *MongoDB*. Najlepsze rezultaty zanotowano podczas testów walidacji *API* oraz obiektów *Cache*, gdzie komunikacja z bazą sprowadzała się do sprawdzenia, czy poszukiwany dokument istnieje. Walidacja *API* okazała się najbardziej wykorzystywać zasoby procesora podczas testów aplikacji w języku *Java*. W testach tych widoczne też było mniejsze wykorzystanie zasobów procesora na maszynie testowej.

Rozdział 7

Podsumowanie

Przedmiotem pracy było porównanie wydajności serwisów *RESTful* w wybranych platformach programowania. Na potrzeby badań opracowano i zaimplementowano dwie identyczne aplikacje w różnych językach programowania: *Java* i *Go*. Na aplikacjach przeprowadzono testy wydajnościowe, na podstawie których oceniono ich wydajność.

Stworzone aplikacje były usługami typu *RESTful*, z którymi komunikacja odbywała się protokołem *HTTP*. Funkcje aplikacji pozwalały na zapisywanie, modyfikowanie i odczytywanie danych zapisanych przez użytkowników. Użytkownicy autoryzowali się specjalnie dla nich wygenerowanym, unikalnym kluczem. Aplikacje do działania wykorzystywały wydajną bazę danych *MongoDB*.

W celu przeprowadzenia testów przygotowane zostało specjalne środowisko składające się z trzech maszyn wirtualnych. Najwydajniejsza z maszyn wykorzystywana była do uruchamiania przygotowanych aplikacji, druga jako baza danych, a trzecia służyła do uruchamiania testów wydajnościowych.

Składnie języków użytych w aplikacjach zasadniczo różnią się od siebie. Jedną z nich jest fakt, że język *Java* jest językiem obiektowym, podczas gdy *Go* pozwala na zastosowanie niewielu technik programowania obiektowego. Również obsługa błędów w obu językach jest inna. *Java* wykorzystuje do tego mechanizm rzucania i łapania wyjątków, a w *Go* obsługa błędnych sytuacji opiera się na zwracaniu z funkcji i metod obiektów typu *Error*.

Ze względu na zastosowaną technologię (*Java EE*) aplikacja w języku *Java*

wymagała zastosowania kontenerów aplikacji. W celu porównania, testy przeprowadzane były z wykorzystaniem dwóch najpopularniejszych kontenerów: *Tomcat* i *Jetty*.

Na podstawie przeprowadzonych testów można jednoznacznie stwierdzić, że aplikacja w języku *Go* osiągała najlepsze rezultaty. Wyniki przepustowości tej aplikacji były wyższe od aplikacji w języku *Java* dla każdego testowego przypadku i nie zależały od obciążenia. Czasy odpowiedzi były również zdecydowanie krótsze, a ich rozłożenie było bardziej równomierne.

Aplikacja w języku *Java* uruchomiona na serwerach *Tomcat* i *Jetty* charakteryzowała się dużym zróżnicowaniem przepustowości. Czasy odpowiedzi również były dłuższe, a ich rozkład nierównomierny. Wyniki uzyskane na serwerze *Jetty* w większości przypadków były lepsze od serwera *Tomcat*.

