

Kierunek: **Informatyka techniczna**
Specjalność: **Inżynieria systemów informatycznych**

**PRACA DYPLOMOWA
MAGISTERSKA**

**Analiza wydajnościowa interfejsów
API w technologiach C# oraz NodeJS**

**Performance analysis of C# and NodeJS
APIs**

Maciej Grzela

Opiekun pracy
dr inż., Michał Kucharzak

Streszczenie

Celem dokumentu jest analiza wydajności usług sieciowych implementowanych jako interfejsy programowania aplikacji, przy wykorzystaniu stosu technologicznego C# / .NET oraz JavaScript / NodeJS. Zdecydowano się nie tylko, na porównanie efektywności realizacji podstawowych operacji wykonywanych na danych w kontekście zmennego ruchu sieciowego oraz różnych systemów bazodanowych, ale także dokonano ewaluacji wydajności technik programowania współbieżnego, obsługi operacji asynchronicznych, wprowadzenia wzorca projektowego podziału odpowiedzialności, czy też wpływu wdrożenia systemów internetowych na odmiennych platformach chmurowych.

Ponadto, zaproponowano autorski mechanizm wyliczania czasu przechowywania wpisu w pamięci podręcznej i porównano go z szeroko wykorzystywanym mechanizmem statycznym. W odniesieniu do oceny wydajności operacji współbieżnych, zaimplementowano algorytm genetyczny dla problemu komiwojażera i zbadano liczbę wykonywanych iteracji i także współczynnik rezultatu względem rozwiązania optymalnego. Co się tyczy ewaluacji obsługi wywołań asynchronicznych, przeprowadzono ocenę efektywności natywnych klientów dla obu środowisk analizując średnich czasów pozyskiwania danych dla zmiennej liczby otrzymywanych encji bazodanowych. Jako zewnętrzne źródło pozyskiwania danych zaimplementowano odseparowaną lokalną usługę sieciową. Zbadano również wpływ wykorzystania zaawansowanego wzorca projektowego podziału odpowiedzialności, a także w jaki sposób, wprowadzenie rozbudowanych technik optymalizacji, możliwych dzięki przytoczonemu wzorcowi, wpływa na czas odpowiedzi interfejsów API na poszczególny rodzaj żądania. Na koniec, dokonano obserwacji dysproporcji pomiędzy czasem przetwarzania żądania usług sieciowych uruchamianych w dedykowanych środowiskach chmurowych, oraz na platformie typu infrastructure-as-a-service.

Przeprowadzone badania wykazały silną zależność wykorzystywanego silnika bazy danych, w odniesieniu do maksymalnego ruchu sieciowego, jaki są w stanie obsługiwać API. Ponadto, znaczącą dysproporcję dotyczącą obsługi operacji współbieżnych faworyzującą rozwiązanie bazujące na języku C#, czy też silną korelację pomiędzy niską złożonością natywnego klienta http API języka JavaScript a wydajnością jego działania. Analizując wydajność rozwiązań pamięci podręcznej zaobserwowano przewagę autorskiego mechanizmu dla stałego poziomu natężenia generowanego w stałym czasie, przy uwzględnieniu losowego rozmieszczenia punktów unieważnień.

Słowa kluczowe: interfejsy programowania aplikacji, C# .NET, JavaScript, NodeJS, mappery obiektowo-relacyjne, operacje współbieżne, operacje asynchroniczne, obsługa pamięci podręcznej, wzorzec podziału odpowiedzialności, replikacja transakcyjna, platformy chmurowe

Spis treści

1. Wstęp	7
1.1. Geneza pracy	7
1.2. Cel i zakres pracy	9
1.3. Struktura pracy	10
2. Wprowadzenie teoretyczne	11
2.1. Wykorzystywane terminy	11
2.2. Interfejsy programowania aplikacji	16
2.3. Testowanie oprogramowania	24
2.4. Wykorzystywane narzędzia i technologie	27
2.5. Przegląd literatury	32
3. Opis problemu badawczego	39
3.1. Przedstawienie aspektów problemu badawczego	39
3.2. Sformułowanie scenariuszy badawczych	43
4. Implementacja środowiska badawczego	53
4.1. Realizacja topologii fizycznych	53
4.2. Budowa interfejsów programowania aplikacji	58
4.3. Konfiguracja generycznych oraz dedykowanych platform chmurowych	69
4.4. Konfiguracja narzędzia do realizacji badań	70
5. Przeprowadzone badania	74
5.1. Wpływ zastosowanego systemu bazodanowego na efektywność działania interfejsu API	74
5.2. Wpływ zastosowanej technologii na wydajność realizacji operacji współbieżnych	81
5.3. Wpływ zastosowanej technologii na efektywność obsługi operacji asynchronicznych	84
5.4. Wpływ implementacji wzorca projektowego CQRS na wydajność obsługi żądania	85
5.5. Porównanie efektywności obsługi zapytań dla odmiennych mechanizmów pamięci podręcznej	87
5.6. Zmiennaść wydajności api wdrożonego na generycznej oraz dedykowanej platformie chmurowej	90
6. Podsumowanie	95
6.1. Uzyskane efekty pracy	95
6.2. Perspektywy rozwoju badań	96
Literatura	98

Spis rysunków

2.1. Proces przetwarzania żądania wewnątrz interfejsu API	20
2.2. Zasada działania oprogramowania mappera obiektowo-relacyjnego w kontekście jednolitego interfejsu operacji na zbiorze danych	21
2.3. Proces uwierzytelnienia oraz autoryzacji użytkownika przed interfejsem API	22
2.4. Schemat przetworzenia żądania przez interfejs API dla architektury z jednym modelem danych	23
2.5. Schemat przetworzenia żądania przez interfejs API dla architektury wykorzystującej wzorzec projektowy CQRS	23
4.1. Konfiguracja pierwsza lokalnej topologii fizycznej środowiska badawczego	54
4.2. Konfiguracja druga lokalnej topologii fizycznej środowiska badawczego	55
4.3. Konfiguracja trzecia lokalnej topologii fizycznej środowiska badawczego	56
4.4. Konfiguracja czwarta lokalnej topologii fizycznej środowiska badawczego	57
4.5. Konfiguracja pierwsza rozległej topologii fizycznej środowiska badawczego	58
4.6. Struktura obiektów wewnątrz rozwiązania systemu internetowego zaimplementowanego w języku C#	60
4.7. Struktura obiektów wewnątrz rozwiązania systemu internetowego zaimplementowanego w języku JavaScript	61
4.8. Schemat blokowy algorytmu genetycznego	62
4.9. Format klucza wpisu przechowywanego w pamięci podręcznej	64
4.10. Schemat wykonywania procedur zapisu oraz odczytu z wykorzystaniem replikacji transakcyjnej	68
4.11. Konfiguracja pierwsza planu testowego JMeter	72
4.12. Konfiguracja druga planu testowego JMeter	73
5.1. Czas odpowiedzi na żądanie dla konfiguracji .NET/SQL Server w kontekście testu funkcjonalnego	76
5.2. Obszary satysfakcji, tolerancji oraz frustracji wskaźnika APDEX względem poszczególnych systemów bazodanowych dla API zaimplementowanego w C#	76
5.3. Obszary satysfakcji, tolerancji oraz frustracji wskaźnika APDEX względem poszczególnych systemów bazodanowych dla API zaimplementowanego w JavaScript	77
5.4. Średnie czasy odpowiedzi na żądanie względem liczby procesów generujących oraz systemu bazodanowego	78
5.5. Średnie czasy odpowiedzi na żądanie oraz procent niepoprawnych odpowiedzi względem liczby procesów generujących	80
5.6. Uśredniony czas realizacji operacji asynchronicznych względem zmiennej liczby pobieranych encji	85
5.7. Wydajność działania interfejsów API dla operacji odczytu oraz zapisu, przed i po zastosowaniu wzorca podziału odpowiedzialności	86
5.8. Porównanie czasów odpowiedzi na żądanie dla odmiennych mechanizmów pamięci podręcznej	89

5.9. Wydajność działania mierzona czasem wewnętrznego przetwarzania operacji CRUD dla platform Azure oraz DigitalOcean - Interfejs API C# .NET	91
5.10. Wydajność działania mierzona czasem wewnętrznego przetwarzania operacji CRUD dla platform Heroku oraz DigitalOcean - Interfejs API NodeJS	92
5.11. Liczba błędnych odpowiedzi względem typu żądania oraz platformy wdrożeniowej	93

Spis tabel

2.1. Zbiór dozwolonych metod protokołu hipertekstowego	16
2.1. Zbiór dozwolonych metod protokołu hipertekstowego	17
2.2. Zbiór najczęściej wykorzystywanych linii nagłówkowych w kontekście żądania protokołu hipertekstowego	17
2.2. Zbiór najczęściej wykorzystywanych linii nagłówkowych w kontekście żądania protokołu hipertekstowego	18
2.3. Zbiór najczęściej zwracanych linii nagłówkowych w kontekście odpowiedzi protokołu hipertekstowego	19
2.4. Zbiór kodów statusu odpowiedzi protokołu hipertekstowego	19
3.1. Scenariusz badawczy - badanie przeprowadzone w kontekście systemów bazodanowych	43
3.2. Scenariusz badawczy - badanie przeprowadzone w kontekście realizacji operacji współbieżnych	45
3.3. Scenariusz badawczy - badanie przeprowadzone w kontekście obsługi operacji asynchronicznych	46
3.4. Scenariusz badawczy - badanie przeprowadzone w kontekście zastosowania wzorca projektowego CQRS	48
3.5. Scenariusz badawczy - badanie przeprowadzone w kontekście wykorzystania mechanizmów pamięci podręcznej	49
3.6. Scenariusz badawczy - badanie przeprowadzone w kontekście wdrażania oprogramowania na platformach chmurowych	51
4.1. Wykaz hiperparametrów zaimplementowanego algorytmu genetycznego	63
5.1. Wykaz punktów końcowych wykorzystywanych w badaniu wpływu wykorzystanego systemu bazodanowego na działanie API	75
5.2. Wydajność realizacji algorytmu genetycznego dla problemu komiwojażera w zależności od czasu przetwarzania oraz technologii - współczynnik jakości rozwiązania .	83
5.3. Wydajność realizacji algorytmu genetycznego dla problemu komiwojażera w zależności od czasu przetwarzania oraz technologii - liczba iteracji pętli algorytmu .	83
5.4. Macierz istotności statystycznej dla współczynników jakości rozwiązania pozyskanych w ramach badania wydajności obsługi operacji współbieżnych	84
5.5. Wydajność metod obsługi pamięci podręcznej względem technologii oraz momentu unieważnienia wpisu	88

Rozdział 1

Wstęp

1.1. Geneza pracy

Usługi sieciowe, zarówno te dostępne publicznie jak i te realizowane dla celów prywatnych, pełnią kluczową rolę w kontekście funkcjonowania współczesnej sieci internetowej. Zapewne nikt z nas, nie jest w stanie wyobrazić sobie kształtu obecnego Internetu bez takich rozwiązań sieciowych jak obsługa poczty elektronicznej, realizacja transferu plików, czy też przede wszystkim dostęp do aplikacji oraz witryn internetowych. Szczególnie w obrębie ostatniej spośród wymienionych usług, na przestrzeni ostatnich lat zauważać można bardzo dużą liczbę zmian dotyczących sposobu ich definiowania oraz realizacji. Powodem pojawiania się tych zmian, jest niewątpliwie konieczność zachowania bądź też zwiększenia poziomów wydajności, niezawodności oraz bezpieczeństwa oferowanych rozwiązań, uwzględniając coraz to większy ruch sieciowy, generowany przez nieustannie zwiększającą się liczbę użytkowników Internetu. Ponadto, od nowoczesnego systemu internetowego, wymaga się coraz to większego poziomu skalowalności, a także płynności działania.

Poparciem niniejszych słów, może być treść wydawanego w kilkuletnich odstępach czasu raportu firmy Cisco, dotyczącego przewidywań oraz trendów sieciowych (tj. Cisco Annual Internet Report). Zgodnie z przedstawionymi w przytoczonym raporcie informacjami, a także porównując informacje te, z faktycznymi wartościami wskaźników dotyczących ruchu w internecie, zaobserwować możemy niemalże trzykrotny wzrost globalnego ruchu sieciowego na przestrzeni ostatnich pięciu lat. Ponadto, liczba klienckich urządzeń sieciowych, wykorzystywanych w celu uzyskania dostępu do usług udostępnianych w Internecie, na przestrzeni analogicznego przedziału czasowego, zwiększyła się z wartości 2,4 urządzenia na osobę, do poziomu niemalże czterech hostów sieciowych przypadających na pojedynczego reprezentanta globalnej populacji.

Należy także zwrócić uwagę, jakiego typu ruch sieciowy pełni dominującą rolę w kontekście dzisiejszego Internetu. Ponad 80% globalnego konsumenckiego ruchu internetowego stanowią dane dotyczące usług wideo, około dziesięciu procent światowego ruchu obejmują pozostałe treści udostępniane w ramach aplikacji oraz witryn internetowych, a pozostałe 10% to ruch generowany m.in. przez usługi transferu plików, poczty elektronicznej, czy też gier online. Na podstawie tych informacji, zauważać można, że ponad 90% całości danych, przesyłanych w ramach globalnej sieci, musi być przetwarzanych przez aplikacje internetowe, bądź usługi sieciowe z nimi powiązane. Dlatego też, zaawansowane witryny internetowe komunikujące się z usługami sieciowymi, zwane dziś systemami internetowymi, tworzone są z wykorzystaniem coraz to bardziej udoskonalonych modeli architektonicznych, pozwalających na coraz to łatwiejszą budowę i rozwój rozwiązań przystosowanych do potrzeb aktualnego ruchu sieciowego [7].

Jednym z pierwszych, a także najbardziej podstawowych podejść do projektowania i implementacji systemów internetowych było wprowadzenie modelu architektury definiującego aplika-

cje monolityczne. W modelu tym, użytkownik aplikacji, wykorzystując oprogramowanie klienckie, którym w tym przypadku jest przeglądarka internetowa, wysyłał żądanie uzyskania zasobu definiując odpowiedni adres url (*ang. Uniform Resource Locator*). Żądanie to, odwoływało się bezpośrednio do fizycznego zasobu zlokalizowanego na serwerze, który przed dostarczeniem do klienta był przetwarzany przez serwer w celu uzupełnienia go danymi uzyskanymi z zewnętrznych źródeł - m.in. z systemu bazodanowego. Odpowiednio przygotowana statyczna zawartość odpowiedzi serwera, przybierająca postać pliku HTML (*ang. HyperText Markup Language*) była następnie przesyłana bezpośrednio do przeglądarki internetowej. Podejście to, wyróżniało się całkowitym brakiem dynamiki działania systemu internetowego, ponieważ każde zdarzenie wywoływanie przez oprogramowanie klienta, wymagało zaadresowania i wygenerowania nowego żądania w kierunku serwera, którego odpowiedzią była nowa zawartość warstwy prezentacyjnej systemu.

W związku z zauważeniem pewnej regularności dotyczącej funkcjonowania większości systemów internetowych, związanej z faktem niejednokrotnego generowania nieznacznie różniących się od siebie odpowiedzi serwera, a także w związku z rozwojem języka skryptowego JavaScript oraz technologii Flash, aplikacje w ramach architektury monolitycznej zaczęły uwzględniać obsługę żądań zawierających przetworzone fragmenty warstwy prezentacyjnej. Ponadto, możliwa stała się dynamiczna podmiana określonych fragmentów treści, bez konieczności ponownego pozyskiwania pozostały zawartości widoku. Usprawnienie to, opierające się na technice realizacji żądań asynchronicznych w ramach JavaScript (*ang. AJAX - Asynchronous JavaScript and XML*) pozwoliło na poprawę wydajności działania aplikacji internetowych przyczyniając się do zmniejszenia częstotliwości generowania zapytań, a także redukcji rozmiaru pojedynczej odpowiedzi serwera. Rozwiążanie to, nie wpływało jednakże bezpośrednio na strukturę systemu, której głównymi mankamentami były: pojedynczy centralny punkt przetwarzania żądań, a także brak separacji logiki działania systemu od warstwy prezentacyjnej.

Niedoskonałości omówionego powyżej modelu zostały zniwelowane poprzez wprowadzenie architektury zorientowanej na serwisy (*ang. SOA - Service Oriented Architecture*). W podejściu tym, dokonano separacji warstwy prezentacyjnej systemu, a także wszystkich pozostałych funkcjonalności dotyczących logiki biznesowej oraz przetwarzania danych. Reużywalne oraz autonomiczne usługi sieciowe pozwalały na realizację określonych funkcji systemu, a sposób komunikacji klienta z usługą, jak i komunikacji pomiędzy poszczególnymi serwisami definiowany był przez standaryzowane kontrakty. Zdefiniowanie architektury zorientowanej na serwisy umożliwiło budowę skalowalnych systemów internetowych, których poszczególne części mogły być realizowane w dowolnej technologii, a implementacja nowej funkcjonalności nie wymagała przebudowy pozostałych komponentów. Rozwiążanie to, wprowadzało jednak dodatkowy narzut dla każdej z przesyłanych wiadomości, wynikający ze ścisłe określonej struktury żądania, tworzonej z wykorzystaniem języka XML (*ang. Extensible Markup Language*). Ponadto, wraz ze wzrostem poziomu zaawansowania systemu internetowego, autonomiczność oraz reużywalność poszczególnych komponentów malała ze względu na powstawanie specyficznych dla określonego rozwiązania zależności [37].

W związku z coraz to większymi wymaganiami dotyczącymi aplikacji internetowych, dominująca ówcześnie architektura rozproszonych usług sieciowych zastąpiona została poprzez model uwzględniający warstwę kliencką oraz interfejs programowania aplikacji (*ang. Application Programming Interface*). W przypadku nowoczesnych systemów internetowych, oba z tych komponentów budowane są w oparciu o architekturę n-warstwową (*ang. N-Tier Architecture Application*). W ramach niniejszego modelu, klient wysyła żądanie do interfejsu API, który na początku przetwarza jego treść, a następnie wywołuje usługę utworzoną w celu realizacji określonego zadania. Celem serwisu jest przetworzenie logiki biznesowej dla danej funkcjonalności, a także odwołanie się do usług dostępu do danych w celu ich uzyskania z zewnętrznego źródła informacji. Odpowiednio przygotowana odpowiedź jest następnie przekazywana do warstwy

obsługi żądania, która zwraca ją określonemu klientowi. W przeciwnieństwie do pierwszego z przytoczonych modeli, odpowiedzią API nie jest dokument HTML, a jedynie dane dotyczące zasobu, które chce uzyskać klient. Sam zasób natomiast, nie jest elementem warstwy prezentacji systemu a zbiorem danych lub typem operacji, które można na tym zbiorze wykonać. Upraszczając, stwierdzić można, że API pełni rolę pośrednika pomiędzy warstwą prezentacji a zbiorem danych oraz operacji ich przetwarzania, a także dostarczania. Poszczególne usługi realizujące logikę biznesową aplikacji zawarte są bezpośrednio wewnętrz API, co nie oznacza jednakże, że nie mogą odwoływać się do serwisów zewnętrznych. Takie podejście do budowania systemów internetowych zapewnia zarówno skalowalność poszczególnych aplikacji wchodzących w skład systemu, jak i rozwiązuje problemy architektury SOA związane z zależnościami występującymi pomiędzy usługami. Dlatego też, architektura ta jest powszechnie wykorzystywana w celu budowy i zarządzania nowoczesnymi oraz zaawansowanymi systemami internetowymi [30].

Zarówno zdecentralizowana architektura zorientowana na serwisy, jak i centralna architektura oparta o interfejs programowania aplikacji, w przeciwnieństwie do architektury monolitycznej, dostarcza zdecydowanie więcej możliwości związanych z ewaluacją działania poszczególnych komponentów systemu. Dzięki powstaniu ostatnich dwóch, spośród trzech przedstawionych modeli architektonicznych, możliwe jest nie tylko zbudowanie efektywnie działającej aplikacji internetowej, ale także ciągła ocena poprawności implementacji jej komponentów, w celu ustawicznego doskonalenia całego systemu.

Niniejsza praca, traktować będzie o ewaluacji efektywności działania interfejsów programowania aplikacji, w kontekście jednych z dwóch najpopularniejszych środowisk rozwoju oraz uruchamiania api. Ponadto, porównane zostaną parametry wydajnościowe w kontekście określonych przypadków użycia interfejsu API, będącego niezbędną częścią powszechnie wykorzystywanej architektury systemów internetowych.

1.2. Cel i zakres pracy

Celem pracy jest porównanie wydajności działania interfejsów programowania aplikacji, tworzących z wykorzystaniem języków programowania C# oraz JavaScript. Interfejsy, wykonywane są w dwóch różnych środowiskach uruchomieniowych. Dla języka C#, środowiskiem tym jest platforma .NET, natomiast dla języka JavaScript – platforma NodeJS. Analiza porównawcza, obejmuje zarówno aspekty dotyczące efektywności działania samego interfejsu programowania aplikacji, jak i elementów wchodzących w skład tworzonego systemu. Wśród omawianych rozwiązań, wyróżnić należy mappery obiektowo-relacyjne, systemy bazodanowe, czy też mechanizmy zarządzania pamięcią podręczną. Niektóre spośród wymienionych modułów stanowią integralną część API, natomiast pozostałe służą do rozszerzenia jego funkcjonalności.

Zakres pracy obejmuje: przegląd literaturowy, implementację środowiska badawczego, realizację badań oraz opracowanie wyników. Przegląd literatury tyczy się aspektów związanych ze strukturą i zasadą działania interfejsów programowania aplikacji, a także kwestii dotyczących wykonywania pomiarów wydajności dla poszczególnych operacji sieciowych. Operacje sieciowe, realizowane są w ramach obsługi żądania przez API. Etap implementacji środowisk badawczych składa się z budowy interfejsów w oparciu o porównywane środowiska rozwoju i uruchamiania aplikacji, a także konfiguracji platformy lokalnej oraz platform chmurowych, pozwalających na przeprowadzanie analizy działania systemów. Realizacja badań, przeprowadzona została pod kątem pomiaru czasu odpowiedzi na żądania użytkownika końcowego biorąc pod uwagę aspekty: wywołania serii żądań, obsługi współbieżności procesów, dostępności zasobów platformy hostingowej, a także możliwości oferowanych przez mappery obiektowo-relacyjne oraz systemy bazodanowe. Celem etapu opracowania wyników jest przedstawienie, wizualizacja oraz analiza różnic wartości czasów odpowiedzi interfejsów API na poszczególne

żądania, w odniesieniu do przeprowadzonych badań. Zastosowanymi kryteriami oceny podczas przeprowadzanej analizy jest czas odpowiedzi interfejsu programowania aplikacji dla wygenerowanego żądania, a także maksymalna liczba żądań jakie jest w stanie obsłużyć określone API. Przedstawione kryteria, uwzględniane zostały w odniesieniu do wykorzystywanego środowiska uruchomieniowego oraz technologii implementacyjnej. Przeprowadzone badania, mają służyć wskazaniu zarówno pozytywnych aspektów, jak i problemów dotyczących wydajności działania aplikacji tworzonych z wykorzystaniem porównywanych technologii. Ponadto, celem jest także przedstawienie możliwości zwiększenia efektywności implementowanych interfejsów programowania aplikacji.

1.3. Struktura pracy

Niniejsza praca, podzielona została na sześć rozdziałów.

Pierwszy z nich, napisany został w celu zobrazowania dziedziny rozważanego problemu, a także podkreślenia jego wagi w kontekście zagadnienia usług sieciowych. Ponadto, w rozdziale tym zdefiniowano cel popełnionej pracy oraz przedstawiono zakres czynności realizowanych w ramach przeprowadzonych badań.

W rozdziale drugim dokonano wprowadzenia teoretycznego do tematyki interfejsów programowania aplikacji oraz testowania usług sieciowych. Wprowadzenie to, w odniesieniu do interfejsów API dotyczy zarówno struktury i zasady działania omawianej usługi sieciowej, jak i sposobu realizacji połączeń tej usługi z zewnętrznymi źródłami danych. W ramach wprowadzenia do tematyki testowania usług sieciowych wyjaśniono fundamentalne pojęcia teorii testowania oraz omówiono dostępne modele realizacji testów. Co więcej, nakreślono strategię wykonywania pomiarów wydajności w kontekście usług pracujących w sieciach komputerowych. W niniejszym rozdziale, zawarto również przegląd pozycji literaturowych, pomocnych w trakcie realizacji badań, a także przegląd technologii informatycznych, zastosowanych w procesie implementacji środowiska badawczego oraz wykonania pomiarów.

W ramach trzeciego z rozdziałów, zdefiniowano i omówiono każdy z aspektów rozważanego problemu badawczego. Dzięki temu, możliwe stało się sformułowanie zbioru rozważanych scenariuszy badawczych.

W celu realizacji badań opartych o zdefiniowane w rozdziale trzecim scenariusze badawcze, należy zaprojektować oraz zaimplementować odpowiednio dostosowane środowisko badań. Poszczególne kroki realizacji tego środowiska, zarówno te, dotyczące jego fizycznej struktury, jak i te, które tyczą się implementacji interfejsów programowania aplikacji, opisane zostały w rozdziale czwartym niniejszej pracy.

Piąty z rozdziałów, ma na celu przedstawienie rezultatów wynikających z przeprowadzonych prac naukowych. Rezultaty te, w obrębie niniejszego rozdziału zostały zgrupowane względem zdefiniowanych uprzednio scenariuszy badawczych, realizowanych w odpowiednio przystosowanym środowisku. Ponadto, dla uzyskanych wartości pomiarowych, dotyczących kryteriów poszczególnych badań, wykonano testy parametryczne, dzięki którym możliwa jest ocena istotności statystycznej zaobserwowanych różnic wynikowych. Co więcej, wyniki każdego z realizowanych scenariuszy badawczych poddane zostały krytycznej analizie.

Ostatni z rozdziałów pełni rolę podsumowania. Autor przedstawia w nim uzyskane efekty wykonanej pracy, a także nakreśla możliwości związane z dalszym rozwojem badań.

Rozdział 2

Wprowadzenie teoretyczne

2.1. Wykorzystywane terminy

W niniejszej pracy, posłużono się terminologią dystynktywną z punktu widzenia realizacji, rozwoju oraz ewaluacji usług sieciowych. Najbardziej istotne spośród wykorzystywanych terminów wymieniono poniżej. Dla każdego z pojęć, przedstawiono obcojęzyczne tłumaczenie, a także zdefiniowano spójny oraz zwięzły opis.

Usługa sieciowa

Web Service

Rodzaj systemu informatycznego cechującego się permanentnym wykonywaniem zdefiniowanych funkcji, tuż po uzyskaniu żądania. Żądanie to, przybiera postać danych, przekazanych w ramach systematycznej struktury. Sposób dostarczenia żądania, jego format, a także metoda odpowiedzi na żądanie, definiowane są poprzez protokół sieciowy z którego korzysta dana usługa.

Interfejs Programowania Aplikacji

Application Programming Interface

Zbiór reguł oraz struktur programistycznych określający metodę oraz cel interakcji pomiędzy komponentami oprogramowania. Pojęcie interfejsu programowania aplikacji jest niezależne od warstwy implementacji systemu i może tyczyć się dowolnego rodzaju programu komputerowego. Interfejs API definiowany jest na poziomie kodu źródłowego poszczególnych fragmentów oprogramowania, a jego zadaniem jest dostarczenie wymaganych specyfikacji struktur programistycznych, a także protokołu, pozwalającego na ich wykorzystanie przez zewnętrzny komponent programowy.

API wykonane w technologii REST

RESTful API

Interfejs programowania aplikacji, bazujący zarówno w swojej strukturze, jak i funkcjonalności na zbiorze ścisłe określonych reguł dostarczanych w ramach metodologii REST. Reguły metodologii tej, implementowane są najczęściej w stosunku do interfejsów programowania aplikacji, które wykorzystują w kontekście zewnętrznej komunikacji protokół hipertekstowy. Podstawowa charakterystyka interfejsu programowania aplikacji opartego o zbiór reguł REST definiowana

jest poprzez bezstanowość transmisji danych, pojęcia zasobów i reprezentacji, a także cechę jednolitości interfejsu komunikacyjnego.

Kontroler

Controller

Struktura programistyczna (w przypadku języków programowania opartych o paradygmat obiektowy - klasa), której celem jest obsługa żądania dostarczonego od aplikacji klienckiej, zweryfikowania zgodności jego treści, a następnie przekierowanie wykonania programu do odpowiedniej metody warstwy logiki biznesowej. Po zakończeniu wszystkich operacji dotyczących omawianego zapytania, metoda dostępna w ramach klasy kontrolera formułuje odpowiedź zwracaną bezpośrednio do konsumenta żądania.

Serwis

Service

Struktura programistyczna (w przypadku języków programowania opartych o paradygmat obiektowy - klasa), której zadaniem jest realizacja obliczeń związanych z określonym obiektem, lub domeną, w ramach której określony obiekt się znajduje. Klasy serwisów (nazywane również klasami logiki biznesowej), stanowią centralny punkt przetwarzania w ramach interfejsów programowania aplikacji, a także pełnią rolę pośrednika pomiędzy komponentami odpowiedzialnymi za zarządzanie żądaniem (tj. metodami kontrolerów) oraz pozyskiwanie danych z określonych źródeł (tj. metodami repozytoriów).

Repozytorium

Repository

Struktura programistyczna (w przypadku języków programowania opartych o paradygmat obiektowy - klasa), której rolą jest wykonywanie operacji na obiektach modelu danych komunikując się bezpośrednio z wykorzystywanym systemem bazodanowym. Operacje zawarte wewnątrz metod klas repozytoriów mogą mieć postać kwerend lub komend definiowanych w języku zapytań dostarczany przez serwer bazodanowy, mogą operować na dostępnej w pamięci aplikacji strukturze danych, bądź też odwoływać się do zewnętrznego zbioru bazodanowego, manipulując instancjami klas zdefiniowanego wewnątrz aplikacji modelu. W ostatnim z omawianych przypadków, aktualizacje wartości w ramach encji modelu danych są identyfikowane przez mapper obiektowo-relacyjny, który generuje polecenia języka zapytań systemu bazodanowego, mające na celu synchronizację stanu danych aplikacji oraz zewnętrznego źródła informacji. Metody klas repozytoriów są wywoływane wewnątrz metod logiki biznesowej.

Mapper obiektowo-relacyjny

Object-relational mapper

Oprogramowanie, którego głównym zadaniem jest konwersja struktury klas modelu danych do fizycznej organizacji tabel w ramach systemu bazodanowego. Ponadto, mapper obiektowo-relacyjny dostarcza zbiór właściwości oraz metod stanowiących fasadę dla niskopoziomowych procedur dostępu do bazy danych, a także modyfikacji danych w niej zawartych.

Pamięć podręczna

Cache

Wydzielony fragment pamięci cechujący się szybkim czasem dostępu, wysoką przepustowością transmisji, a także ograniczonym okresem trwałego przechowywania danych. Pamięć ta, w kontekście webowego interfejsu programowania aplikacji, wykorzystywana jest w celu przechowywania wyników często realizowanych operacji, a także magazynowania uprzednio dostarczonych do klienta fragmentów odpowiedzi na żądania.

Przetwarzanie współbieżne

Concurrent Computing

Technika programistyczna oparta o wykorzystanie wielu współistniejących procesów oraz wątków, dostępnych w obrębie jednej aplikacji, a także odwołujących się do współdzielonych struktur danych. Poszczególne wątki stanowią elementy składowe pojedynczego procesu i są uruchamiane na tej samej centralnej jednostce przetwarzania. Procesor wykonuje operacje przełączania pomiędzy kontekstami poszczególnych wątków, dzięki czemu uzyskać można wrażenie, wykonania wielu spośród tych elementów w sposób równoległy. Z punktu widzenia interfejsu programowania aplikacji, który implementuje technikę przetwarzania współbieżnego, wyróżnić możemy punkty końcowe, które definiowane są jako osobne wątki aplikacji. Dlatego też, interfejs programowania aplikacji cechuje się dostępnością pomimo jednoczesnego przetwarzania wielu, niekiedy długotrwałych żądań klientów.

Algorytm metaheurystyczny

Metaheuristic algorithm

Technika projektowania algorytmów nie zapewniających gwarancji uzyskania optimum dla rozważanego problemu, jednakże pozwalająca na zbudowanie systemu, dostarczającego rozwiązanie złożonego zagadnienia w akceptowalnym czasie, a także uzyskiwanego przy wykorzystaniu akceptowalnej ilości zasobów sprzętowych. Algorytm metaheurystyczny, poza konwencjonalnymi regułami stosowanymi w ramach standardowych wzorców programowania, implementuje reguły rozwiązywania problemów oparte na losowości, bądź też wywnioskowane na podstawie zjawisk fizycznych.

Punkt końcowy interfejsu programowania aplikacji

API Endpoint

Punkt końcowy usługi sieci web definiuje jeden z końców kanału komunikacyjnego pomiędzy aplikacją kliencką a serwerową. W momencie interakcji interfejsu programowania aplikacji z odrębnym systemem, punkt styku dwóch usług sieciowych w ramach omawianej interakcji nazywany jest punktem końcowym. W odniesieniu do wewnętrznej struktury interfejsu programowania aplikacji, punkt końcowy wywołuje związaną z nim metodę klasy kontrolera, a samo powiązanie identyfikowane jest (w przypadku usługi sieciowej bazującej na protokole hipertekstowym i metodologii REST) poprzez nazwę zasobu, rodzaj metody, a także parametry żądania.

Żądanie realizowane w ramach usługi protokołu hipertekstowego

HTTP Request

Struktura danych, wysyłana od aplikacji klienckiej (tj. aplikacji internetowej, przeglądarki, czy też programu klienta HTTP) w kierunku usługi sieciowej. Żądanie protokołu hipertekstowego charakteryzuje się jednoznacznie zdefiniowaną strukturą, uwzględniającą m.in. unikalny identyfikator zasobu, listę zdefiniowanych nagłówków, ciało żądania oraz jedną z dziewięciu dołączalnych metod HTTP.

Odpowiedź usługi protokołu hipertekstowego

HTTP Response

Struktura danych, wysyłana przez usługę sieciową w kierunku aplikacji klienckiej. Odpowiedź HTTP, ma na celu poinformowanie klienta serwisu webowego o statusie realizacji, wysłanego przez niego uprzednio żądania. Podstawowymi elementami odpowiedzi usługi protokołu hipertekstowego są: ciało odpowiedzi (zdefiniowane najczęściej z wykorzystaniem notacji JSON lub języka XML), kod odpowiedzi (liczba determinująca stan wykonania żądania), a także zbiór informacji nagłówkowych dotyczących typu danych zawartych w odpowiedzi, czy też fizycznych informacji o serwerze usługi sieciowej.

Kod odpowiedzi usługi protokołu hipertekstowego

HTTP Response Code

Liczba determinująca status realizacji żądania wysłanego przez aplikację kliencką. Kod odpowiedzi stanowi jedną z wymaganych składowych dotyczących standardowego rezultatu zwartego w ramach usługi opartej o protokół hipertekstowy. Wyróżnić możemy pięć kategorii kodów odpowiedzi, niosących ze sobą odmienną informację. Kategoriami tymi są: kody informacyjnej odpowiedzi (100-199), kody poprawnej odpowiedzi (200-299), kody wiadomości o przekierowaniu (300-399), kody błędu aplikacji klienckiej (400-499), oraz kody błędu aplikacji serwerowej (500-599).

Czas odpowiedzi usługi protokołu hipertekstowego

HTTP Response Time

Wyrażony w milisekundach, przedział czasu od momentu otrzymania żądania wygenerowanego przez aplikację kliencką, do chwili zwrócenia rezultatu wykonywanych przez usługę sieciową obliczeń. Liczba ta, stanowi jedną z wartości pomiarowych, w kontekście efektywności działania interfejsu programowania aplikacji.

Obiektowa notacja JavaScript

JavaScript Object Notation - JSON

Format definicji, reprezentacji, a także wymiany danych w postaci obiektów niezależny od określonego języka programowania. Obiektowa notacja JavaScript jest powszechnie wykorzystywana jako format komunikatów przekazywanych pomiędzy interfejsami programowania aplikacji a systemami klienckimi. W odróżnieniu od języka reprezentacji danych opartego o znaczniki *Extensible Markup Language - XML*, obiektowa notacja JavaScript cechuje się mniejszym rozmiarem przesyłanych obiektów (poprzez redukcję liczby metadanych), jednolitym standardem

niezależnym od technologii, a także brakiem przechowywania informacji o typie poszczególnej wartości zadanego obiektu.

Testy wzorcowe

Benchmark

Rodzaj ewaluacji oprogramowania, której zadaniem jest określenie referencyjnego poziomu wydajności dla testowanego systemu. Metryki, uzyskane w ramach testów wzorcowych, mogą zostać wykorzystane jako wartości ograniczeń względem testów obciążeniowych oraz przeciążeniowych.

Testy dymne

Smoke testing

Metoda testowania oprogramowania, której celem jest sprawdzenie poprawności funkcjonowania poszczególnych elementów systemu. Testy dymne, wykonywane są przed testami wydajnościowymi, po to aby upewnić się co do braku błędów implementacyjnych w ramach analizowanego oprogramowania.

Testy wydajności podstawowej

Baseline performance testing

Metoda ewaluacji oprogramowania, pozwalająca na weryfikację działania systemu w warunkach analogicznych do realiów standardowego działania. Na podstawie testów wydajności podstawowej, określić można wartości metryk, które będą miały zastosowanie jako punkt odniesienia dla kolejnych rodzajów testów. Ponadto, wykorzystując standard pomiaru wydajności aplikacji internetowych (taki jak np. APDEX), wartości uzyskane w ramach ewaluacji podstawowych, mogą posłużyć w celu określenia punktów satysfakcji, tolerancji oraz frustracji.

Testy obciążające

Load testing

Rodzaj testów, które mają na celu określenie maksymalnego poziomu natężenia operacji, jakie mogą być generowane w kierunku oprogramowania. W kontekście niniejszej pracy, operacjami tymi są żądania wysyłane do interfejsu programowania aplikacji. Kluczowym aspektem testu obciążeniowego jest zdefiniowanie progu obciążenia aplikacji, powyżej którego system jest nie w stanie generować poprawnych odpowiedzi w akceptowalnym czasie.

Testy przeciążeniowe

Stress testing

Metoda ewaluacji oprogramowania, w ramach której natężenie operacji generowanych w kierunku testowanego oprogramowania zwiększone jest ponad ustalony próg tolerancji. Celem testu przeciążeniowego jest obserwacja sposobu działania systemu, w momencie, w którym nie jest on w stanie przetwarzać otrzymywanych żądań w sposób poprawny.

Asercja

Assertion

Wyrażenie typu prawda/fałsz, zdefiniowane w dowolnym miejscu programu, które przyjmuje wartość prawdziwą w momencie spełnienia hipotezy zawartej w ramach określonego przypadku testowego. Praktyczne podejście do procesu testowania funkcjonalności oprogramowania, sprawdza się do definiowania hipotez oraz ciągów operacji w kontekście przypadków testowych, a następnie weryfikacji tych hipotez z wykorzystaniem asercji.

2.2. Interfejsy programowania aplikacji

Webowy interfejs programowania aplikacji to usługa sieciowa, której celem jest realizacja zadań zleconych przez oprogramowanie klienta. Zadania te, dotyczą operacji wykonywanych w kontekście określonych zasobów. Wyróżnić możemy operacje zwane zapytaniami (tj. dotyczące pozyskiwania danych z ich źródeł), a także komendami (tj. związane z wykonywaniem operacji na danych).

Interfejsy API, budowane są z wykorzystaniem protokołu HTTP, dlatego też w ich kontekście możemy mówić o komunikacji bezstanowej definiującej pojęcia żądania oraz odpowiedzi. W związku z charakterystyką protokołu hipertekstowego, zarówno żądanie jak i odpowiedź cechuje się regularną strukturą zawierającą predefiniowane elementy.

Żądanie protokołu http wysyłane jest od aplikacji klienta do interfejsu API. Podstawową składową tego polecenia stanowi unikalny identyfikator zasobu URI (*ang. Uniform Resource Identifier*), na podstawie którego możliwe jest określenie fragmentu dziedziny obsługiwanej modelu danych. Informacja ta jednak, nie jest wystarczająca w kontekście realizacji jednej z funkcjonalności, zdefiniowanych w ramach API. Żądanie klienta, musi zostać uzupełnione o jedną z dziewięciu ustalonych metod http, obsługiwany wersję protokołu, a także zbiór linii nagłówkowych. Opcjonalnie, informacja wysyłana w kierunku interfejsu, może zostać wzbogacona o zawartość tekstuową określającą całe żądanie (*ang. Request body*). Taki zbiór informacji, pozwala na jednoznaczną identyfikację fragmentu kodu programu, który ma zostać wykonany wewnątrz interfejsu programowania aplikacji. W tabelach 2.1 oraz 2.2 przedstawiono kolejno listę zdefiniowanych metod protokołu hipertekstowego wraz z wyjaśnieniem ich przeznaczenia, a także zbiór najczęściej wykorzystywanych linii nagłówkowych, w kontekście realizacji żądań.

Tab. 2.1: Zbiór dozwolonych metod protokołu hipertekstowego

Nazwa metody	Opis
GET	Pozyskanie danych dotyczących pojedynczej instancji określonego zasobu lub grupy instancji z opcjonalnym uwzględnieniem warunków kwalifikacji poszczególnej instancji do grupy.
POST	Definiowanie nowej instancji dotyczącej określonego typu zasobu. Przy zastosowaniu metody POST, wymagane jest zdefiniowanie ciała żądania, jako części składowej generowanej instrukcji.
PUT	Aktualizacja pełni zawartości instancji występującej w ramach odwołania się do określonego zasobu. Przy zastosowaniu metody PUT, wymagane jest zdefiniowanie ciała żądania, jako części składowej generowanej instrukcji.
DELETE	Usunięcie istniejącej instancji dotyczącej określonego typu zasobu.