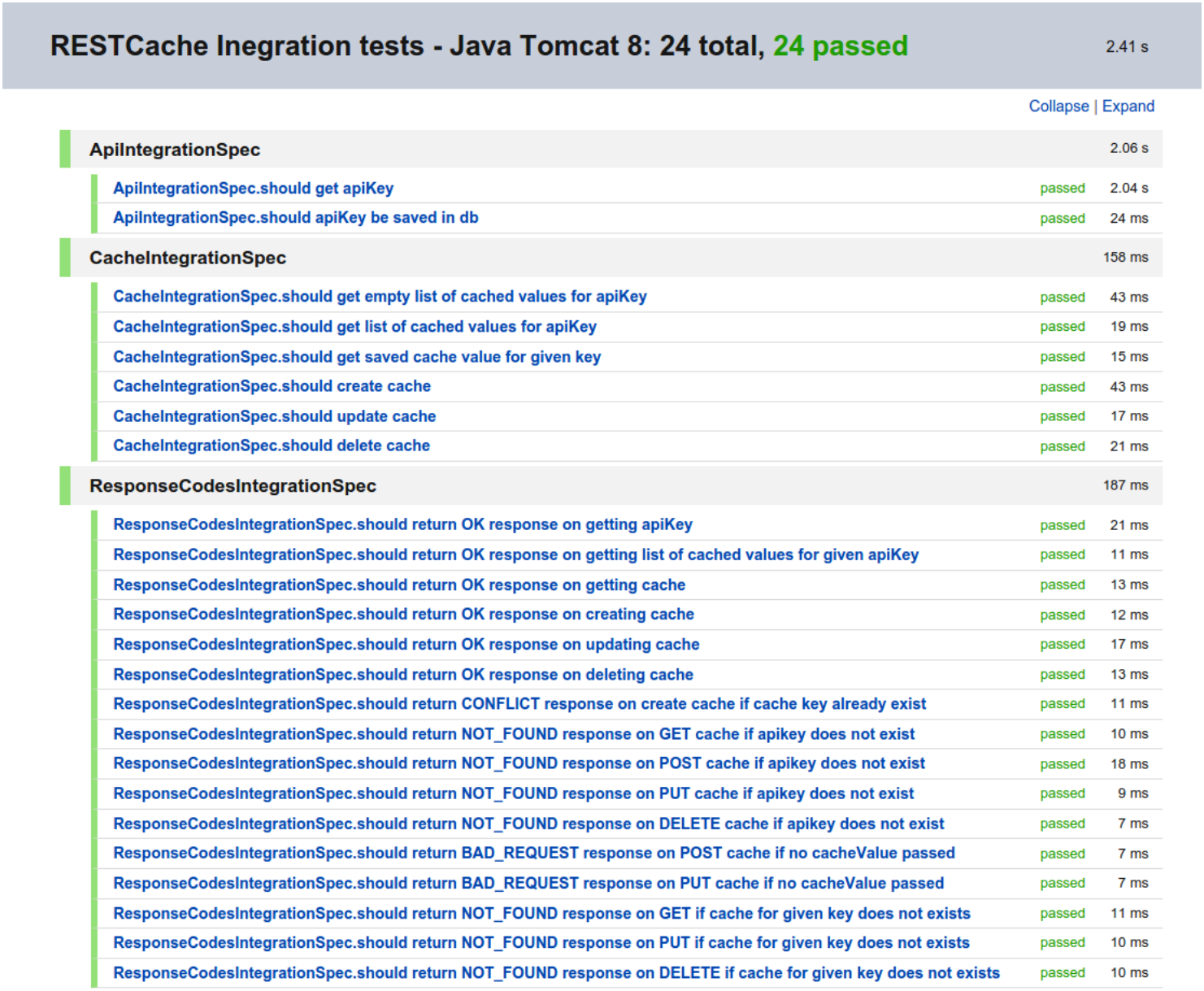
Aplikacja w języku *Go* wykorzystywała do działania znacznie mniej zasobów systemowych od serwerów uruchamiających aplikację w języku *Java*. Serwer *Jetty* wymagał nieznacznie większych zasobów procesora i pamięci niż *Tomcat*. W porównaniu z aplikacją w języku *Go* oba serwery wymagały około 6 razy więcej pamięci i nieznacznie bardziej wykorzystywały procesor.

Niskie wykorzystanie zasobów systemowych przez aplikację jest ważne, jeśli architektura systemu składa się z małych usług, które są uruchamiane w rozproszonym środowisku. Dzięki temu można uruchomić wiele aplikacji, na dużo mniejszej ilości serwerów, co przekłada się na utrzymanie całego systemu.