Tab. 2.1: Zbiór dozwolonych metod protokołu hipertekstowego

Nazwa metody	Opis
PATCH	Aktualizacja fragmentu zawartości instancji występującej w ramach odwołania się do określonego zasobu. Przy zastosowaniu metody PATCH, wymagane jest zdefiniowanie ciała żądania, jako części składowej generowanej instrukcji.
HEAD	Pozyskanie zbioru linii nagłówkowych, które byłyby dostarczone wraz z ciałem odpowiedzi w ramach żądania wykorzystującego metodę GET. Wygenerowanie żądania HEAD umożliwia określenie charakteru danych, przed ich ewentualnym pozyskaniem.
OPTIONS	Pozyskanie informacji dotyczących charakterystyki oraz struktury serwera. Definiując żądanie typu OPTIONS, klient może dowiedzieć się o dopuszczalnych metodach HTTP obsługiwanych przez serwer, czy też uzyskać informacje o nazwie serwera oraz wykorzystywany systemie operacyjnym.
CONNECT	Ustanowienie dwukierunkowej komunikacji pomiędzy klientem a serwerem. W przypadku realizacji komunikacji szyfrowanej, żądanie typu CONNECT pozwala na zestawienie zabezpieczonego tunelu pomiędzy hostami.
TRACE	Wygenerowanie komunikatu diagnostycznego w ramach pętli zwrotnej, którego celem jest osiągnięcie każdego z hostów, biorących udział w komunikacji.

Tab. 2.2: Zbiór najczęściej wykorzystywanych linii nagłówkowych w kontekście żądania protokołu hipertekstowego

Linia nagłówkowa	Znaczenie	Dopuszczalna zawartość
accept	Typ zawartości, którą jest w stanie przetwarzać aplikacja kliencka	Identyfikator typu MIME (ang. <i>Multipurpose Internet Mail Extensions</i>) lub zapis */* oznaczający dowolną zawartość
accept-encoding	Sposób kodowania znaków, rozumiany przez stronę klienta	Zbiór formatów kodowania zdefiniowany w ramach rejestru formatów IANA
accept-language	Język naturalny, preferowany przez stronę kliencką	Pojedyncza wartość reprezentująca określony kraj lub region, bądź też lista niszcznych wartości wraz z parametrem istotności poszczególnego kodu lokalizacji
content-length	Długość ciała żądania wyrażona w bajtach	Liczba naturalna
content-type	Format zawartości ciała żądania	Identyfikator typu MIME wraz ze sposobem kodowania wiadomości

Tab. 2.2: Zbiór najczęściej wykorzystywanych linii nagłówkowych w kontekście żądania protokołu hipertekstowego

Linia nagłówkowa	Znaczenie	Dopuszczalna zawartość
cookie	Zbiór informacji pozwalających na wprowadzenie oraz utrzymanie stanowego charakteru transmisji	Zestaw par klucz-wartość, gdzie klucz jest wartością tekstową, a wartość przyjmuje postać dowolną
origin	Informacja determinująca pochodzenie żądania	Ciąg tekstowy składający się z nazwy protokołu, nazwy hosta oraz numeru portu
user-agent	Specyfikacja techniczna oprogramowania klienta	Ciąg znaków zawierający informacje o nazwie produktu, jego wersji, platformie sprzętowej, czy też systemie operacyjnym

Po wykonaniu kodu programu przypisanego do określonego rodzaju polecenia generowanego przez aplikację kliencką, z interfejsu programowania aplikacji zwracana jest odpowiedź na żądanie (*ang. HTTP response*). Analogicznie do instrukcji realizacji danej czynności, także odpowiedź dotycząca statusu jej wykonania jest ustrukturyzowana zgodnie z wytycznymi zawartymi w definicji protokołu hipertekstowego. W ramach rezultatu zwróconego przez API wyróżnić należy: adres docelowy klienta, kod statusu, ciało odpowiedzi, a także zbiór linii nagłówkowych. Informacja zawarta w ramach kodu statusu, determinuje powodzenie realizowanej operacji, a treść dostarczanych linii nagłówkowych, może zostać wykorzystana w celu wnioskowania o charakterystyczce odbywającej się komunikacji. Ciało odpowiedzi powinno zawierać dane dotyczące definiowanego w ramach identyfikatora żądania zasobu, w przypadku żądań wykorzystujących metodę GET. W kontekście pozostałych żądań, zgodnie z wytycznymi dokumentu RFC (*ang. Request For Comments*) o numerze 7230, powinno ono posiadać charakter informacji pomocniczej, bądź też pozostać puste [9]. W ramach tabel 2.3 oraz 2.4, wymienione zostały kolejno: zbiór najczęściej zwracanych linii nagłówkowych w kontekście odpowiedzi na żądanie, a także przedziały liczbowe dla kodów statusu odpowiedzi, wraz z ich semantyką.

Przedstawiony w niniejszy sposób interfejs programowania aplikacji scharakteryzować należy jako deterministyczny system wejściowo-wyjściowy. Ponadto, należy zauważyc, że w ramach systemu tego, występuje zjawisko inercji, powodowane koniecznością realizacji zdefiniowanego w ramach API kodu programu. Na podstawie tego założenia, ewaluację działania oraz wydajności interfejsu programowania aplikacji przeprowadzić można poprzez wprowadzanie określonego wejścia (tj. generowanie żądania) oraz obserwację wartości zwróconej na wyjściu (tj. uzyskana odpowiedź).

Proces przetwarzania żądania wewnątrz interfejsu API

Po uzyskaniu żądania otrzymanego od strony klienta, zadaniem interfejsu programowania aplikacji jest wybór określonej klasy kontrolera, a także zawartej w niej metody. Każda z klas kontrolerów stworzona jest w celu obsługi operacji związanych z konkretnym zasobem, a poszczególna metoda tej klasy implementuje zachowanie które ma zostać wywołane w kontekście dostarczonego typu oraz identyfikatora polecenia.

Wewnątrz metody klasy warstwy kontrolerów, wywoływane zostają operacje zdefiniowane w usługach warstwy biznesowej. Usługi te, realizowane mogą być zarówno wewnątrz api jak i stanowić odrębny system internetowy. Klasy warstwy logiki biznesowej, zwane serwisami, zło-

Tab. 2.3: Zbiór najczęściej zwracanych linii nagłówkowych w kontekście odpowiedzi protokołu hipertekstowego

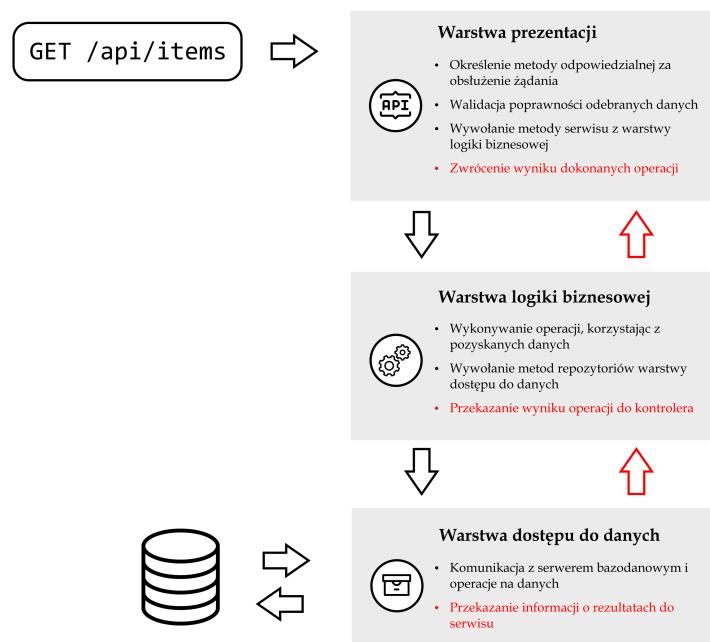
Linia nagłówkowa	Znaczenie	Dopuszczalna zawartość
access-control-allow-credentials	Określenie, czy odpowiedź serwera ma być osiągalna z kodu JavaScript aplikacji klienckiej, w momencie gdy nagłówek żądania dotyczący poświadczeń, zezwala na ich dołączenie	Wartość prawda/fałsz
access-control-allow-origin	Informacja dotycząca pochodzenia klienta, który może ubiegać się o otrzymanie odpowiedzi od serwera	adres hosta klienckiego lub symbol gwiazdki oznaczający zezwolenie dla wszystkich hostów
cache-control	Dane konfiguracyjne dotyczące obsługi pamięci podręcznej	Zbiór par klucz-wartość określających zachowanie pamięci cache w kontekście określonej komunikacji
content-length	Długość ciała odpowiedzi wyrażona w bajtach	Liczba naturalna
content-type	Format zawartości ciała odpowiedzi	Identyfikator typu MIME wraz ze sposobem kodowania wiadomości
cross-origin-resource-policy	Polecenie ignorowania żądań realizowanych pomiędzy źródłami bądź witrynami w kontekście określonego zasobu	Wartość prawda/fałsz
expires	Data wygaśnięcia ważności nieważszej odpowiedzi	Określona data
server	Nazwa hosta dostarczającego odpowiedź klientowi	Ciąg znaków

Tab. 2.4: Zbiór kodów statusu odpowiedzi protokołu hipertekstowego

Przedział liczbowy	Semantyka w kontekście odpowiedzi
100 - 199	Zbiór kodów informacyjnych - żądanie jest aktualnie przetwarzanie
200 - 299	Zbiór kodów poprawnej odpowiedzi - wysłane żądanie zostało zrealizowane poprawnie
300 - 399	Zbiór kodów przekierowań - istnieć może wiele akceptowalnych odpowiedzi dla żądania bądź realizacja określonej operacji wymusza odwołanie się pod adres identyfikujący odmienny zasób
400 - 499	Zbiór kodów błędu po stronie klienta - wygenerowane żądanie zawiera błędy, oczekiwany zasób nie istnieje, klient nie jest uwierzytelny lub nie posiada określonego poziomu uprawnień
500 - 599	Zbiór kodów błędu po stronie serwera - pomimo poprawnej struktury wygenerowanego żądania, serwer nie jest w stanie zrealizować przydzielonej mu operacji

żone są z metod, których głównym celem jest weryfikacja poprawności otrzymanych informacji w kontekście obsługiwanych zasobów, a także pozyskiwanie danych oraz wykonywanie operacji na nich, poprzez odwoływanie się do metod warstwy dostępu do danych.

Zbiór klas warstwy dostępu do danych, stanowi ostatni z logicznych poziomów, definiowanych w ramach architektury API. Fragmenty kodu zdefiniowane w tej warstwie, zwane repozytoriami, mają za zadanie obsłużyć komunikację pomiędzy interfejsem programowania aplikacji, a określonym źródłem danych. Ponadto, metody klas repozytoriów, dostarczają warstwie logiki biznesowej interfejs operacji na danych. Dzięki temu, żądanie może być przetwarzane od warstw najwyższych (tj. warstwy kontrolerów) do warstwy najniższej (tj. warstwy dostępu do danych), natomiast odpowiedź na żądanie jest konsolidowana w kierunku odwrotnym [18]. Na ilustracji 2.1 przedstawiono przepływ informacji wewnątrz interfejsu API, od momentu wygenerowania żądania do chwili uzyskania odpowiedzi.



Rys. 2.1: Proces przetwarzania żądania wewnątrz interfejsu API

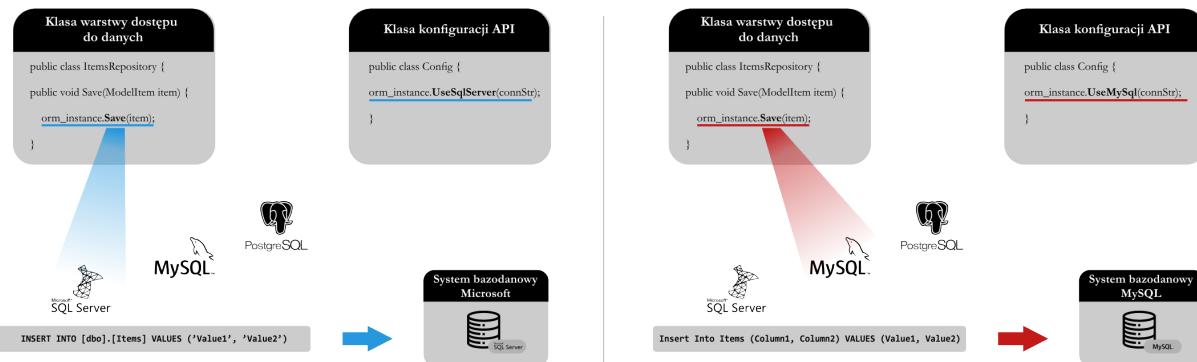
Konwersja obiektowo-relacyjna

W celu uproszczenia procesu pozyskiwania oraz modyfikacji danych z zewnętrznych źródeł, a także unifikacji sposobu interakcji z nimi, w ramach interfejsów programowania aplikacji, powszechnie wykorzystywane jest oprogramowanie zwane mapperem obiektowo-relacyjnym (*ang. Object-Relational Mapper*). Założeniem oprogramowania tego, jest zdefiniowanie warstwy abstrakcji pomiędzy interfejsem programowania aplikacji a językiem programowania bądź zbiorem poleceń, wykorzystywanym w ramach obsługi źródła danych.

Podstawowe składowe oprogramowania typu ORM to jednolity interfejs operacji na zbiorze danych, klasy kontekstu bazodanowego, a także metody obsługi komunikacji z bazą danych.

Dzięki wprowadzeniu jednolitego interfejsu operacji na danych, niezależnie od źródła informacji z jakim komunikuje się API, wydanie konkretnego polecenia do dowolnego systemu bazodanowego równoznaczne jest z każdorazowym wywołaniem funkcji o takiej samej sygnaturze. Stosując takie podejście, konstruktor interfejsu programowania aplikacji nie staje się uzależniony od źródła danych z którym pracuje. Ponadto, istnieje możliwość zamiany lub połączenia

dodatkowego systemu bazodanowego, a operacja ta, nie wpływa w jakikolwiek sposób na działanie interfejsu API. Niniejsza zależność została zilustrowana na rysunku 2.2



Rys. 2.2: Zasada działania oprogramowania mappera obiektowo-relacyjnego w kontekście jednolitego interfejsu operacji na zbiorze danych

Dystynktywnym elementem oprogramowania mappera obiektowo-relacyjnego jest klasa kontekstu bazodanowego. Klasa ta, jest kontenerem struktur w ramach których wyróżnić możemy zbiory elementów modelu danych, a także konfigurację poszczególnych ich właściwości. Podstawową ideą omawianej konwersji dziedziny obiektowej do domeny relacyjnej jest zdefiniowanie zbioru klas, opisujących wykorzystywane zasoby, a następnie odwzorowanie ich w relacyjnym modelu danych, obsługiwany przez wybrany system bazodanowy. Klasa kontekstu pozwala na określenie, które spośród struktur danych zdefiniowanych w ramach API powinny zostać rzucone na obiekty tabel generowanych w obrębie bazy danych. Ponadto, dla właściwości każdej z klas modelu danych, zdefiniować należy konfigurację, która zostanie przetransformowana do modelu relacyjnego. W zakresie klasy kontekstu bazy danych, opisywane są także relacje, jakie mają zostać wygenerowane pomiędzy poszczególnymi elementami modelu.

W celu nawiązania, utrzymania, a także zakończenia komunikacji z zewnętrznym źródłem danych, oprogramowanie ORM wykorzystuje klasy zwane konektorami. Klasy te, dostarczają przejrzysty interfejs obsługi połączenia, który następnie jest opakowywany w zunifikowany interfejs, dostępny bezpośrednio dla twórcy API [35].

Uwierzytelnienie oraz autoryzacja

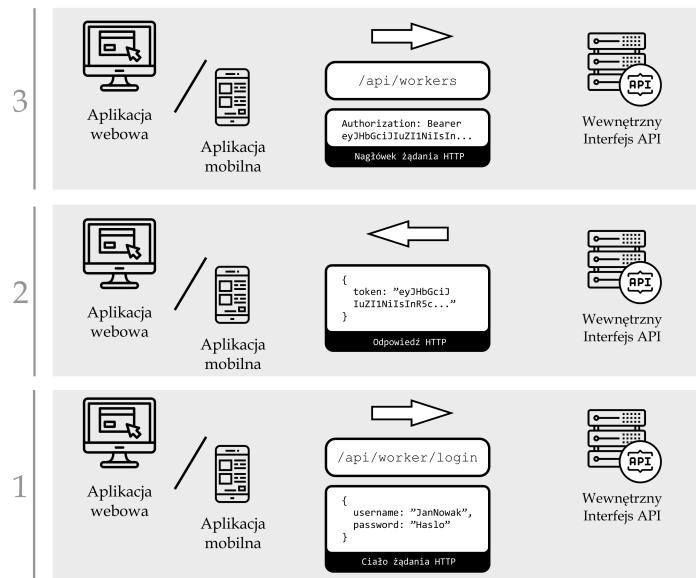
Proces uwierzytelnienia oraz autoryzacji użytkownika odwołującego się do interfejsu programowania aplikacji, przedstawić należy w trzech następujących krokach.

Pierwszym z nich, jest wygenerowanie żądania odwołującego się do punktu końcowego odpowiedzialnego za obsługę uwierzytelnienia wewnętrz API. Żądanie to, musi posiadać ciało, zawierające informacje poświadczające o konkretnym użytkowniku. Najczęściej, informacją tą, jest nazwa użytkownika oraz hasło.

Następnie, dostarczone referencje są analizowane przez mechanizmy uwierzytelniania implementowane w ramach API. W rezultacie tych operacji, zwrócona zostaje pozytywna odpowiedź zawierająca token autoryzujący bądź też negatywna, posiadająca w sobie informację o błędzie uwierzytelnienia klienta.

Strona kliencka może autoryzować dysponowane operacje przed interfejsem programowania aplikacji, uwzględniając w ramach linii nagłówkowej żądania token uwierzytelniający. Dostarczona w ten sposób informacja, pozwala na identyfikację użytkownika w ramach interfejsu API, a także na określenie przypisanego użytkownikowi poziomu uprawnień. W ramach struktury tokenu, zawarta jest także informacja o jego czasie ważności, dlatego też, procedura uwierzytelniania musi być regularnie ponawiana [19].

Na rysunku 2.3, zilustrowany został proces uwierzytelnienia i autoryzacji aplikacji klienta przez interfejsem programowania aplikacji.



Rys. 2.3: Proces uwierzytelnienia oraz autoryzacji użytkownika przed interfejsem API

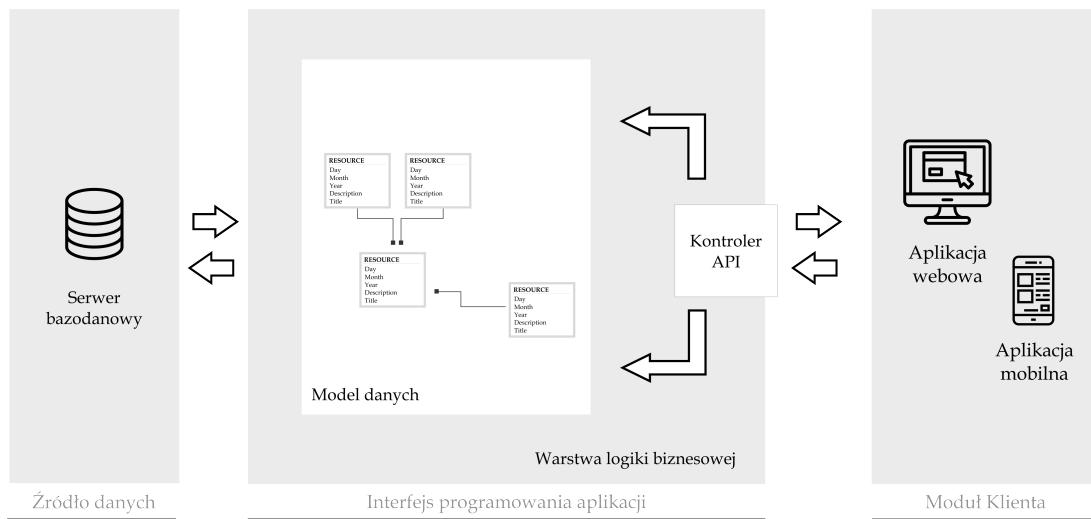
Separacja zapytań oraz komend w kontekście odwołań do źródeł danych

Wraz z rosnącą liczbą żądań obsługiwanych w ramach zaawansowanych interfejsów programowania aplikacji, zauważone zostało zjawisko asymetrii w kontekście typów wiadomości generowanych przez klientów. Zapytania dotyczące pozyskiwania danych z API realizowane jest z nieporównywalnie większą częstością niż operacje ich modyfikacji. Dlatego też, zdefiniowany został wzorzec projektowy dotyczący separacji zapytań oraz komend generowanych względem usługi sieciowej (*ang. Command Query Responsibility Segregation*).

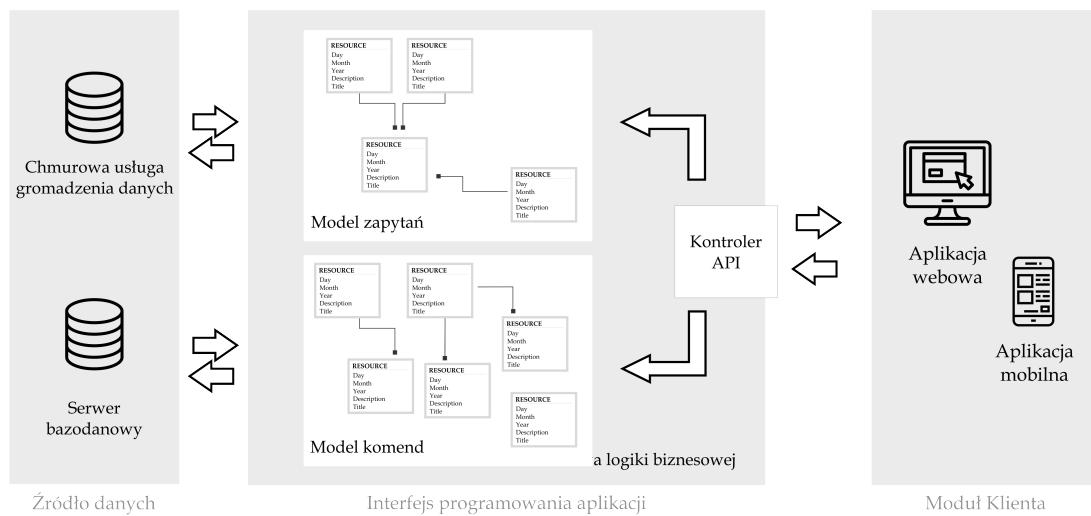
Zastosowanie przedstawionego powyżej wzorca projektowego wiąże się z koniecznością budowy dwóch osobnych modeli danych. Pierwszy z nich, wykorzystywany jest w kontekście odczytu informacji. Na modelu tym, dokonywana jest najczęściej operacja optymalizacji, której celem jest redukcja rozmiaru składowych modelu, a także szybkości przetwarzania bardziej złożonych struktur będących jego częścią. Drugi z modeli danych, znajduje zastosowanie w aspekcie modyfikacji określonych zasobów. Biorąc pod uwagę standardowy sposób eksploatacji interfejsu programowania aplikacji, model ten cechować się może niższą wydajnością. W zależności od specyfiki określonej usługi sieciowej, optymalizowany może być albo model odczytu, albo też model zapisu. Ponadto, implementując wzorzec projektowy separacji zapytań oraz komend, wykorzystać można dwa osobne zewnętrzne źródła danych, przystosowane do wydajniejszego wykonywania określonego typu operacji, bądź też dostosowane do obsługi większego ruchu sieciowego.

Niewątpliwy zaletami, wynikającymi z zastosowania opisywanego wzorca projektowego są: zwiększenie efektywności operacji realizowanych z dużą częstotliwością, możliwość korzystania z osobnych źródeł danych dla operacji odczytu oraz zapisu, zachowanie zasady pojedynczej odpowiedzialności (*ang. Single Responsibility Principle*) względem klas logiki biznesowej API, a także redukcja liczby wstrzykiwanych zależności (*ang. Dependency Injection*) w ramach klas kontrolerów interfejsu [31].

Na ilustracjach 2.4 oraz 2.5 przedstawiono kolejno schemat przetwarzania żądań wewnętrznych API z uwzględnieniem wzorca CQRS, a także przy wykorzystaniu pojedynczego modelu danych.



Rys. 2.4: Schemat przetwarzania żądania przez interfejs API dla architektury z jednym modelem danych



Rys. 2.5: Schemat przetwarzania żądania przez interfejs API dla architektury wykorzystującej wzorzec projektowy CQRS

Odwołując się do ilustracji 2.5 należy zaznaczyć, że jest to jedna z możliwości implementacji wzorca CQRS, uwzględniająca wykorzystanie odseparowanych źródeł danych. Architektura separacji zapytań oraz komend w kontekście żądań wysyłanych do interfejsu API, może być także wprowadzona przy wykorzystaniu pojedynczego systemu bazodanowego.

Konwencja REST

Niezależnie od struktury wewnętrznej omawianych usług sieciowych, współczesne interfejsy programowania aplikacji projektowane są tak, aby ich zewnętrzna warstwa (tj. widziana z perspektywy aplikacji klienckiej) cechowała się jednolitą kompozycją.

Jednym z najpopularniejszych sposobów zapewnienia jednolitego interfejsu komunikacyjnego pomiędzy klientami i usługami sieciowymi, przetwarzającymi informacje z wykorzysta-

niem protokołu HTTP, jest konwencja oraz styl architektoniczny REST (*ang. Representational State Transfer*).

Konwencja ta, definiuje zbiór zasad dotyczących m.in. zachowania usługi sieciowej w kontekście przetwarzania żądania konkretnego typu, struktury i elementów odpowiedzi na określone żądanie, semantyki wykorzystywanych statusów rezultatu przetwarzania, bezstanowego charakteru komunikacji, czy też syntaktyki odwołań do poszczególnych punktów końcowych.

W kontekście stopnia implementacji stylu architektonicznego REST w ramach interfejsu programowania aplikacji, wprowadzić należy pojęcie modelu dojrzałości Richardsona (*ang. Richardson Maturity Model*). Pojęcie to, definiuje cztery poziomy przystosowania interfejsu API do omawianej w niniejszej sekcji konwencji.

W odniesieniu do poziomu zerowego, powinnością interfejsu programowania aplikacji jest udostępnienie usług w ramach pojedynczego adresu sieciowego, niezależnie od wykorzystywanych metod HTTP. Struktura żądania klienckiego, w sposób jednoznaczny dostarczać ma informację na temat wykonywanego wewnątrz usługi sieciowej działania.

Zasada poziomu pierwszego, odnosi się do charakterystyki interfejsu API jako usługi zorientowanej na zasoby. Niezależnie od czynności, jaka ma zostać wykonana przez omawianą usługę sieciową, opis tej czynności wskazywać ma na zasób którego ona dotyczy.

Reguła stanowiąca definicję poziomu trzeciego, związana jest z semantyką poszczególnych typów żądań protokołu hipertekstowego. Żądanie o takim samym adresie sieciowym, pełnić powinno odmienną rolę, w zależności od rodzaju żądania HTTP.

Ostatnim z poziomów dojrzałości interfejsu programowania aplikacji opartego o konwencję REST jest reguła HATEOAS (*ang. Hypertext As The Engine Of Application State*). Reguła ta, definiuje interfejs API jako źródło informacji dotyczącej obsługi stanu całego systemu internetowego (tj. usługi sieciowej wraz z aplikacjami klienckimi). Klient, po uzyskaniu odpowiedzi serwera na żądanie, powinien na podstawie zawartości tej odpowiedzi móc zdefiniować przyszłe czynności, które wolno mu wykonać [38].

2.3. Testowanie oprogramowania

Aspekt badawczy niniejszej pracy, związany jest z realizacją procesu ewaluacji wydajności interfejsów programowania aplikacji, pod kątem wykorzystania odmiennych środowisk implementacyjnych oraz uruchomieniowych. Proces ten, jest tylko jednym z wielu elementów domeny testowania oprogramowania, której charakterystyka uwzględnia zbiór sztywno zdefiniowanych reguł cechujących się wysokim poziomem sformalizowania. W następnych sekcjach niniejszego akapitu dokonane zostało wprowadzenie dotyczące zagadnienia ewaluacji oprogramowania, narysowane zostały zasady testowania systemów informatycznych, zdefiniowano taksonomię technik testowania, a także omówiono proces przeprowadzania ewaluacji wydajności usługi sieciowej jaką jest interfejs programowania aplikacji.

Wprowadzenie do zagadnienia ewaluacji oprogramowania

Ewaluacja poszczególnych składowych tworzonego oprogramowania jest niezbędną częścią procesu budowy systemu informatycznego, niezależnie od jego charakterystyki, czy też wykorzystywanej do jego budowy technologii. W rozumieniu ogólnym, proces testowania często sprowadzany jest do zbioru dwóch czynności. Czynnościami tymi są uruchamianie oprogramowania, a także eksploracja jego funkcjonalności w celu dostrzegania tych, w ramach których zauważać można niezgodność ich działania w stosunku do specyfikacji. Takie wnioskowanie jednak jest niepełne, i uwzględnia ono tylko jeden z etapów składających się na cały proces testowania. Dziedzinę ewaluacji cech programów komputerowych, poszerzyć należy ponadto o takie ele-

menty jak: planowanie testów, wybór kryteriów oceny oprogramowania, nadzór oraz kontrolę realizacji badań, projektowanie przypadków testowych, czy też analizę spełnienia ustalonych kryteriów zakończenia.

Wyróżnić możemy znaczącą liczbę definicji testowania oprogramowania, a każda z nich wprowadza inny poziom szczegółowości. Ponadto, wiele spośród formułowanych pojęć nawiązuje do różnych aspektów omawianego procesu. Zgodnie z jedną z najbardziej generycznych definicji, wprowadzoną przez Hetzla w publikacji [13], proces testowania oprogramowania określić należy jako zbiór wszystkich czynności, które nakierowane są na weryfikację atrybutów i właściwości programu, a także sprawdzenie tego, czy określony system spełnia założone wymagania. Definicja ta, względem wielu innych popularnych sformułowań dotyczących ewaluacji oprogramowania, uwzględnia możliwość zastosowania statycznych technik testowania. Ponadto, jej autor bierze pod uwagę fakt, że w ramach procesu ewaluacji, oceniany powinien być każdy z artefaktów tworzonych w ramach systemu, a nie tylko i wyłącznie kod źródłowy programu.

Taksonomia technik testowania

Jako jedno z podstawowych kryteriów podziału technik testowania oprogramowania, wskazać należy rodzaj czynności wykonywanej przez stronę testującą, której realizacja prowadzi do uzyskania charakterystyki programu poddanego ewaluacji. Według kryterium tego, wyróżnić należy statyczne oraz dynamiczne techniki testowania.

Pierwsze, spośród przytoczonych metod, opierają się na analizie artefaktów oprogramowania (takich jak m.in.: kod źródłowy, specyfikacja, dokumentacja, czy też lista wymagań) bez ich wykonywania. Jako praktyczne przykłady przedstawionej techniki, zdefiniować należy: generowanie metryk kodu źródłowego programu, analizę przepływu sterowania, formalne dowodzenie poprawności działania, a także interpretację grafów wywołań.

Metody dynamiczne natomiast, związane są z weryfikacją właściwości poszczególnych elementów systemu informatycznego w trakcie jego wykonywania. Ten rodzaj testowania, nie uwzględnia formalnych struktur liczbowych, jakimi są grafy przepływu sterowania, czy też metryki kodu źródłowego programu. Zorientowany jest on, na odbiór systemu z perspektywy korzystającego z niego klienta.

Innym z rozważanych kryteriów podziału technik testowania jest ich umiejscowienie według określonego fragmentu procesu wytwarzanego. W nawiązaniu do tego aspektu, zdefiniować należy pojęcie poziomu testów, które jest powiązaniem sposobu ewaluacji oprogramowania z etapem jego realizacji. Istotą rozróżniania danych poziomów testów jest założenie różnorodności celów testowania, a także testowanych obiektów, określanych w kontekście każdej z warstw. W odniesieniu do najbardziej popularnych systematyk poziomów testowania wyróżnić można następujące elementy:

- testy jednostkowe (zwane także modułowymi lub testami komponentów)
- testy integracyjne
- testy systemowe
- testy akceptacyjne

Pierwszy z poziomów, dotyczy znajdowania niezgodności specyfikacyjnych w obrębie logicznie oddzielonych jednostek oprogramowania (*ang. Software Units*). Każda z jednostek, powinna być testowana w izolacji od pozostałych elementów systemu. Ze względu na złożoność rozwiązań informatycznych, a także statystycznie wysoki współczynnik wzajemnych zależności modułowych, warunek ten często nie może zostać spełniony. W takich sytuacjach, aby dostarczyć zależność do testowanej jednostki, budowane są moduły zastępcze, imitujące poprawne zachowanie określonego fragmentu programu (*ang. Mocks*). Omawiany poziom testowania,

często postrzegany jest jako jeden z etapów procesu wytwórczego, szczególnie w ramach takich technik jak rozwój oprogramowania napędzany testowaniem (*ang. Test Driven Development*).

Celem kolejnego z poziomów testowania jest weryfikacja poprawności współdziaływania indywidualnych komponentów, a także prawidłowości funkcjonowania interfejsów definiowanych pomiędzy nimi. Przykładem współdziałania jednostek oprogramowania może być współpraca interfejsu programowania aplikacji, będącego systemem poddawanym analizie w ramach niniejszej pracy, a także określonego silnika bazodanowego. W zależności od liczności weryfikowanych powiązań pomiędzy poszczególnymi jednostkami oprogramowania, a także w odniesieniu do liczby samych modułów będących częścią testowanego fragmentu systemu, wyróżnić możemy testy małej oraz dużej skali integracji.

Na temat testów systemowych, należy mówić wtedy, gdy wszystkie z elementów rozwiązania informatycznego zostały ze sobą powiązane w sposób spójny. Celem testów, realizowanych w ramach tego poziomu, jest weryfikacja wysokopoziomowej funkcjonalności oprogramowania, a także wykonywanie scenariuszy ewaluacji systemu z poziomu regularnego użytkownika (*ang. End-to-End testing*).

Ostatnim z wymienionych poziomów ewaluacji są testy akceptacyjne. Przedmiotem oceny w ramach tego rodzaju testów jest gotowe rozwiązanie informatyczne w postaci komercyjnego produktu. Podmiot odpowiedzialny za realizację omawianych testów przygotowuje listę kryteriów akceptacji (*ang. acceptance criteria*), a następnie na podstawie obsługi testowanego rozwiązania, potwierdza lub odrzuca spełnienie każdego z nich. Celem omawianych ewaluacji nie jest znajdowanie błędów działania systemu, a nabranie zaufania co do jakości jego funkcjonalnych oraz niefunkcjonalnych atrybutów.

Wykonując testy definiowane w ramach kolejnych poziomów ewaluacji, weryfikowane zostają na początek funkcjonalne, a następnie niefunkcjonalne elementy testowanego systemu. Jako weryfikację elementów funkcjonalnych, rozumieć należy wszystkie te czynności, które podejmowane są w ramach wszystkich wymienionych powyżej poziomów testów, z wyjątkiem testów akceptacyjnych. Ewaluacja niefunkcjonalna natomiast, odnosi się tylko do ostatniego spośród wyróżnionych poziomów testowania.

Podział charakteryzujący przedmiot ewaluacji względem omówionych aspektów definiuje pojęcia testów funkcjonalnych oraz niefunkcjonalnych i w kontekście niniejszej pracy jest on podziałem kluczowym.

Badania przeprowadzone w ramach tej pracy posiadają charakter testów niefunkcjonalnych, a ich wykonanie poprzedzone jest weryfikacją funkcjonalną, której poprawność traktowana jest jako wymóg.

Ewaluacja wydajności interfejsów programowania aplikacji

Zgodnie z teorią przedstawioną w sekcji 2.2 niniejszej pracy interfejs programowania aplikacji postrzegać można jako deterministyczny system wejściowo-wyjściowy o charakterze dyskretnym. Takie podejście, w znaczący sposób ułatwia proces ewaluacji wydajności interfejsów API.

Definiowanie interfejsu API jako systemu pobudzanego pojedynczym wejściem, a także generującego pojedynczą wartość wyjściową, pozwala na wykorzystanie sposobu oceny wydajności zwanego testem czarnoskrzynkowym (*ang. Black-box testing*). W ramach tego rodzaju testu, określone kryterium ewaluacji wyliczane jest jako różnica wartości pomiaru na wyjściu systemu, względem tej, której kalkulacja nastąpiła na jego wejściu. Taki rodzaj testu, umożliwia wyliczenie metryki wydajności, bez konieczności przygotowywania systemu do przeprowadzenia procesu testowania.

Podstawowym kryterium oceny wydajności interfejsu programowania aplikacji jest czas odpowiedzi na żądanie. Metrykę tę, należy określić jako czas od momentu wygenerowania żądania

przez stronę klienta, do chwili uzyskania przez niego odpowiedzi. Zauważać należy również zależność czasu odpowiedzi na żądanie, zarówno od rozmiaru żądania jak i wielkości jego odpowiedzi. Ponadto, czynnikiem wpływającym na uzyskany rezultat pomiaru jest niewątpliwie przepustowość łącza sieciowego pomiędzy klientem a serwerem.

Aby rezultaty uzyskane w ramach oceny wydajności API mogły zostać postrzegane jako rzetelne, omówione powyżej czynniki muszą cechować się statyczną charakterystyką, bądź też zostać całkowicie wyeliminowane z procesu testowego. W ramach niniejszej pracy, rozmiar odpowiedzi generowanej przez interfejsy programowania aplikacji jest stały niezależnie od zastosowanej technologii, a także środowiska uruchomieniowego. Wynika to z zastosowania pojedynczego i ustrukturyzowanego modelu danych, który jest identyczny, niezależnie od API. W odniesieniu do zmienności prędkości łącza internetowego, aspekt ten został wyeliminowany poprzez przeprowadzanie testów w obrębie lokalnej sieci komputerowej, a także umiejscowienie interfejsów oraz systemów bazodanowych w ramach tej właśnie sieci.

Kolejnym z kryteriów oceny wydajności, uwzględnianym w ramach procesu testowania usług sieciowych jest poprawność uzyskanej odpowiedzi w odniesieniu do liczby klientów, równolegle generujących żądania. Kryterium to, charakteryzuje się silną korelacją z czasem odpowiedzi na pojedyncze zapytanie.

Uwzględniając oba opisane powyżej parametry, podstawowy schemat scenariusza badawczego dotyczącego ewaluacji wydajności API składa się z następujących testów:

- testy linii bazowej - pojedynczy klient generuje żadania w kierunku API w celu zdefiniowania średniego czasu odpowiedzi usługi w standardowych warunkach jej działania. W ramach niniejszej pracy, podczas wykonywania omawianego testu, zdefiniowane zostaną ponadto wartości współczynników satysfakcji, toleracji oraz frustracji, będące składowymi wskaźnika jakości APDEX. Wartości te, stanowią uogólnioną ocenę wydajności i posłużą jako punkt odniesienia dla kolejnych testów.
- testy obciążeniowe - uwzględniana zostaje zmienna liczba klientów generujących żądania, w kontekście której ustalane są średnie czasu odpowiedzi na żądanie, a także dokonywane zostaje odniesienie uzyskanego rezultatu do współczynników zdefiniowanych uprzednio w ramach miary APDEX.
- testy przeciążające - liczba klientów generujących żądanie zostaje dobrana w taki sposób, aby doprowadzić do obciążenia testowanej usługi, niepozwalającego na poprawne funkcjonowanie interfejsu programowania aplikacji. Ewaluacja ta, ma na celu znalezienie punktu krytycznego w kontekście działania testowanego oprogramowania.

Poza czarnoskrzynkowymi testami wydajności, cechującymi się omówioną powyżej strukturą, przeprowadzone mogą zostać ewaluacje efektywności działania poszczególnych fragmentów usługi sieciowej, w ramach których interfejs programowania aplikacji postrzegać należy w odmienny sposób, aniżeli jako system wejściowo-wyjściowy.

Przykładem oceny wydajności określonego fragmentu interfejsu programowania aplikacji może być ewaluacja modułu realizacji zaawansowanych operacji obliczeniowych, dostępnego z poziomu punktu końcowego API. W takim przypadku, oprogramowanie musi zostać dostosowane do przeprowadzenia procedury testowej, a metryka wydajności - powiązana z postępem realizacji obliczeń.

2.4. Wykorzystywane narzędzia i technologie

Zarówno w trakcie procesu implementacji badanych interfejsów programowania aplikacji, jak i procedurze przeprowadzenia badań pod kontem ich wydajności, wykorzystano obszerny zbiór sprawdzonych i powszechnie stosowanych rozwiązań technologicznych. W ramach niniejszej sekcji, opisane zostanie każde z nich.

C#

C# jest wieloparadygmatowym, a także nowoczesnym językiem programowania ogólnego przeznaczenia, charakteryzującym się bezpieczeństwem i niezawodnością w aspekcie typowania struktur danych. Pierwsza z wersji tego języka, stworzona została przez Andersa Hejlsberga w roku 1998. Od tamtej chwili, do momentu napisania niniejszej pracy, upublicznionych zostało 9 kolejnych, stabilnych wydań projektu C#. Każda z następnych wersji omawianego języka programowania wprowadzała zarówno usprawnienia w kontekście ekosystemu budowy i komplikacji programów źródłowych, jak i wzbogacała interfejs bibliotek funkcyjnych o kluczowe z punktu widzenia doświadczonego programisty rozwiązania. Do rozwiązań tych, zaliczyć należy między innymi: mechanizmy programowania współbieżnego, typy anonimowe, operatory zmiennych typów niezdefiniowanych, obsługę referencji, typy generyczne, czy też wyrażenia lambda.