Podsumowując należy podkreślić, że testowanie wydajności serwisów i całych systemów jest ważnym procesem związanym z ich wytwarzaniem. Dzięki testom można określić jak szybko aplikacja odpowiada na działania użytkowników oraz jaką ich ilość jest w stanie obsłużyć równocześnie, bez zauważalnych spadków wydajności. Testy wydajnościowe pozwalają też wykryć najsłabsze punkty systemu i wyeliminować je. Rezygnacja z testowania wydajności systemu może być bardzo kosztowna, szczególnie gdy będzie on działał produkcyjnie, a jego popularność przerośnie założenia projektantów.

Dodatek A

Wyniki testów akceptacyjnych

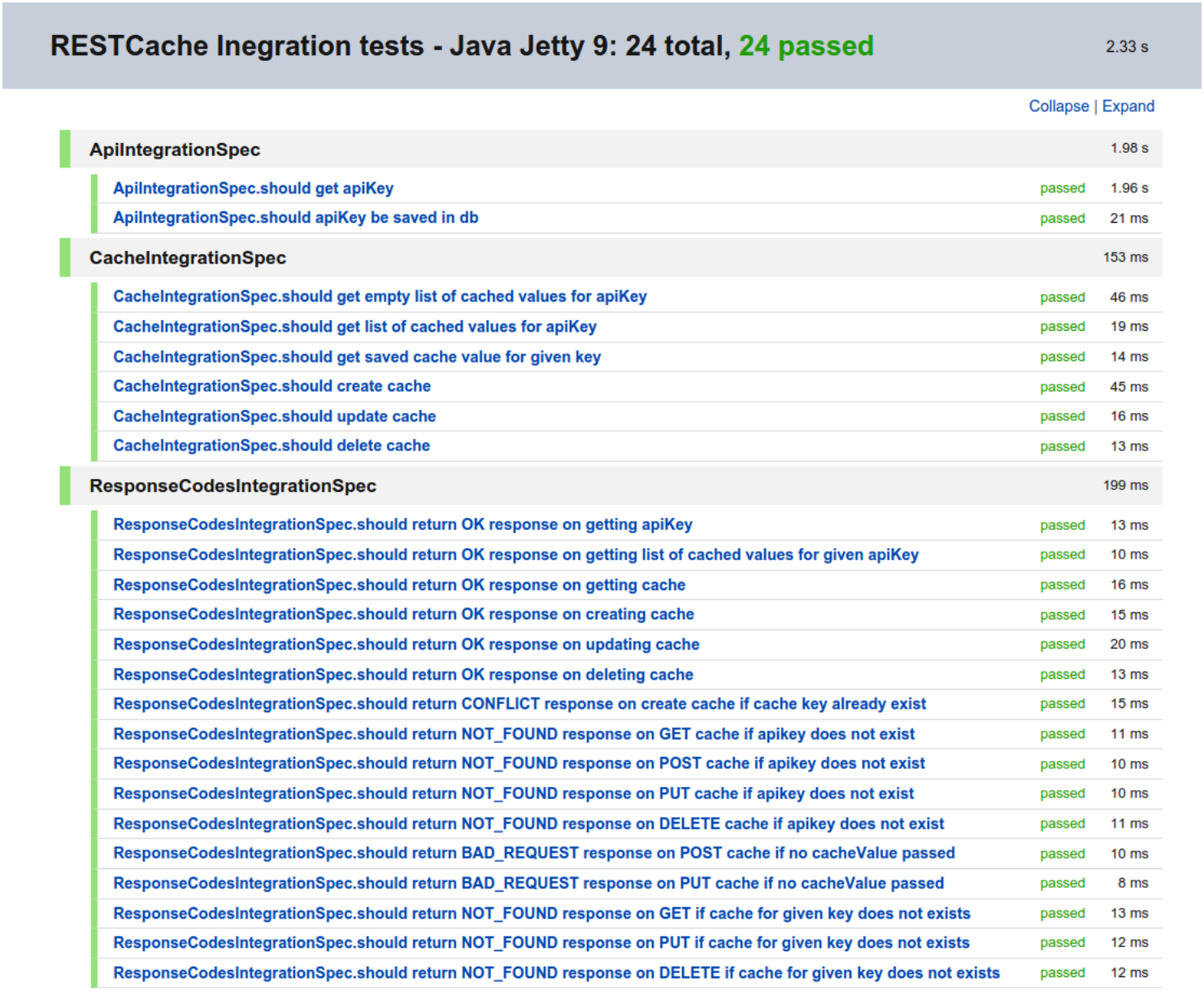


RESTCache Inegration tests - Java Tomcat 8: 24 total, 24 passed 2.41 s

Collapse | Expand

ApiIntegrationSpec		2.06 s
ApiIntegrationSpec.should get apiKey	passed	2.04 s
ApiIntegrationSpec.should apiKey be saved in db	passed	24 ms
CacheIntegrationSpec		158 ms
CacheIntegrationSpec.should get empty list of cached values for apiKey	passed	43 ms
CacheIntegrationSpec.should get list of cached values for apiKey	passed	19 ms
CacheIntegrationSpec.should get saved cache value for given key	passed	15 ms
CacheIntegrationSpec.should create cache	passed	43 ms
CacheIntegrationSpec.should update cache	passed	17 ms
CacheIntegrationSpec.should delete cache	passed	21 ms
ResponseCodesIntegrationSpec		187 ms
ResponseCodesIntegrationSpec.should return OK response on getting apiKey	passed	21 ms
ResponseCodesIntegrationSpec.should return OK response on getting list of cached values for given apiKey	passed	11 ms
ResponseCodesIntegrationSpec.should return OK response on getting cache	passed	13 ms
ResponseCodesIntegrationSpec.should return OK response on creating cache	passed	12 ms
ResponseCodesIntegrationSpec.should return OK response on updating cache	passed	17 ms
ResponseCodesIntegrationSpec.should return OK response on deleting cache	passed	13 ms
ResponseCodesIntegrationSpec.should return CONFLICT response on create cache if cache key already exist	passed	11 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on GET cache if apikey does not exist	passed	10 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on POST cache if apikey does not exist	passed	18 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on PUT cache if apikey does not exist	passed	9 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on DELETE cache if apikey does not exist	passed	7 ms
ResponseCodesIntegrationSpec.should return BAD_REQUEST response on POST cache if no cacheValue passed	passed	7 ms