W ramach niniejszej pracy, język C# wykorzystany został do implementacji jednego z dwóch zbiorów interfejsów programowania aplikacji. Ze względu na zastosowanie rozwiązań z zakresu przetwarzania współbieżnego (tj. operacji asynchronicznych oraz wielowątkowych) udostępnianych przez omawiany język programowania, interfejsy API realizowane w tej technologii mogą obsługiwać w sposób równoległy żądania pochodzące od wielu klientów, a także utrzymywać sekwencyjny charakter przetwarzanych procedur niezależnie od czasu ich wykonywania. Należy także zwrócić uwagę na mechanizm wewnętrznych usprawnień wydajnościowych implementowany w ramach kompilatora i uruchamiany w momencie tłumaczenia kodu języka do tzw. języka pośredniego (*ang. Intermediate Language*). Dzięki zastosowaniu przedstawionego mechanizmu, operacje zdefiniowane przez programistę mogą być modyfikowane w procesie komplikacji, tak aby nie wpływać na zaimplementowaną funkcjonalność, zwiększając jednocześnie wydajność generowanego programu [12].

.NET Core

.NET Core postrzegać należy jako środowisko budowy, komplikacji oraz wykonywania rozwiązań implementowanych w języku C#. Przedstawiana technologia stanowi podzbiór bibliotek, dzięki którym programista jest w stanie budować systemy różnorodnego przeznaczenia, a także uruchamiać je w wielu wspieranych środowiskach programowych. W przeciwieństwie do technologii .NET Framework będącej poprzednikiem .NET Core, aplikacje tworzone na bazie omawianej biblioteki mogą być wydawane nie tylko na system operacyjny Windows, ale także na systemy Linux oraz MacOS.

W ramach omawianego środowiska wykorzystywany zostaje komponent języka C# zwany biblioteką standardową (*ang. .NET Standard Library*). Biblioteka ta jest wspólna dla wielu środowisk uruchomieniowych, a zawarte w niej funkcjonalności, traktować należy jako metody ogólnego przeznaczenia.

Ponadto, środowisko .NET Core, w ramach procesu budowy i komplikacji rozwiązania nawiązuje komunikację z komponentem wspólnej infrastruktury (*ang. Common Infrastructure*). Komponent ten, podobnie jak biblioteka standardowa, współdzielony jest przez wiele środowisk wykonawczych. W kontekście wspólnej infrastruktury, wspomnieć należy o wspólnej specyfikacji języka (*ang. CLS - Common Language Specification*), wspólnym systemie typów (*ang. CTS - Common Type System*), a także środowisku uruchomieniowym wspólnego języka (*ang. CLR - Common Language Runtime*). Wykorzystanie między innymi tych trzech elementów, pozwala na budowę systemu dostępnego na wielu platformach [36].

W kontekście realizowanej pracy, technologia .NET Core użyta została jako środowisko uruchomieniowe dla interfejsów programowania aplikacji tworzonych w języku C#. W obrębie technologii tej, poza przedstawionymi powyżej komponentami, wyróżnić możemy natywną bi-

bliotekę ASP.NET Core, stanowiącą zbiór metod przydatnych w procesie definiowania internetowych usług sieciowych oraz aplikacji webowych. Dzięki zastosowaniu ASP.NET Core operacje takie jak, między innymi: obsługa definicji kontrolerów API, zarządzanie stanem ciała żądania oraz jego rzutowaniem na określony typ danych, czy też implementacja mechanizmów uwierzytelniania i autoryzacji klienta, wykonane mogą zostać na wysokim poziomie abstrakcji z jednoczesnym zapewnieniem należytego poziomu ich wydajności.

Entity Framework Core

Entity Framework Core stanowi narzędzie stworzone przez firmę Microsoft, którego zastosowaniem jest mapowanie obiektowo-relacyjne realizowane w kontekście usług internetowych tworzących z wykorzystaniem języka C# oraz uruchamianych na platformie .NET Core. Przedstawiona biblioteka zapewnia programiście zorientowany obiektowo interfejs, za pomocą którego może on uzyskać dostęp do danych, a także je definiować oraz przetwarzać. Zbiory obiektów mogą być składowane zarówno w relacyjnych jak i niereleacyjnych bazach danych. Niniejsza biblioteka, podobnie do środowiska uruchomieniowego .NET Core, jest rozwiązaniem wieloplatformowym i może być wykorzystywana przy budowie systemów internetowych wdrażanych na systemach Windows, Linux oraz MacOS.

Zastosowanie biblioteki mapera obiektowo-relacyjnego jaką jest Entity Framework Core umożliwia zastosowanie podejścia zorientowanego na kod źródłowy w kontekście aplikacji komunikujących się i wykorzystujących zewnętrzne zbiory danych (*ang. Code-First Approach*). Podejście to, polega na definiowaniu w ramach kodu źródłowego zbioru klas modelu danych, które będą następnie przekształcane do postaci tabel określonego systemu bazodanowego. Przedstawiona operacja przekształcenia wykonywana jest bezpośrednio za pomocą mechanizmów mapera obiektowo-relacyjnego.

Niezaprzecjalną zaletę wykorzystania biblioteki ORM jaką jest Entity Framework Core stanowi możliwość operowania na jednolitym interfejsie realizacji operacji na danych, niezależnie od obsługiwanej systemu bazodanowego. Oznacza to, że w momencie zmiany dostawcy zewnętrznego źródła danych, zawartość kodu źródłowego programu nie musi podlegać modyfikacji [32].

MediatR

MediatR to otwartoźródłowa biblioteka języka C#, z wykorzystaniem której zaimplementowany może zostać wzorzec projektowy, dotyczący separacji odpowiedzialności za obsługę zapytań oraz komend przetwarzanych przez usługę sieciową. Kluczowym elementem biblioteki MediatR jest para generycznych interfejsów, za pomocą których implementowana jest obsługa zarówno żądań jak i zapytań dotyczących danych. Interfejsami tymi są kolejno: *IRequest* - struktura programistyczna implementowana przez klasy definiujące zawartość ciała żądania lub komendy, a także powiązany z nią *IRequestHandler*, który jest konkretyzowany przez klasę definicji metody obsługi żądania bądź operacji na danych.

Ponadto, należy również podkreślić znaczenie metody *Send* dostępnej w ramach głównego API pakietu MediatR. Dzięki niej, wywołana może zostać procedura obsługi określonego żądania lub operacji, z dowolnego miejsca kodu źródłowego interfejsu programowania aplikacji [4].

JavaScript

JavaScript to wielofunkcyjny oraz wieloplatformowy skryptowy język programowania cechujący się wysokim poziomem abstrakcji. Najbardziej popularnym przeznaczeniem omawianego

języka jest budowa systemów internetowych, a także mobilnych. Historyczną rolą technologii JavaScript było udostępnianie programiście funkcjonalności umożliwiających określanie różnorodnych sposobów interakcji pomiędzy użytkownikiem serwisu internetowego, a jego statycznymi elementami. Podstawowym środowiskiem wykonania oraz interpretacji omawianego języka była uwcześnie przeglądarka internetowa. Wraz z pojawieniem się serwerowego środowiska uruchomieniowego NodeJS, przeznaczonego dla języka JavaScript, popularność omawianej technologii wzrosła w gwałtownym tempie. Zmianie uległo również główne przeznaczenie technologii, która od tej pory stała się pełnoprawnym językiem programowania, stosowanym w kontekście budowy zarówno systemów internetowych, rozwiązań mobilnych, jak i programów desktopowych.

Język JavaScript uznać należy za technologię charakteryzującą się typowaniem słabym oraz dynamicznym. W związku z zastosowaniem przez twórców rozwiązania takiego właśnie podejścia, tworzone kody programów narażone są na występowanie zjawisk niezgodności typów, a także niejawnej koercji. Ponadto, w kontekście mechanizmów omawianego języka, realizacja operacji przetwarzania współbieżnego oraz wykonania metod asynchronicznych, zależna jest w całkowitym stopniu od rozwiązań implementacyjnych poczynionych w ramach środowiska uruchomieniowego. Oznacza to, że przetwarzanie i wykonywanie operacji wielowątkowych może cechować się zróżnicowaną wydajnością, w zależności od konkretnego interpretera języka.

Niewątpliwymi zaletami technologii JavaScript są: składnia cechująca się niskim poziomem złożoności poleceń, możliwość dowolnego wykorzystywania wielu spośród wspieranych paradigmatów programowania, modułowość i skalowalność implementowanych rozwiązań, a także elastyczność w kontekście operowania na wykorzystywanych strukturach danych [17].

W ramach niniejszej pracy, język JavaScript zastosowany został w celu implementacji jednego z dwóch zbiorów badanych interfejsów programowania aplikacji. Tworzone w omawianym języku API, wykonywane będą w środowisku uruchomieniowym NodeJS.

TypeScript

TypeScript stanowi statycznie typowany nadzbiór języka JavaScript. Określenie to, oznacza że omawiana technologia nie jest stricte językiem programowania, a tylko określona grupą instrukcji oraz procedur, które włączyć można do języka JavaScript, po to, aby zapewnić w jego kontekście statyczny sposób typowania danych. Technologia TypeScript nie może być wykorzystywana samodzielnie, a środowisko wykonawcze JavaScript jest wymagane w celu uruchomienia skompilowanego modułu, definiowanego zgodnie ze składnią omawianego języka.

Kluczowym elementem przedstawianej technologii jest transpilator języka TypeScript o nazwie tsc (*ang. TypeScript Compiler*). Program ten, uruchamiany jest tuż przed rozpoczęciem procedury intepretacji kodu JavaScript i przekształca on metody odpowiedzialne za obsługę typów danych, do struktur dostępnych w ramach standardowej implementacji języka. Dlatego też, z punktu widzenia środowiska uruchomieniowego, dostępne programistom mechanizmy definicji typów czy interfejsów, nie są znane.

Celem zastosowania omawianego nadzbioru językowego jest możliwość kontroli zgodności definiowanych obiektów programistycznych pod kątem ich wewnętrznej struktury. Ponadto, wykorzystanie TypeScript umożliwia weryfikację faktu nieumyślnego odwołania się do struktury typu nieokreślonego, jeszcze przed rozpoczęciem procesu interpretacji kodu [3].

NodeJS

NodeJS jest środowiskiem uruchomieniowym języka JavaScript zbudowanym w oparciu o otwartoźródłowy silnik interpretacji kodu Chrome V8. Dzięki zastosowaniu omawianego środowiska uruchomieniowego, kod źródłowy języka JavaScript może być wykonywany poza ekosys-

temem przeglądarki internetowej. Rozwój niniejszej technologii, doprowadził do diametralnej zmiany w obszarze zastosowania języka JavaScript, a także gwałtownego wzrostu jego popularności w kontekście budowy systemów internetowych.

Podobnie do rozwiązania Microsoft .NET Core, platforma NodeJS składa się nie tylko ze środowiska uruchomieniowego, ale także ze zbioru bibliotek oraz narzędzi linii komend. Aplikacje budowane na bazie omawianej technologii cechują się zastosowaniem architektury sterowanej zdarzeniami (*ang. Event-driven architecture*), która ponadto wzbogacana jest (dzięki wykorzystaniu mechanizmu pętli) o możliwość obsługi operacji asynchronicznych. Co więcej, rozwiązania definiowane na podstawie środowiska NodeJS posiadają budowę modułową, co przyczynia się do zwiększenia ich zdolności w kontekście skalowania systemów.

Należy również uwypuklić generyczność charakterystyki środowiska NodeJS. Nie jest ono przeznaczone stricte do definiowania i wdrażania usług sieciowych, a wykorzystywane jest do uruchamiania dowolnego kodu języka JavaScript, jaki może zostać stworzony za pomocą jego składni. Dlatego też, aby dostarczać mechanizmy dotyczące specyficznych funkcjonalności, ekosystem NodeJS może być rozbudowywany poprzez otwartoźródłowe moduły. Liczność modułów tych, a także popularność ich wykorzystania stanowi niewątpliwie o sile omawianej technologii [6].

ExpressJS

ExpressJS stanowi bibliotekę środowiska NodeJS, dostarczającą zbiór metod pozwalających na budowę webowych interfejsów programowania aplikacji w tym właśnie środowisku. Pakiet ExpressJS cechuje się minimalizmem w kontekście złożoności udostępnianych programistycie operacji, elastycznością dotyczącą współpracy z zewnętrznymi pakietami, a także wysoką wydajnością działania tworzonych aplikacji, poprzez eliminację złożonych funkcjonalności przetwarzania zasobów w obrębie żądania.

Koncepcja przetwarzania zasobu uzyskanego od klienta w ramach ExpressJS sprowadza się do potokowej obsługi dostarczonego wejścia, przez kolejne funkcje pośredniczące (*ang. Middleware functions*). Ostatnia z funkcji ma za zadanie zwrócić odpowiedź wygenerowaną na podstawie operacji wykonywanych przez wszystkie poprzednie metody. Całkiem funkcji pośredniczącej może być zarówno kod źródłowy zdefiniowany przez programistę, jak i ten dostarczony poprzez referencję do zewnętrznego pakietu. Do każdej z metod przekazywane są parametry żądania, odpowiedzi, a także referencji do następnego middleware. Dzięki uruchomieniu napisanej w taki sposób aplikacji w środowisku NodeJS, poszczególne funkcje pośredniczące mogą być realizowane asynchronicznie [21].

Prisma

Prisma ORM to narzędzie pełniące rolę mapera obiektowo-relacyjnego wykorzystywanego w kontekście implementowania interfejsów programowania aplikacji w języku JavaScript. Dystynktynną cechą omawianego narzędzia jest prostota definiowania rzutowanego modelu danych. W przypadku pakietu Prisma, cała struktura modelu danych opisywana jest w ramach jednego pliku zwanego plikiem schematu. W pliku tym, określone zostają zarówno właściwości poszczególnych encji modelu, jak i relacje które między nimi występują.

Analogicznie do narzędzia Entity Framework Core, Prisma ORM wspiera zarówno relacyjne jak i nierelacyjne systemy bazodanowe. Ponadto, zauważać należy kompatybilność omawianego narzędzia z omawianą wyżej technologią TypeScript [27].

Mongoose

Biblioteka Mongoose stanowi narzędzie rzutowania obiektowego modelu danych definiowanego w ramach kodu programu, do postaci obiektowej nierelacyjnej bazy danych MongoDB. Technologii tej nie należy nazywać maperem obiektowo-relacyjnym, gdyż wykonywane przez nią operacje synchronizują struktury danych, które zarówno po stronie kodu, jak i po stronie źródła danych cechują się obiektową naturą.

Połączenie interfejsu programowania aplikacji z nierelacyjnym źródłem danych obsługiwany poprzez bibliotekę Mongoose, prowadzić może do zwiększenia efektywności oraz zmniejszenia czasu wykonania operacji na danych. Jednakże, w związku z naturą przechowywanych informacji, a także brakiem uwzględniania w ich strukturze metadefinicji, zastosowanie omawianego mechanizmu może również prowadzić do braku spójności źródła danych, a także niemożliwości zastosowania usprawnień wydajnościowych w kontekście bazy danych [14].

Apache JMeter

Narzędzie Apache JMeter to otwartoźródłowe oprogramowanie stworzone w języku Java. Program ten, wykorzystywany jest przeprowadzania ewaluacji wydajności oprogramowania sieciowego opierającego swoje działanie o protokoły HTTP oraz FTP. Testy efektywności działania usług mogą zostać przeprowadzane w trybie lokalnym (tj. z wykorzystaniem jednego hosta wysyłającego żądania do określonej usługi sieciowej), jak i w trybie rozproszonym (tj. budowana zostaje hierarchia hostów będących generatorami żądań). W ramach niniejszej pracy, zastosowany został drugi z przedstawionych trybów ewaluacji.

Działanie oprogramowania Apache JMeter sprowadza się do wykonywania pętli testowej, w ramach określonych grup wątków. Pętla testowa symuluje sekwencyjne generowanie żądań w kierunku serwera, z uwzględnieniem stałej wartości opóźnienia pomiędzy wysyłanymi pakietami. Grupa wątków natomiast, określa współbieżny charakter testów obciążenia usługi sieciowej i może być utożsamiana zarówno z konkretnymi wątkami procesora lokalnego hosta, jak i z oddzielnie pracującymi generatorami żądań w trybie rozproszonym.

W celu zdefiniowania testu wydajności w ramach Apache JMeter, zbudowany powinien zostać plan testowy. Jest to podstawowa jednostka wyróżniana w ramach niniejszego oprogramowania i skupia ona w sobie między innymi komponenty grup wątków, próbników (*ang. Samplers*), a także elementów nasłuchujących na odpowiedź usługi (*ang. Listeners*). Zarówno próbniki, jak i elementy nasłuchujące mogą być powielane w ramach planu testowego, a także indywidualnie konfigurowane, w zależności od specyfiki usługi sieciowej.

Oprogramowanie Apache JMeter obsługiwać można zarówno z poziomu narzędzia linii komend, jak i udostępnionego graficznego interfejsu użytkownika [11].

2.5. Przegląd literatury

W niniejszym rozdziale przedstawione zostaną pozycje literaturowe, do których odnosić się będzie opisywana praca dyplomowa. Pozycje te, podzielone zostały na oddzielne grupy, związane z określoną tematyką.

Na początku, przedstawiona zostanie literatura powiązana z aspektem budowy interfejsów programowania aplikacji oraz będąca wprowadzeniem do wykorzystywanych technologii. Następnie, opisane zostaną pozycje traktujące o wydajności interfejsów API, a także o analizie działania powszechnie dostępnych serwisów internetowych opartych o metodologię REST. Kolejne prace, skupiać się będą na tematyce testowania usług sieciowych, teorii testowania, a także konfiguracji narzędzi dla testów rozproszonych. W następnej kolejności, wspomniane zostaną

prace naukowe oraz dokumenty standaryzacyjne dotyczące sposobu działania protokołu przesyłania danych hipertekstowych. Ostatnią grupą pozycji literaturowych będą prace referencyjne dotyczące badań wydajności systemów internetowych.

Pozycja [36] stanowi wprowadzenie do zaawansowanych koncepcji języka C#, a także dostarcza informacji związanych z wykorzystaniem tego języka w środowiskach uruchomieniowych .NET oraz .NET Core. W początkowych rozdziałach przedstawiono sposób budowy, komplikacji oraz wykonywania programu w środowisku .NET. Kolejno opisana została struktura bazowych aplikacji uruchamianych w tym właśnie środowisku i tworzonych za pomocą języka C#. Ostatnim elementem wprowadzenia do opisywanej technologii było przedstawienie struktur języka w kontekście obiektowego paradygmatu programowania. W następnych sekcjach literatury, w sposób wyczerpujący poruszono tematykę bardziej zaawansowanych aspektów programowania w języku C#, którymi są między innymi: kolekcje i typy generyczne, delegaty i wyrażenia lambda, czy też cykl życia obiektu w pamięci programu. Ważnym tematem, poruszonym w ramach tej książki jest struktura oraz zasada działania środowiska .net core, będącego podstawowym elementem interfejsów programowania aplikacji tworzonych w języku C#.

Analogiczną do przedstawionej powyżej pozycji literaturowej, dotyczącą jednak technologii NodeJS oraz języka JavaScript jest [6]. W ramach tej pracy zawarto obszerne wprowadzenie do platformy NodeJS uwzględniające ponadto kwestie obsługi operacji wejścia/wyjścia, wykonywania natywnego kodu JS, czy też przetwarzania operacji przez silnik NodeJS oraz bibliotekę libuv. Znaczna część pracy, obejmuje przedstawienie zaawansowanych wzorców projektowych, których głównym przeznaczeniem jest obsługa zdarzeń oraz operacji asynchronicznych. Wspomniane zostały także rozwiązania dotyczące skalowalności aplikacji z wykorzystaniem mechanizmów kolejkowania wiadomości.

Niezależnie od wykorzystywanej technologii, interfejsy programowania aplikacji, które zostały zbudowane na potrzeby tej pracy dyplomowej, oparte są o styl architektoniczny RESTful. Styl ten, jest pewnym zbiorem zasad projektowania usług sieciowych, określającym zarówno aspekty sposobu komunikacji klienta z usługą sieciową, jak i techniczne wymagania dotyczące przetwarzania żądań. Dobre praktyki, które uwzględnia metodologia REST, zawarte zostały w pozycji literaturowej [34]. Autorzy tego dokumentu, na wstępie dokonują porównania architektury zorientowanej na zasoby, będącej podstawą konwencji REST, z popularną uprzednio architekturą zorientowaną na usługi. Następnie, przedstawiane są najlepsze praktyki, cele oraz reguły REST dotyczące projektowania interfejsu programowania aplikacji. Co więcej, w omawianej książce zawarte zostały także podstawowe oraz zaawansowane wzorce projektowania API, uwzględniające aspekty bezstanowości, paginacji, osiągalności, a także identyfikacji zasobów interfejsu. Końcowe rozdziały książki, wprowadzają w kwestie testowania oraz bezpieczeństwa REST API, omawiają technikę kompozycji usług RESTful, a także przedstawiają rozwiązania (biblioteki oraz języki programowania) pozwalające na tworzenie interfejsów API zgodnych z metodologią REST.

Podstawowym celem działania interfejsu programowania aplikacji jest dostarczenie danych do konsumenta, bądź też ich manipulacja zgodnie z jego żądaniem. Aby operować na danych, interfejs API musi komunikować się ze źródłem danych, którym najczęściej jest serwer bazodanowy. W celu dostarczenia metod komunikacji pomiędzy API a źródłem danych, które jednocześnie są niezależne od wykorzystanego źródła, a także pozwalają na zarządzanie danymi z poziomu struktur języka, stworzone zostały biblioteki zwane maperami obiektowo-relacyjnymi (ang. Object-Relational Mappers). Dla API napisanego w języku C# podstawowym rozwiązaniem ORM jest biblioteka Entity Framework Core, która przedstawiona została w pozycji [32]. Pozycja ta, uwzględnia zarówno opis działania najczęściej wykorzystywanych metod służących do manipulacji danymi, jak i rolę klasy kontekstu bazodanowego w procesie tłumaczenia operacji programistycznych na polecenia bazodanowe. Ponadto, dowiedzieć możemy się jak przetwar-

rząc zaawansowane typy danych (takie jak np. DateTime), czy też w jaki sposób wykorzystywać zapytania LINQ do budowania kwerend.

Dla interfejsu programowania aplikacji napisanego w języku JavaScript i uruchamianego w środowisku NodeJS, w przeciwnieństwie do platformy .NET, zastosować możemy zdecydowanie większą liczbę bibliotek pełniących rolę maperów obiektowo-relacyjnych. Biblioteki te, zostały opisane w pozycjach [5] i [20]. Pozycja [5] pełni rolę całkowitego wprowadzenia do tematyki tworzenia interfejsów API, korzystając z platformy NodeJS, frameworka ExpressJS oraz nierelacyjnej bazy danych MongoDB. Rodził piąty tej pracy, traktujący o wykorzystaniu baz danych NoSQL, przybliża tematykę jednego z najczęściej wykorzystywanych maperów obiektowo-relacyjnych dla Node czyli mongoose. Przedstawiono tutaj sposób zestawienia połączenia z serwerem bazodanowym, tworzenia encji modelu, przekształcanego następnie na struktury bazy danych, a także wykonywania operacji dostępu do danych i ich modyfikacji. W pracy [20] natomiast, porównano nierelacyjne podejście do składowania danych typu geograficznego z podejściem relacyjnym, wykorzystując w tym przypadku biblioteki mongoose i sequelize. Oba mapery obiektowo relacyjne zostały użyte w ramach interfejsu API wykorzystującego technologie NodeJS/ExpressJS. Celem opisywanej pracy było przedstawienie różnic w czasach odpowiedzi API na uzyskane żądanie, dla różnej liczby danych geolokalizacyjnych, uwzględniając zastosowanie relacyjnych i nierelacyjnych baz danych.

Następne pozycje literaturowe, związane są z analizą usług REST oraz wydajnością webowych interfejsów programowania aplikacji.

Pozycja [24] stanowi analizę 500 serwisów internetowych z listy alexa.com4000 najpopularniejszych dostępnych publicznie usług sieciowych. Twórcy każdego z 500 serwisów deklarują zgodność swoich produktów z konwencją REST. Przeprowadzona analiza dotyczyła kluczowych aspektów technicznych związanych z funkcjonowaniem API, stopnia zgodności API z regułami dotyczącymi metodologii REST, a także przestrzegania najlepszych praktyk projektowania interfejsów programowania aplikacji, takich jak m.in. zastosowanie mechanizmu wersjonowania. W trakcie analizy, zaobserwowano określone trendy dla aplikacji REST API, takie jak m.in. rozpowszechnione wsparcie notacji JSON, czy wykorzystywanie narzędzi do dokumentacji generowanej programowo. Ponadto, zauważono, że tylko ok. 0.8% analizowanych serwisów webowych przestrzega w sposób ścisły reguł zawartych w ramach konwencji REST.

Wydajność interfejsów programowania aplikacji, jako jeden z elementów miary jakości API została przedstawiona w pozycji [2]. Na początku pracy, jej autorzy wskazują na interakcję interfejsu programowania aplikacji z systemami klienckimi. Opisany został tutaj zestaw protokołów sieciowych wykorzystywanych podczas formułowania i transmisji żądania, system zunifikowanych lokacji zasobów, a także semantyka interakcji w zależności od wykorzystywanych typów żądań protokołu hipertekstowego. Ponadto, wskazano najczęstsze przyczyny błędów przepływu danych dla http, uwzględniając działanie usługi DNS, błędy połączenia, błędy leżące po stronie klienta, a także błędy wynikające z działania serwera. Kolejna część pracy, związana jest ze składowymi metryki jakości, do których według autorów, poza wydajnością, zaliczyć możemy: dostępność, procent żądań dla których uzyskano pozytywną odpowiedź, osiągalność, a także możliwość sprawdzenia stanu usługi w dowolnym momencie jej działania. Dodatkowo, w niniejszej pracy zaproponowano podejście oraz zestaw narzędzi pozwalających na dokonanie ewaluacji jakości interfejsu programowania aplikacji, zgodnie z przyjętą normą jakości.

Kolejnym etapem następującym po zdefiniowaniu metryki wydajności, jest ustalenie wartości tejże metryki w kontekście testowanych usług sieciowych. Przytoczone poniżej pozycje literaturowe, związane są z wykonywaniem pomiarów wydajności API, czyli testowaniem.

Pozycja [33] stanowi obszerne wprowadzenie do teorii testowania oprogramowania. W pierwszych rozdziałach tego dokumentu, wyjaśniono czym jest testowanie, dlaczego jest ono niezbędne podczas tworzenia oprogramowania, a także jak wygląda podstawowy proces wykonywania testów. Następnie przedstawiono proces testowania w kontekście tworzenia oprogra-

mowania. Uwzględniono tu zarówno modele cyklu życia rozwoju systemów w powiązaniu z testowaniem, poziomy realizowanych testów, ich typy, jak i sposoby zarządzania testami. Kolejne rozdziały tyczą się testowania statycznego (tj. testowania funkcjonalności lub modułu na poziomie jego specyfikacji lub implementacji bez wykonywania kodu testowanego oprogramowania), dostarczają teorii związanej z poszczególnymi technikami testowania rozwiązań oraz przedstawiają aspekt organizacji, planowania, monitorowania oraz uwzględniania ryzyka w czasie dokonywania ewaluacji systemów. W ostatnim z rozdziałów dokumentu, autorzy przedstawiają narzędzia przydatne w procesie testowania, a także sposób ich efektywnego wykorzystania w codziennej pracy.

Pozycja [29] zawiera wiele analogicznych treści do pracy opisanej powyżej, jednakże rozwija ona w sposób wyczerpujący, wspomniane tylko w poprzedniej pracy aspekty. W części drugiej dokumentu zawarto dogłębną analizę zagadnienia testowania statycznego, uwzględniając m.in. testowanie zgodności ze standardami oprogramowania, symboliczne wykonywanie kodu, a nawet wprowadzając aparat matematyczny do formalnego dowodzenia poprawności fragmentów oprogramowania. W ramach tej książki, przedstawiono także dynamiczną analizę systemu (tj. testowanie funkcjonalności lub modułu na poziomie wykonywanego kodu) uwzględniając często występujące błędy związane m.in. z nieumiejętnym zarządzaniem strukturami pamięci programu. Ponadto, uwzględniono zagadnienie priorytetyzacji przypadków testowych, wprowadzając pojęcie miary średniego procenta wykrytych usterek. Autor dokumentu przedstawia także testowanie charakterystyk jakościowych zgodnie z normą ISO 9126 oraz ISO 25010, tworzenie dokumentacji w ramach zarządzania testowaniem, czy chociażby zarządzanie incydentami występującymi w ramach procesu ewaluacji oprogramowania.

W ramach pozycji [23], dowiedzieć możemy się ponadto o testowaniu usług internetowych. Przedstawiono tutaj podstawową strukturę standardowej usługi sieciowej (w tym przypadku – usługi e-commerce) cechującej się architekturą trójwarstwową. Ponadto, wyjaśniono rolę każdej z warstw systemu, a także przedstawiono aspekty testowania oprogramowania w kontekście każdej z nich. Dodatkowo, zawarte zostały przykładowe przypadki testowe, dotyczące zarówno prezentacji danych w systemie, jak i dostępu do danych poprzez serwer webowy. Dla zaprezentowanych przypadków testowych, przedstawione zostały także scenariusze realizacji testów w postaci listy czynności jakie należy podjąć, aby dokonać ewaluacji systemu.

Aspekty technologii testowania oprogramowania ujęte zostały także w pozycji [22]. Artykuł ten, stanowi sekcję wprowadzającą do książki pt. Tutorial: Software Testing and Validation Techniques, tego samego autora. Pozycja ta, przedstawia przekrój technik oraz technologii testowania oprogramowania wykorzystywanych na przestrzeni ostatnich ok. 30 lat. Opisane zostały tutaj zarówno teoretyczne podstawy testowania, narzędzia i techniki analizy statycznej i dynamicznej, oceny efektywności przeprowadzanych testów, a także badania przeprowadzane w dziedzinie testowania i walidacji oprogramowania. Omawiany artykuł, wyszczególnia pozytywne oraz negatywne aspekty poszczególnych technik oraz wskazuje przydatność określonych rozwiązań do testowania oprogramowania różnego typu.

Ostatnią przytoczoną w ramach tego przeglądu literaturowego pozycją, dotyczącą teorii ewaluacji oprogramowania jest [16]. Pozycja ta, stanowi normę międzynarodowej organizacji normalizacyjnej (ang. International Organization for Standardization) dotyczącą weryfikacji jakości oprogramowania. Uwzględniono tu przede wszystkim znaczenie pojęć stosowanych w dziedzinie testowania oprogramowania, wprowadzono definicje dla określonych terminów oraz zjawisk występujących w ramach ewaluacji systemów, a także określono zgodność wprowadzanych przez standard konceptów, z konceptami zawartymi w standardach pochodnych. Główną część dokumentu, stanowi wprowadzenie szkieletu modelu jakości, uwzględniającego określone modele jakościowe, modele jakości w użyciu, a także modele jakości produktu. Dodatkowo, przedstawiono cel oraz sposób wykorzystania modeli jakościowych, wyjaśniono różnicę w postrzeganiu modeli jakościowych z punktu widzenia różnych interesariuszy, a także zdefiniowano

relacje pomiędzy określonymi modelami. Dokument ten, wraz z normą ISO 9126, stanowią definicję pojęcia jakości w kontekście testowania oprogramowania.

Pozycja [26] stanowi przegląd narzędzi wykorzystywanych do testowania działania systemów komputerowych. Na początku książki, wprowadzany jest termin zapewnienia jakości (ang. Quality Assurance), który w dzisiejszych czasach definiuje zakres odpowiedzialności osoby testującej oprogramowanie. Kolejno, przedstawiane są kryteria sukcesu dotyczące tworzonego systemu, a także fazy poszczególnych modeli rozwoju oprogramowania zorientowanych na procesy. Analogicznie do pozycji literaturowych przedstawionych uprzednio, w ramach tej pozycji określone zostały metryki i definicje jakości oprogramowania oraz omówiony został proces realizacji testów. Główna część omawianego dokumentu skupiona jest wokół narzędzi stosowanych do realizacji ewaluacji oprogramowania. Wyszczególniono tutaj narzędzie WinRunner, przedstawiając między innymi wykorzystywany w tym programie skryptowy język testów (ang. Test Script Language). Ponadto, przedstawiono architekturę oraz najważniejsze funkcjonalności narzędzi SilkTest, SQA Robot, LoadRunner, TestDirector, QuickTest Professional a także Apache JMeter. Ostatni z wymienionych programów, wykorzystywany zostanie w ramach niniejszej pracy dyplomowej, dlatego też dalszy przegląd tej pozycji literaturowej skupiony będzie na rozdziale dotyczącym właśnie tego narzędzia. Opis funkcjonalności aplikacji JMeter został w niniejszej pozycji podzielony na sekcje związane z testowaniem rozwiązań bazodanowych wykorzystujących interfejs JDBC (ang. Java DataBase Connectivity), a także sekcję dotyczącą testowania aplikacji bazujących w swoim działaniu na protokole hipertekstowym. Przedstawiono tutaj sposób tworzenia grup wątków reprezentujących użytkowników aplikacji, generowania żądania protokołu hipertekstowego, uruchomienia mechanizmu nasłuchiwanego na odpowiedź serwisu, dodawania licznika czasu, a także zapisywania i przeglądania rezultatów przeprowadzonego testu.

W ramach dokumentów [25] oraz [11] przedstawiono pełen zakres funkcjonalności dostępnych w ramach narzędzia Apache JMeter. Pierwsza z prac (tj. [25]), skupia się na wykorzystaniu narzędzia w celu wykonywania testów wydajności usług sieciowych, natomiast druga z pozycji (tj. [11]), przedstawia aplikację JMeter dla różnych kontekstów jej potencjalnego użycia. W obu pracach wyszczególnione zostają podstawowe elementy, na które składa się środowisko testowe. Elementami tymi są: grupy wątków, komponenty próbujące, kontrolery, komponenty nasłuchujące, liczniki czasu oraz asercje. Ponadto, omówiono elementy graficznego interfejsu użytkownika dla aplikacji, przedstawiono proces instalacji oraz uruchamiania narzędzia JMeter, a także zdefiniowano pojęcie planu testów. W kontekście pracy [25], poza wymienionymi uprzednio kwestiami, zobrazowany został także proces wykonywania testu przeciążeniowego dla usługi zorientowanej na serwisy (ang. Service-Oriented Application). Proces ten uwzględniał: tworzenie grupy wątków, konfigurację struktury żądania wysyłanego do usługi, uruchomienie testu, a także pozyskanie wyniku. W pracy [11] natomiast, analogiczny proces, możemy zaobserwować dla monolitycznej aplikacji internetowej oraz interfejsu programowania aplikacji. Ponadto, przedstawione zostały zaawansowane opcje konfiguracji elementów nasłuchujących oraz liczników czasu, a także pokazany został proces wykorzystania pośredniczącego serwera http, w celu dokumentowania realizowanych żądań.

Następne pozycje literaturowe omówione w ramach tej pracy, dotyczą budowy oraz zasady działania internetowego protokołu hipertekstowego (ang. Hypertext Transfer Protocol), a także implementacji mechanizmu zarządzania stanem. Mechanizm ten, w związku z naturą protokołu http, nie jest w nim domyślnie realizowany.

Pozycja [8] stanowi techniczny dokument dotyczący semantyki oraz budowy internetowego protokołu hipertekstowego w wersji 1.1. Zdefiniowano w nim pojęcie zasobu żądania oraz omówiono cykl życia jego przetwarzania. Wskazano także moment, w którym zasób rekonstruowany jest przez serwer na podstawie jego efektywnego identyfikatora URI (ang. Uniform Resource Identifier). Ponadto, nakreślono pojęcie reprezentacji danych przesyłanych za pomocą proto-

kołu http, definiując określone pola nagłówkowe dotyczące: typu danych, sposobu kodowania, języka danych, a także lokalizacji zasobu. Kolejne rozdziały dokumentu zawierają informacje dotyczące definicji dozwolonych metod protokołu http oraz znaczenia jakie te metody wprowadzają w kontekście operacji na zasobie. W dokumencie przedstawiono także kody statusu odpowiedzi na żądnie, grupując je w sposób semantyczny. Dla każdego z przedstawionych kodów statusu nakreślono kontekst, w jakim odpowiedź, oznaczona tym właśnie kodem, powinna być zwracana klientowi. Na końcu pracy, omówiono kwestie związane z bezpieczeństwem protokołu takie jak: ataki bazujące na wstrzykiwaniu kodu czy ochrona przed ujawnianiem informacji wrażliwych w identyfikatorach zasobów.

Pozycja [10] pozwala na poszerzenie wiedzy dotyczącej protokołu hipertekstowego w bardziej praktycznym kontekście. Podobnie jak w dokumencie [8], przedstawiono tutaj informacje teoretyczne dotyczące architektury protokołu, definicji zasobów czy też ujednoliconego formatu ich adresowania. Ponadto, wskazano i scharakteryzowano określone typy połączeń realizowanych z wykorzystaniem protokołu hipertekstowego. Co więcej, dla każdego z nich rozważono kwestie związane z wydajnością połączenia pomiędzy klientem a serwerem. Kolejne rozdziały pracy [10] traktują o identyfikacji klienta w ramach serwera, jego uwierzytelniania przed serwerem, a także szyfrowania danych przesyłanych pomiędzy tymi dwiema jednostkami. W niniejszej pracy wspomniano także o internacjonalizacji żądań w kontekście zastosowania nagłówka ‘Accept-Language’. Ostatnie rozdziały dokumentu dotyczą kwestii publikowania i dystrybucji zawartości. Wyszczególnione zostały tu takie elementy jak: web hosting, systemy publikacji treści, czy też mechanizm przekierowań oraz równoważenia obciążień.

Zgodnie z charakterystyką protokołu http, realizuje on komunikację w sposób bezstanowy. Oznacza to, że domyślnie, pomiędzy klientem a serwerem nie jest utrzymywana sesja połączeniowa, a każde żądanie generowane przez klienta w kierunku serwera rozpatrywane jest indywidualnie. Rozwiążanie takie, pozwala na znaczące przyspieszenie działania protokołu hipertekstowego, a także uproszczenie jego konstrukcji. Jednakże, szczególnie w przypadku aplikacji internetowych komunikujących się z serwerem http, bezstanowy charakter protokołu bywa problematyczny w aspekcie kontekstu wysyłanych sekwencyjnie żądań. Dlatego też, do protokołu http wprowadzono mechanizm zarządzania stanem opisany w dokumencie [1]. Dokument ten, definiuje pola nagłówkowe o nazwach ‘HTTP Cookie’ oraz ‘Set-Cookie’. Pola te, mogą być używane przez serwery http w celu przechowywania stanu w ramach aplikacji klienckich, dając serwerom tym możliwość zarządzania, zawierającą stan sesją, przy wykorzystaniu protokołu bezstanowego. W niniejszym dokumencie, dla obu przedstawionych pól wyszczególniono atrybuty składowe pola, a także określono znaczenie każdego z nich. Ponadto, dokument definiuje wymagania dla klienta http, dotyczące możliwości wykorzystania mechanizmu zarządzania stanem. Pod uwagę wzięte zostały także kwestie bezpieczeństwa takie jak identyfikatory sesji, słaba poufność danych, czy też zaufanie do usługi nazw domenowych w celu prawidłowego działania mechanizmu zarządzania stanem.

Ostatnia grupa pozycji literackich, zawartych w ramach niniejszego przeglądu literaturo-wego dotyczy badań związanych z testowaniem wydajności aplikacji internetowych w środowisku rozproszonym. Pozycje przedstawione poniżej, będą stanowić prace referencyjne względem niniejszej pracy dyplomowej.

Artykuł [15] dotyczy porównania wydajności działania interfejsów programowania aplikacji tworzonych z wykorzystaniem platform .NET Core 3.1 oraz .NET 5. Celem powstania tego dokumentu była weryfikacja zjawiska wzrostu wydajności działania programów, tworzonych i uruchamianych z wykorzystaniem nowszej z platform firmy Microsoft. Praca ta, ma także na celu pomóc pozwolić odpowiedzieć na pytanie, czy kod źródłowy interfejsu programowania aplikacji o określonych funkcjonalnościach, a także korzystający z określonych narzędzi, powinien zostać zaktualizowany w taki sposób, aby wspierać najnowszą, stabilną wersję środowiska .NET. W ramach dokumentu, w celu realizowania pomiarów wydajności wykorzystano

opisane w poprzednich akapitach narzędzie Apache JMeter, a także dedykowaną środowisku .NET, bibliotekę BenchmarkDotNet. Kolejne rozdziały artykułu przedstawiają przygotowane środowisko testowe, plan wykonywanych testów, a także uzyskane rezultaty wraz z ich analizą. Autor pracy, zobrazował wyniki sześciu testów wydajnościowych, biorących pod uwagę proces serializacji oraz deserializacji obiektów typu JSON za pomocą bibliotek NewtonsoftJson, a także System.Text.Json. Ponadto, przygotowany został test wyszukiwania wzorca z obszernym ciągiem tekstem oraz test wykorzystania punktu końcowego jako klienta zewnętrznego API. Na podstawie otrzymanych rezultatów, wnioskować możemy o około 24 procentowym średnim wzroście wydajności wykonywania operacji realizowanych w ramach testów. Ponadto, wykazano także dość znaczący (około 35 procentowy) średni spadek wydajności nowego rozwiązania względem poprzednika, w kontekście testów obciążeniowych.