ResponseCodesIntegrationSpec.should return BAD_REQUEST response on PUT cache if no cacheValue passed	passed	7 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on GET if cache for given key does not exists	passed	11 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on PUT if cache for given key does not exists	passed	10 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on DELETE if cache for given key does not exists	passed	10 ms

Rysunek A.1: Wynik testów integracyjnych aplikacji w języku Java na kontenerze Tomcat 8



RESTCache Inegration tests - Java Jetty 9: 24 total, 24 passed 2.33 s

Collapse | Expand

ApiIntegrationSpec		1.98 s
ApiIntegrationSpec.should get apiKey	passed	1.96 s
ApiIntegrationSpec.should apiKey be saved in db	passed	21 ms
CacheIntegrationSpec		153 ms
CacheIntegrationSpec.should get empty list of cached values for apiKey	passed	46 ms
CacheIntegrationSpec.should get list of cached values for apiKey	passed	19 ms
CacheIntegrationSpec.should get saved cache value for given key	passed	14 ms
CacheIntegrationSpec.should create cache	passed	45 ms
CacheIntegrationSpec.should update cache	passed	16 ms
CacheIntegrationSpec.should delete cache	passed	13 ms
ResponseCodesIntegrationSpec		199 ms
ResponseCodesIntegrationSpec.should return OK response on getting apiKey	passed	13 ms
ResponseCodesIntegrationSpec.should return OK response on getting list of cached values for given apiKey	passed	10 ms
ResponseCodesIntegrationSpec.should return OK response on getting cache	passed	16 ms
ResponseCodesIntegrationSpec.should return OK response on creating cache	passed	15 ms
ResponseCodesIntegrationSpec.should return OK response on updating cache	passed	20 ms
ResponseCodesIntegrationSpec.should return OK response on deleting cache	passed	13 ms
ResponseCodesIntegrationSpec.should return CONFLICT response on create cache if cache key already exist	passed	15 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on GET cache if apikey does not exist	passed	11 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on POST cache if apikey does not exist	passed	10 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on PUT cache if apikey does not exist	passed	10 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on DELETE cache if apikey does not exist	passed	11 ms
ResponseCodesIntegrationSpec.should return BAD_REQUEST response on POST cache if no cacheValue passed	passed	10 ms
ResponseCodesIntegrationSpec.should return BAD_REQUEST response on PUT cache if no cacheValue passed	passed	8 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on GET if cache for given key does not exists	passed	13 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on PUT if cache for given key does not exists	passed	12 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on DELETE if cache for given key does not exists	passed	12 ms