Analogiczne badania przeprowadzono w ramach pracy [39]. W tym przypadku jednak, nie skupiały się one na aspekcie porównania technologii, a na sposobie wykonywania pomiarów, a także definiowaniu kryteriów oceny jakości. W pracy tej, interfejs programowania aplikacji zbudowany w oparciu o metodologię REST poddawany był zmiennym obciążeniom (tj. testy linii bazowej, testy obciążeniowe oraz testy przeciążeniowe). W czasie dokonywania ewaluacji monitorowano średni czas odpowiedzi serwera, zgodność kodów statusu zawartych w ramach uzyskiwanych odpowiedzi, informacje o zużyciu zasobów sprzętowych serwera, czy też wartość wskaźnika satysfakcji klienta. Rezultaty przeprowadzonych badań wykazały kluczowe znaczenie optymalizacji kodu źródłowego aplikacji, w kontekście realizacji rozbudowanych i skalowalnych usług sieciowych.

Rozdział 3

Opis problemu badawczego

W ramach niniejszego rozdziału omówiony został podjmowany problem badawczy. W związku z jego złożonością, autor pracy zdecydował się na podział tego zagadnienia na określone aspekty, będące różnorodnymi względem funkcjonalności internetowych interfejsów programowania aplikacji. Na koniec tej części pracy, na podstawie sformułowanych kontekstów badawczych, zdefiniowano listę scenariuszy realizacji badań.

3.1. Przedstawienie aspektów problemu badawczego

Podejmowany w ramach niniejszej pracy problem badawczy, tyczy się wydajności usług sieciowych, których charakterystyka wyróżnia zastosowanie wielu odrębnych komponentów programistycznych. Komponentami tymi, mogą być zarówno: biblioteki obsługujące proces mapowania obiektowo-relacyjnego, zastosowany wzorzec projektowy w kontekście wewnętrznej architektury API, rodzaj wykorzystywanego zewnętrznego źródła danych, czy też wybrana chmurowa platforma wdrożeniowa. W związku z mnogością zagadnień występujących w ramach niniejszego problemu badawczego, zdecydowano się na dokonanie podziału jego opisu w taki sposób, aby na podstawie każdego z aspektów, możliwe było zdefiniowanie wyspecyfikowanych scenariuszy realizacji badań.

Każdy z aspektów problemu badawczego wymieniony poniżej, rozpatrywany będzie względem dwóch odrębnych zestawów technologii programistycznych (tj. C#/.NET oraz JavaScript/NodeJS). Scenariusze badawcze, opracowane na podstawie, każdego z opisanych w tym podrozdziale aspektów, uwzględniać będą porównanie uzyskanych wyników badań dla obu wymienionych rozwiązań informatycznych.

Wydajność interfejsu API względem liczby żądań generowanych przez klientów

Pierwszym z omawianych aspektów rozważanego problemu badawczego jest wpływ wydajności działania interfejsu programowania aplikacji, względem liczby klientów, którzy w sposób równoległy generują żądania w kierunku API.

Wydajność, w kontekście tego właśnie aspektu, interpretowana jest poprzez metryki czasu odpowiedzi na żądanie, a także procentowe wartości wykorzystania zasobów sprzętowych interfejsu programowania aplikacji, takich jak centralna jednostka przetwarzania, czy też pamięć operacyjna o dostępie swobodnym.

Zgodnie z obowiązującymi praktykami realizacji pomiarów wydajności usług sieciowych, charakterystyka ta, powinna być wyliczana w oparciu o technikę testowania rozproszonego (*ang. Distributed Testing*). Przedstawiana technika, zakłada wykorzystanie wielu odrębnych syste-

mów informatycznych, wykonujących równolegle ewaluację obciążeniową. Wartości metryk, uzyskane dla każdej z maszyn przeprowadzających testy, powinny zostać zgrupowane, a także analizowane jako pochodzące z jednego źródła.

Zmiana wydajności, obserwowana powinna być wraz ze stałym zwiększeniem natężenia liczby klientów, a dla każdego z kolejnych przedziałów liczbowych, uwzględniających kolejne przyrosty wysyłanych pakietów, metryki wydajnościowe powinny być porównywane względem ustalonego wskaźnika referencyjnego. Kalkulacja tego wskaźnika natomiast, wykonywana jest w oparciu o ewaluację wydajności dla standardowych warunków pracy interfejsu programowania aplikacji. Przykładem takich warunków, może być realizacja testu obciążeniowego dla pojedynczej maszyny testującej.

Jednym z podstawowych, otwartych standardów, które posłużyć mogą do budowy wskaźnika referencyjnego jest APDEX (*ang. Application Performance Index*). Indeks ten, pozwala na zdefiniowanie trzech przedziałów liczbowych, określających odczucia klienta testującego oprogramowanie. Przedziały te, przedstawiane są jako progi satysfakcji, tolerancji oraz frustracji. Po wykonaniu testów odbywających się w standardowych warunkach pracy usługi sieciowej, ustalenie wartości metryk wydajnościowych dla każdego z trzech progów jest możliwe, a co za tym idzie, możliwe jest także porównanie wyników uzyskiwanych przy dowolnej liczbie klientów API, względem ustalonych progów.

Na podstawie weryfikacji wydajności interfejsu API względem liczby generowanych równolegle żądań, należy także ustalić graniczną wartość sumy maszyn klienckich, dla których interfejs programowania aplikacji jest w stanie obsługiwać zapytania, w czasie zawierającym się w każdym z przedziałów referencyjnych.

Korelacja charakterystyk wydajnościowych względem określonego zewnętrznego źródła danych

Kolejny z kontekstów problemu badawczego dotyczy wpływu zastosowania odmiennych zewnętrznych źródeł danych, na efektywność pracy usługi sieciowej jaką jest interfejs API.

Koncepcja wydajności w przypadku tego aspektu problemu, rozumiana jest w sposób analogiczny do pojęcia, wprowadzonego w ramach poprzedniej sekcji.

W związku z istotnością uwzgladnienia zewnętrznych źródeł danych, jako elementów z którymi nieustannie komunikują się nowoczesne usługi sieciowe, w ramach niniejszego aspektu omawianego problemu badawczego, weryfikowany jest wpływ sposobu obsługi najpopularniejszych spośród dostępnych nieodpłatnie systemów bazodanowych, na efektywność operacji realizowanych przez poszczególne interfejsy programowania aplikacji. Problem badawczy, uwzględnia zastosowanie zarówno czterech relacyjnych systemów baz danych, jak i jednego nierelacyjnego.

Sposób obserwacji zmiany wydajności usługi sieciowej, również jest analogiczny, do tego, który został przedstawiony w ramach poprzedniego aspektu, jednakże należy zwrócić uwagę, na konieczność ponownego zdefiniowania przedziałów satysfakcji, tolerancji oraz frustracji dla wskaźnika referencyjnego. Referencja do wartości tego wskaźnika, obliczonego bez uwzględnienia połączenia z systemem bazodanowym, prowadziła by do zaniżenia wartości ogólnej wydajności testowanego oprogramowania.

Kluczowym czynnikiem omawianego aspektu problemu badawczego jest zapewnienie deterministycznego charakteru stanu łącza sieciowego, występującego pomiędzy serwerem bazodanowym a usługą interfejsu programowania aplikacji. Dlatego też, w przypadku testów realizowanych w środowisku lokalnym, oba komponenty programowo-sprzętowe powinny znajdować się w tej samej lokalizacji. Należy podkreślić jednak, że ważnym jest, aby obie usługi informatyczne nie były uruchomione w ramach tego samego środowiska sprzętowego. Dzięki temu, pozyskane wartości metryk wykorzystania zasobów sprzętowych nie będą obarczone nie-

dokładnością. W kontekście realizacji badań w środowisku chmurowym natomiast, ważnym jest wdrożenie zarówno API, jak i serwera bazy danych, w ramach tego samego centrum obliczeniowego, a także rozdzielenie obu usług, pomiędzy odmienne fizyczne urządzenia. Ponadto, modyfikacji powinien ulec jeden z elementów kryterium wydajności, który definiowany jest jako czas odpowiedzi interfejsu na żądanie klienta. W związku z dyspersją geograficzną obu stron komunikacji, niemożliwym jest zachowanie przewidywalnego charakteru łączącego sieciowego wykorzystywanego do transmisji danych. Twierdzenie to, implikuje konieczność realizacji pomiaru czasu działania usługi sieciowej w sposób odmienny. Kryterium czasu odpowiedzi na żądanie, rozumiane w tym przypadku jest jako przedział czasowy od momentu pozyskania żądania, do momentu zakończenia wszystkich operacji, realizowanych w kontekście tego żądania.

Efektywność realizacji złożonych obliczeń oraz wsparcia dla programowania współbieżnego i metod asynchronicznych

Niniejszy aspekt problemu badawczego dotyczy wpływu wykorzystania, dostępnych w ramach określonego języka mechanizmów programowania współbieżnego, a także sposobu realizacji operacji asynchronicznych, na efektywność przeprowadzania kalkulacji w obrębie warstwy logiki biznesowej interfejsu programowania aplikacji.

Pojęcie efektywności dokonywanych kalkulacji postrzegane jest poprzez liczbę wykonanych iteracji głównej pętli zaimplementowanego algorytmu metaheurystycznego, rozwiązującego określony problem z rodziny NP-trudnych.

W celu zachowania rzetelności badań, omawiany fragment problemu badawczego uwzględnia zastosowanie analogicznego algorytmu, realizującego operacje w ten sam sposób, a także rozwiązującego ten sam problem obliczeniowy. W tym przypadku, zaobserwować będzie można fakt przystosowania technologii poddawanej ewaluacji, do dokonywania procesu zrównoleglania obliczeń, a także przeprowadzania wewnętrznej optymalizacji określonych linii zdefiniowanego kodu źródłowego.

Zaimplementowany algorytm metaheurystyczny, dostępny będzie bezpośrednio z poziomu punktów końcowych badanych interfejsów programowania aplikacji, a liczba iteracji głównej pętli algorytmu, mierzona będzie dla ustalonego, stałego czasu wykonania programu.

Ponadto, omawiany aspekt badawczy dotyczy także weryfikacji wydajności w kontekście zastosowania metod asynchronicznych. W związku ze znacząco odmienną strukturą badanych środowisk uruchomieniowych oraz języków programowania, mechanizmy obsługi operacji asynchronicznych zaimplementowane są w tych technologiach, na różnych poziomach obsługi programu. W jednym przypadku, obsługa operacji tych, jest wykonywana bezpośrednio w ramach języka programowania, natomiast w kontekście drugiej z technologii, metody których wynik nie jest dostarczany natychmiastowo, muszą zostać obsłużone wewnętrz środowiska uruchomieniowego.

Badanie weryfikacji wydajności dla funkcji asynchronicznych, oparte jest o odwołanie się do interfejsu API, do współpracującej z nim hipertekstowej usługi sieciowej, pełniącej rolę pośrednika w dostępie do zdefiniowanych wewnętrz niej informacji. Przedstawiona usługa sieciowa, zostanie zaimplementowana jako odrębne oprogramowanie i będzie niezależna od obu porównywanych technologii.

W kontekście drugiej z części aspektu problemu badawczego, metryką wydajności będzie czas odpowiedzi interfejsu na żądanie.

Wpływ zastosowania wzorca projektowego podziału odpowiedzialności na efektywność realizacji operacji bazodanowych

Rozważany aspekt problemu badawczego tyczy się wpływu implementacji optymalizacji wydajnościowych w kontekście komunikacji interfejsu programowania aplikacji z zewnętrznym źródłem danych.

Wykorzystując konwencjonalną trójwarstwową architekturę interfejsu API, stosowany zostaje ten sam model danych, zarówno do operacji odczytu jak i zapisu. Powoduje to brak możliwości dostosowania modelu, względem specyfiki konkretnego rodzaju operacji. Wprowadzenie wzorca projektowego separacji zapytań oraz komend ma na celu umożliwienie dokonania optymalizacji wydajnościowych wyizolowanych fragmentów modelu danych, a także wykorzystywanie ich tylko i wyłącznie w kontekście jednego typu operacji.

Co więcej, optymalizacja może być wykonana nie tylko na poziomie modelu danych, ale także w ramach fizycznych struktur zawartych wewnętrz obsługiwanej systemu bazodanowego. Dlatego też, przedstawiany aspekt problemu badawczego dotyczy zastosowania zarówno odrębnych modeli danych wewnętrz API, odrębnych struktur programistycznych obsługujących dane modelu, jak i odseparowanych zewnętrznych źródeł danych.

Oba zastosowane źródła danych, powinny cechować się taką samą strukturą, jednakże każde z nich powinno wprowadzać charakterystyczne dla typu wykonywanych operacji, usprawnienia wydajnościowe. Ponadto, aby zachować spójność zawartości dostępnej dla klienta w ramach API, po odwołaniu się do systemu bazodanowego w celu zapisania rekordu, musi on zostać następnie zreplikowany do źródła danych obsługującego operację odczytu.

Wydajność, rozumiana poprzez czas odpowiedzi interfejsu API na żądanie klienta, powinna zostać porównana z tą, wykazywaną przez usługę sieciową opierającą się na architekturze 3-warstwowej i wykorzystującą pojedyncze źródło danych.

Wpływ zastosowania mechanizmów pamięci podręcznej na wydajność interfejsów API

Problem badawczy w kontekście wykorzystania mechanizmów pamięci podręcznej, dotyczy porównania efektywności działania interfejsów programowania aplikacji implementujących standardowy oraz autorski mechanizm przechowywania rezultatów wykonanych uprzednio żądań.

Standardowy mechanizm przechowywania żądań w ramach pamięci podręcznej uwzględniać powinien stały czas ważności pojedynczego wpisu, a także jego unieważnienie w przypadku wykonania operacji modyfikującej dane. W takim przypadku, czas odpowiedzi na żądanie powinien być zwiększy w momencie konieczności odwołania się API do zewnętrznego źródła danych, a następnie zredukowany w przedziale czasowym, w ramach którego wpis pamięci podręcznej jest aktywny.

Zaimplementowany autorski mechanizm pamięci podręcznej wyróżniać będzie się zmiennym czasem ważności poszczególnych wpisów, który zależny będzie od prawdopodobieństwa wywołania określonego punktu końcowego, na podstawie informacji o liczbie historycznych wywołań. Czym większe istnieje prawdopodobieństwo ponownego wywołania punktu końcowego, tym czas ważności rezultatu przechowywanego w pamięci podręcznej będzie większy. Analogicznie do standardowego mechanizmu pamięci podręcznej, operacja modyfikacji danych unieważnia wszystkie spośród wpisów, które odwołują się do przekształconych informacji.

Celem niniejszego aspektu badawczego w ramach rozważanego problemu jest porównanie zmiany wydajności działania API, postrzeganej jako średni czas odpowiedzi na żądanie w ustalonym, stałym przedziale czasowym. Porównywane zostaną mechanizmy standardowy oraz autorski, odrębnie dla każdej z technologii programistycznych.

Wpływ wdrożenia interfejsu API na dedykowanej platformie chmurowej na jego efektywność działania

Ostatni z przedstawianych aspektów problemu badawczego odnosi się do wpływu wydajności pracy interfejsu API, w zależności od rodzaju zastosowanej platformy chmurowej, na jakiej zostanie on wdrożony.

Interfejs programowania aplikacji jest usługą, której funkcjonowanie jest nieodłącznie powiązane z serwerem internetowym. Serwer sieci Web pełni rolę warstwy opakowującej, wewnątrz której działać może interfejs API. Wdrożenie rozważanej usługi sieciowej w ramach sieci Internet, wiąże się w związku z tym z uruchomieniem serwera sieci web w ramach komputera eksponowanego w sieci rozległej. Należy również zauważyć konieczność zastosowania hipertekstowego serwera pośredniczącego, po to, aby klient usługi, mógł się z nią komunikować z wykorzystaniem protokołu HTTP.

System informatyczny, składający się z przedstawionych powyżej komponentów może zostać uruchomiony w ramach wirtualnego serwera prywatnego, udostępnianego przez określonego dostawcę infrastruktury serwerowej. Model taki, definiowany jest jako infrastruktura w postaci usługi klienckiej (*ang. Infrastructure as a Service*). Ponadto, przygotowane oprogramowanie może zostać wdrożone na dedykowanej określonej technologii, platformie chmurowej. W ramach platformy tej, użytkownik, za pomocą dostarczonego interfejsu komunikacji może wdrażać oraz konfigurować działanie swojego oprogramowania. Taki model dostarczania zasobu z kolei, nazywany jest platformą w postaci usługi klienckiej (*ang. Platform as a Service*).

Niniejszy aspekt problemu badawczego, dotyczy porównania wydajności API, w zależności od jego wdrożenia na generycznym wirtualnym serwerze prywatnym, a także dedykowanej platformie chmurowej, dostosowanej pod kątem określonego środowiska uruchomieniowego oraz języka programowania.

Wskaźnik ewaluacji efektywności działania interfejsu programowania aplikacji, obejmuje te same metryki, które przedstawione zostały w pierwszym spośród omawianych aspektów problemu badawczego. Kryterium czasu odpowiedzi na żądanie, musi zostać jednakże uniezależnione od niedeterministycznego charakteru łącza internetowego, dlatego też, ten właśnie parametr, będzie dotyczył czasu od momentu otrzymania żądania przez API, do chwili wygenerowania odpowiedzi na żądanie.

3.2. Sformułowanie scenariuszy badawczych

Na podstawie przedstawionych w poprzednim podrozdziale aspektów problemu badawczego, sformułowane zostały konkretne scenariusze badawcze. W każdym ze scenariuszy, zdefiniowano zbiór czynności wykonywanych w ramach określonego badania, wyszczególniono kryteria porównawcze dla danej obserwacji, wymieniono dostosowywalne parametry badania, a także skonkretyzowano czynności, które muszą zostać podjęte, jako warunki konieczne przed wykonaniem badania. Każdy ze scenariuszy badawczych odzwierciedlony został w formie tabeli.

Tab. 3.1: Scenariusz badawczy - badanie przeprowadzone w kontekście systemów bazodanowych

Nazwa scenariusza badawczego:
Wpływ zastosowanego systemu bazodanowego na efektywność działania interfejsu programowania aplikacji przy zmiennej liczbie żądań klientów
Topologia fizyczna środowiska badawczego:

Konfiguracja pierwsza lokalnego środowiska badawczego 4.1

Czynności implementacyjne:

- Zaimplementowanie interfejsów programowania aplikacji w technologiach C#/.NET oraz NodeJS/ExpressJS.
- Konfiguracja interfejsów programowania aplikacji w celu obsługi systemów bazodanowych: Microsoft SQL Server, MySQL, PostgreSQL, SQLite oraz MongoDB.
- Konfiguracja topologii fizycznej środowiska badawczego.
- Konfiguracja narzędzia do wykonywania testów wydajnościowych zgodnie z planem testowym 4.4.

Czynności badawcze:

- Realizacja testów wydajnościowych z uwzględnieniem zmienności liczby klientów.
- Obserwacja oraz gromadzenie wartości pomiarowych dotyczących kryteriów porównawczych.
- Dostosowywanie wartości parametrów przeprowadzanego badania.

Warunki początkowe podjęcia czynności badawczych:

Przed realizacją testów wydajnościowych zagwarantowana zostanie poprawność działania każdego z interfejsów programowania aplikacji poprzez wykonanie ewaluacji funkcjonalnej.

Opis scenariusza badawczego:

Po wykonaniu konfiguracji topologii fizycznej środowiska badawczego, dwa urządzenia klienckie będą wysyłać żądania protokołu hipertekstowego w kierunku interfejsu programowania aplikacji. Liczba żądań tych będzie sukcesywnie zwiększana, poprzez uruchamianie kolejnych równolegle pracujących wątków oprogramowania testowego. Dla każdej wartości liczby urządzeń klienckich, gromadzonych będzie co najmniej 30 próbek, a uzyskane wyniki zostaną następnie uśrednione. Odpowiednie przedziały wartości omawianego parametru wyznaczać będą granicę pomiędzy testami linii bazowej, obciążeniowymi, a także przeciążającymi. Badanie zostanie powtórzone dla każdego z rozważanych systemów bazodanowych w obrębie obu porównywanych technologii programistycznych.

Kryteria porównawcze:

- Czas odpowiedzi interfejsu programowania aplikacji na żądanie klienta.
- Procent odpowiedzi błędnych, uzyskanych względem wszystkich rezultatów.

Parametry badania:

- Liczba klientów równolegle wysyłających żądania.
- Technologia programistyczna zastosowana do implementacji interfejsu programowania aplikacji.
- Rodzaj systemu bazodanowego komunikującego się z interfejsem programowania aplikacji.
- Rodzaj operacji protokołu hipertekstowego.