Rysunek A.2: Wynik testów integracyjnych aplikacji w języku Java na kontenerze Jetty 9

RESTCache Inegration tests - GO: 24 total, 24 passed

2.15 s

Collapse | Expand

ApiIntegrationSpec		1.90 s
ApiIntegrationSpec.should get apiKey	passed	1.88 s
ApiIntegrationSpec.should apiKey be saved in db	passed	19 ms
CacheIntegrationSpec		124 ms
CacheIntegrationSpec.should get empty list of cached values for apiKey	passed	41 ms
CacheIntegrationSpec.should get list of cached values for apiKey	passed	13 ms
CacheIntegrationSpec.should get saved cache value for given key	passed	12 ms
CacheIntegrationSpec.should create cache	passed	38 ms
CacheIntegrationSpec.should update cache	passed	12 ms
CacheIntegrationSpec.should delete cache	passed	8 ms
ResponseCodesIntegrationSpec		123 ms
ResponseCodesIntegrationSpec.should return OK response on getting apiKey	passed	9 ms
ResponseCodesIntegrationSpec.should return OK response on getting list of cached values for given apiKey	passed	7 ms
ResponseCodesIntegrationSpec.should return OK response on getting cache	passed	9 ms
ResponseCodesIntegrationSpec.should return OK response on creating cache	passed	6 ms
ResponseCodesIntegrationSpec.should return OK response on updating cache	passed	10 ms
ResponseCodesIntegrationSpec.should return OK response on deleting cache	passed	10 ms
ResponseCodesIntegrationSpec.should return CONFLICT response on create cache if cache key already exist	passed	8 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on GET cache if apikey does not exist	passed	9 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on POST cache if apikey does not exist	passed	7 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on PUT cache if apikey does not exist	passed	8 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on DELETE cache if apikey does not exist	passed	7 ms
ResponseCodesIntegrationSpec.should return BAD_REQUEST response on POST cache if no cacheValue passed	passed	6 ms
ResponseCodesIntegrationSpec.should return BAD_REQUEST response on PUT cache if no cacheValue passed	passed	6 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on GET if cache for given key does not exists	passed	6 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on PUT if cache for given key does not exists	passed	7 ms
ResponseCodesIntegrationSpec.should return NOT_FOUND response on DELETE if cache for given key does not exists	passed	8 ms

Rysunek A.3: Wynik testów integracyjnych aplikacji w GO

Bibliografia

[1] Jim Webber, Savas Parastatidis, Ian Robinson, "REST in Practice: Hypermedia and Systems Architecture", O'Reilly Media, 2010

[2] Sam Newman, "Building Microservices", O'Reilly Media, 2015.

[3] Eric Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", Addison-Wesley, 2003.

[4] Cay S. Horstmann, Gary Cornell, "Java. Podstawy. Wydanie IX", Helion, 2013.

[5] Rod Jhnson, Ęxpert One-on-One J2EE Design and Development. New edition". Wrox, 2002.

[6] Craig Walls, Śpring w akcji. Wydanie III". Helion, 2013.

[7] Mark Summerfield, "Programming in Go", Addison-Wesley Professional, 2012

[8] David Chisnall, "The Go Programming Language Phrasebook", Addison-Wesley Professional, 2012

[9] `https://www.docker.com/`

[10] `https://www.mongodb.org/`

[11] `https://jmeter.apache.org/`

[12] `https://www.digitalocean.com/`

[13] `https://github.com/julienschmidt/httprouter`