Tab. 3.2: Scenariusz badawczy - badanie przeprowadzone w kontekście realizacji operacji współbieżnych

Nazwa scenariusza badawczego:

Wpływ zastosowanej technologii programistycznej na wydajność realizacji operacji współbieżnych

Topologia fizyczna środowiska badawczego:

Konfiguracja druga lokalnego środowiska badawczego 4.1

Czynności implementacyjne:

- Zaimplementowanie genetycznego algorytmu metaheurystycznego dla symetrycznego problemu komiwojażera w językach programowania C# oraz JavaScript.
- Zaimplementowanie mechanizmów pomiaru czasu wykonania algorytmu.
- Konfiguracja interfejsów programowania aplikacji w celu obsługi algorytmów metaheurystycznych z poziomu punktu końcowego API.
- Konfiguracja topologii fizycznej środowiska badawczego.
- Konfiguracja narzędzia do wykonywania testów wydajnościowych zgodnie z planem testowym 4.4.

Czynności badawcze:

- Realizacja testów wydajnościowych dla porównywanych technologii programistycznych.
- Obserwacja oraz gromadzenie wartości pomiarowych dotyczących kryteriów porównawczych.
- Dostosowywanie wartości parametrów przeprowadzanego badania.
- Analiza statystyczna otrzymanych wyników.

Warunki początkowe podjęcia czynności badawczych:

Przed realizacją testów wydajnościowych zostanie poprawność działania każdego z algorytmów metaheurystycznych. Ponadto, kod źródłowy programów implementujących algorytmy zostanie przekształcony w taki sposób, aby niezależnie od języka programowania, realizował operacje w sposób analogiczny.

Opis scenariusza badawczego:

Po wykonaniu konfiguracji topologii fizycznej środowiska badawczego, pojedyncze urządzenia klienckie będzie wysyłać z ustaloną w planie testowym częstotliwością, żądania wykonania algorytmu. Po odebraniu zapytania od klienta, algorytm będzie uruchamiany, a czas trwania obliczeń będzie zawsze wartością stałą. W trakcie wykonywanych kalkulacji zliczana będzie liczba iteracji głównej pętli kodu algorytmu. Liczba ta, a także współczynnik jakości uzyskanego rezultatu względem rozwiązania optymalnego stanowić będą kryteria porównawcze. Dla każdej z porównywanych technologii programistycznych, w obrębie czterech różnych czasów działania głównej pętli algorytmu, wykonana zostanie seria piętnastu cyklicznych żądań klienta. Po zgromadzeniu wyników badań przeprowadzone zostaną parowe testy statystyczne wykazujące istotność różnic pomiarowych.

Kryteria porównawcze:

- Liczba iteracji głównej pętli algorytmu metaheurystycznego.
- Stosunek uzyskanego rezultatu obliczeń względem rozwiązania optymalnego.

Parametry badania:

- Technologia programistyczna zastosowana do implementacji interfejsu programowania aplikacji.
- Czas wykonywania głównej pętli programu.

Tab. 3.3: Scenariusz badawczy - badanie przeprowadzone w kontekście obsługi operacji asynchronicznych

Nazwa scenariusza badawczego:

Wpływ zastosowanej technologii programistycznej na efektywność obsługi operacji asynchronicznych

Topologia fizyczna środowiska badawczego:

Konfiguracja trzeciego lokalnego środowiska badawczego 4.1

Czynności implementacyjne:

- Zaimplementowanie interfejsów programowania aplikacji w technologiach C#/.NET, NodeJS/ExpressJS oraz Python/Flask.
- Zdefiniowanie punktów końcowych odpowiedzialnych za komunikację badanego interfejsu z zewnętrznym API.
- Konfiguracja topologii fizycznej środowiska badawczego.
- Konfiguracja narzędzia do wykonywania testów wydajnościowych zgodnie z planem testowym 4.4.

Czynności badawcze:

- Realizacja testów wydajnościowych dla porównywanych technologii programistycznych.
- Obserwacja oraz gromadzenie wartości pomiarowych dotyczących kryteriów porównawczych.
- Dostosowywanie wartości parametrów przeprowadzanego badania.

Warunki początkowe podjęcia czynności badawczych:

Przed realizacją testów wydajnościowych zagwarantowane zostanie poprawne połączenie pomiędzy każdym z badanych interfejsów programowania aplikacji a zewnętrzną usługą sieciową. Co więcej, zweryfikowana zostanie poprawność implementacji funkcjonalności zewnętrznej usługi sieciowej (tj. interfejsu API zaimplementowanego w języku Python)

Opis scenariusza badawczego:

Po wykonaniu konfiguracji topologii fizycznej środowiska badawczego, urządzenia klienckie będą równolegle wysyłać żądania http w kierunku interfejsu programowania aplikacji. Po odbioru zapytania od klienta, interfejs API, łączyć się będzie ze znajdującą się w obrębie sieci lokalnej zewnętrzną usługą sieciową. Taki rodzaj operacji, kiedy strona wywołująca zleca wykonanie zadania, a odpowiedź na to zlecenie przyjść może w dowolnym momencie, nazywamy operacją asynchroniczną. W ramach punktu końcowego ewaluowanego API, wykonana zostanie trzydziestokrotnie instrukcja iteracyjna, wewnątrz której natywny klient protokołu hipertekstowego zostanie wywołany w celu pozyskania list obiektów bazodanowych dla sześciu różnych liczebności. Interfejs programowania aplikacji zwracać będzie odpowiedź zawierającą informację o stopniu poprawności wykonania zleconych operacji, a także czas odpowiedzi na żądanie. Tak zdefiniowane badanie, będzie wykonywane dla określonej w planie testowym liczby klientów równolegle generujących żądania.

Kryteria porównawcze:

- Czas odpowiedzi interfejsu programowania aplikacji na żądanie klienta.
- Procent poprawności wykonania zleconych operacji asynchronicznych.
- Liczba obiektów bazodanowych pozyskiwanych w ramach żądania.

Parametry badania:

- Liczba klientów równolegle wysyłających żądania.
- Technologia programistyczna zastosowana do implementacji interfejsu programowania aplikacji.

Tab. 3.4: Scenariusz badawczy - badanie przeprowadzone w kontekście zastosowania wzorca projektowego CQRS

Nazwa scenariusza badawczego:
Wpływ implementacji wzorca projektowego podziału odpowiedzialności na wydajność obsługi żądania klienta
Topologia fizyczna środowiska badawczego:
Konfiguracja czwarta lokalnego środowiska badawczego 4.1
Czynności implementacyjne:
<ul style="list-style-type: none"> • Zaimplementowanie interfejsów programowania aplikacji wykorzystujących wzorzec projektowy CQRS w technologiach C#/.NET oraz NodeJS/ExpressJS. • Konfiguracja interfejsów programowania aplikacji w celu obsługi systemu bazodanowego: Microsoft SQL Server. • Wprowadzenie usprawnień wydajnościowych dotyczących modelu danych przeznaczonego dla operacji odczytu. • Połączenie interfejsu API z dwoma systemami bazodanowymi tego samego typu i konfiguracja automatycznej, transakcyjnej replikacji danych w momencie ich zapisu. • Konfiguracja topologii fizycznej środowiska badawczego. • Konfiguracja narzędzia do wykonywania testów wydajnościowych zgodnie z planem testowym 4.4.
Czynności badawcze:
<ul style="list-style-type: none"> • Realizacja testów wydajnościowych z uwzględnieniem zmienności liczby klientów. • Obserwacja oraz gromadzenie wartości pomiarowych dotyczących kryteriów porównawczych. • Dostosowywanie wartości parametrów przeprowadzanego badania.
Warunki początkowe podjęcia czynności badawczych:
Przed realizacją testów wydajnościowych zagwarantowana zostanie poprawność działania każdego z interfejsów programowania aplikacji poprzez wykonanie ewaluacji funkcjonalnej.
Opis scenariusza badawczego:

Po wykonaniu konfiguracji topologii fizycznej środowiska badawczego, urządzenia klienckie będą wysyłać żądania protokołu hipertekstowego w kierunku interfejsu programowania aplikacji. Liczba żądań tych będzie sukcesywnie zwiększać się, poprzez uruchamianie kolejnych równolegle pracujących wątków oprogramowania testowego. Dla każdej wartości liczby urządzeń klienckich, badanie zostanie wykonane minimum trzydziestokrotnie, a uzyskane wyniki zostaną następnie uśrednione. Odpowiednie przedziały wartości omawianego parametru wyznaczać będą granicę pomiędzy testami linii bazowej, obciążeniowymi, a także przeciążającymi. Badanie zostanie powtórzone dla każdego z rozważanych systemów bazodanowych w obrębie obu porównywanych technologii programistycznych. Obserwacje uzyskane w ramach niniejszego badania, porównane zostaną z tymi, ustalonimi na podstawie pierwszego scenariusza badawczego 3.1.

Kryteria porównawcze:

- Czas odpowiedzi interfejsu programowania aplikacji na żądanie klienta.
- Procent odpowiedzi błędnych, uzyskanych względem wszystkich rezultatów.

Parametry badania:

- Rodzaj operacji protokołu hipertekstowego.
- Liczba klientów równolegle wysyłających żądania.
- Technologia programistyczna zastosowana do implementacji interfejsu programowania aplikacji.

Tab. 3.5: Scenariusz badawczy - badanie przeprowadzone w kontekście wykorzystania mechanizmów pamięci podręcznej

Nazwa scenariusza badawczego:

Porównanie efektywności obsługi żądań klienckich w stałym czasie, uwzględniając odmienne implementacje mechanizmów pamięci podręcznej

Topologia fizyczna środowiska badawczego:

Konfiguracja druga lokalnego środowiska badawczego 4.1

Czynności implementacyjne:

- Zaimplementowanie interfejsów programowania aplikacji w technologiach C#/.NET oraz NodeJS/ExpressJS.
- Zaimplementowanie mechanizmów pamięci podręcznej (rozwiążanie autorskie oraz standar-dowe) w oparciu o bibliotekę Redis.
- Konfiguracja interfejsów programowania aplikacji w celu obsługi systemu bazodanowego MySQL.
- Konfiguracja topologii fizycznej środowiska badawczego.
- Konfiguracja narzędzia do wykonywania testów wydajnościowych zgodnie z planem testo-wym 4.4.

Czynności badawcze:

- Realizacja testów wydajnościowych z uwzględnieniem zmienności czasu trwania testu.
- Obserwacja oraz gromadzenie wartości pomiarowych dotyczących kryteriów porównaw-czych.
- Dostosowywanie wartości parametrów przeprowadzanego badania.

Warunki początkowe podjęcia czynności badawczych:

Przed realizacją testów wydajnościowych zagwarantowana zostanie poprawność działania każ-dego z interfejsów programowania aplikacji, a także każdego z zaimplementowanych mecha-nizmów pamięci podręcznej. Do autorskiego mechanizmu pamięci cache dostarczone zostaną przygotowane informacje historyczne, których zawartość dotyczy częstotliwości wywoływania określonych punktów końcowych interfejsu programowania aplikacji.

Opis scenariusza badawczego:

Po wykonaniu konfiguracji topologii fizycznej środowiska badawczego, pojedyncze urządzenie klienckie będzie w sposób sekwencyjny wysyłać żądania do zdefiniowanego zbioru punktów końcowych. Interfejsy programowania aplikacji, które implementować będą określone mecha-nizmy pamięci podręcznej, zwracać będą odpowiedź na żądanie, której wartość różnić będzie się w sposób znaczący, w zależności od tego, czy API musiało odwołać się do systemu bazoda-nowego, czy też pozyskać dane z cache. Pierwszy z systemów pamięci podręcznej uwzględniać będzie stały czas ważności wpisu, natomiast drugi system (tj. system autorski) wyliczać będzie czas ważności na podstawie częstotliwości odwołań do punktu końcowego. Zebrane rezultaty rozpatrywane będą dla stałego przedziału czasu trwania testu, a także różnych momentów wy-wołań stałej liczby żądań unieważniających. W każdym z przedziałów, wydajność systemu pa-mięci podręcznej determinowana będzie przez średni czas odpowiedzi na żądanie oraz liczbę odwołań których długość realizacji przekracza 500ms. Badanie zostanie powtórzone w obrębie obu porównywanych technologii programistycznych.

Kryteria porównawcze:

- Średni czas odpowiedzi interfejsu programowania aplikacji na żądanie klienta w stałym prze- dziale czasu.
- Liczba żądań, dla których czas odpowiedzi jest większy niż 500 milisekund.

Parametry badania:

- Technologia programistyczna zastosowana do implementacji interfejsu programowania aplikacji.
- Rodzaj zaimplementowanego mechanizmu przechowywania danych w pamięci podręcznej.
- Rozmieszczenie stałej liczby punktów unieważnień wpisów pamięci podręcznej.

Tab. 3.6: Scenariusz badawczy - badanie przeprowadzone w kontekście wdrażania oprogramowania na platformach chmurowych

Nazwa scenariusza badawczego:

Zmienna wydajności interfejsu API wdrożonego na generycznej oraz dedykowanej platformie chmurowej

Topologia fizyczna środowiska badawczego:

Konfiguracja pierwsza rozproszonego środowiska badawczego 4.1

Czynności implementacyjne:

- Zaimplementowanie interfejsów programowania aplikacji w technologiach C#/.NET oraz NodeJS/ExpressJS.
- Konfiguracja interfejsów programowania aplikacji w celu obsługi dedykowanych systemów bazodanowych.
- Wdrożenie interfejsów programowania aplikacji na wirtualnych serwerach prywatnych.
- Wdrożenie interfejsów programowania aplikacji na dedykowanych względem określonej technologii platformach chmurowych.
- Implementacja mechanizmów pomiaru czasu wykonywanych operacji wewnętrz interfejsów API.
- Konfiguracja topologii fizycznej rozproszonego środowiska badawczego.
- Konfiguracja narzędzia do wykonywania testów wydajnościowych.

Czynności badawcze:

- Realizacja testów wydajnościowych z uwzględnieniem zmiennej liczby klientów API.
- Obserwacja oraz gromadzenie wartości pomiarowych dotyczących kryteriów porównawczych.
- Dostosowywanie wartości parametrów przeprowadzanego badania.
- Analiza statystyczna otrzymanych wyników.

Warunki początkowe podjęcia czynności badawczych:

Przed realizacją testów wydajnościowych zagwarantowana zostanie poprawność działania każdego z interfejsów programowania aplikacji poprzez wykonanie ewaluacji funkcjonalnej. Ponadto, zweryfikowana zostanie dostępność każdej z platform chmurowych w czasie wykonywania testów.

Opis scenariusza badawczego:

Po wykonaniu konfiguracji topologii fizycznej rozproszonego środowiska badawczego, urządzenia klienckie będą równolegle generowały żądania hipertekstowe w kierunku interfejsu programowania aplikacji. Poszczególny interfejs API, analogicznie do obsługiwanej przez niego systemu bazodanowego, wdrożony zostanie w określonym środowisku chmurowym. Od momentu uzyskania żądania od aplikacji klienckiej, zliczany będzie czas wykonywania operacji wewnętrz API. Punktem końcowym czasu realizacji obliczeń będzie chwila wygenerowania odpowiedzi na żądanie. Czas odpowiedzi liczony w ten sposób, będzie odpowiednio pomniejszony względem standardowego czasu odpowiedzi na żądanie i nie będzie on uwzględniał faktu dostarczenia oraz zwrócenia wiadomości http w ramach sieci rozległej. Badanie zostanie powtórzone w obrębie obu porównywanych technologii programistycznych, uwzględniając dwa systemy bazodanowe. Systemami tymi są MySQL (w kontekście rozwiązań generycznych), a także Microsoft SQL Server oraz MongoDB w kontekście rozwiązań dedykowanych. Po zgromadzeniu wyników badań przeprowadzone zostaną parowe testy statystyczne wykazujące istotność różnic pomiarowych.

Kryteria porównawcze:

- Czas obsługi żądania wewnętrz interfejsu programowania aplikacji.
- Liczba żądań dla których uzyskano niepoprawny format odpowiedzi.

Parametry badania:

- Rodzaj operacji protokołu hipertekstowego.
- Wykorzystywana platforma chmurowa oraz wirtualny serwer prywatny.

Rozdział 4

Implementacja środowiska badawczego

W niniejszym rozdziale przedstawiono proces implementacji środowiska badawczego pod kątem realizacji zdefiniowanych uprzednio scenariuszy badawczych. Spośród składowych omawianego procesu wyróżnić należy: przygotowanie odmian topologii fizycznych środowiska w zależności od scenariusza badawczego, budowę interfejsów programowania aplikacji z wykorzystaniem porównywanych technologii, czy też zastosowanie mechanizmów pozwalających na weryfikację działania API w aspekcie programowania wspólnienego. Ponadto, zaprezentowana została konfiguracja dedykowanych porównywanym technologiom platform chmurowych, a także struktura elementów składających się na plan testowy, identyfikujący przeprowadzoną ewaluację w ramach narzędzia przeznaczonego do realizacji badań.

4.1. Realizacja topologii fizycznych

W zależności od scenariusza badawczego, wykorzystywanego w kontekście przeprowadzonych ewaluacji, zbudowanych zostało pięć odmiennych wariantów topologii fizycznych. Cztery spośród nich, tyczą się badań wykonywanych w środowisku lokalnym, natomiast piąta z topologii, związana jest z obserwacją funkcjonowania interfejsów programowania aplikacji uruchomionych w obrębie określonych platform chmurowych.

W odniesieniu do każdej z topologii zbudowanej w środowisku lokalnym, zauważać należy fakt zastosowania techniki rozproszonego testowania (*ang. Distributed Testing*), a także jasny podział odpowiedzialności w kontekście wszystkich wykorzystywanych urządzeń. Procedura zbierania obserwacji za każdym razem realizowana jest wewnątrz odpowiednio dostosowanej lokalnej sieci komputerowej cechującej się brakiem dostępu do sieci Internet. Ponadto, w obszarze połączonych ze sobą urządzeń systemu komputerowego, dezaktywowane zostały protokoły i usługi sieciowe generujące cykliczne komunikaty rozgłoszeniowe. Dzięki temu, wyeliminowano błędy pomiarowe o charakterze niedeterministycznym.

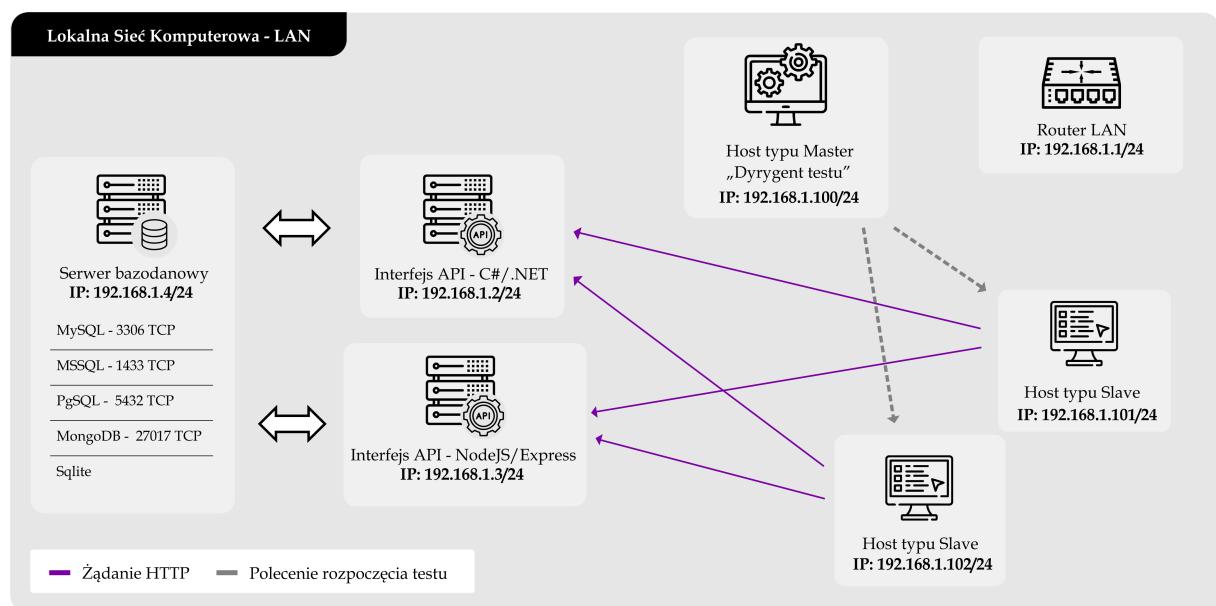
Odwołując się do topologii sieciowej zbudowanej w środowisku rozległym, zauważać należy odmienny sposób pozyskiwania informacji o czasie przetwarzania pojedynczego żądania. W tym przypadku, wykorzystywane narzędzie pomiarowe służy jako generator żądań, jednakże czasy odpowiedzi na żądanie, zwrócone przez to narzędzie nie mogą być brane pod uwagę. Informacja o czasie przetwarzania żądania zostaje dostarczana bezpośrednio z interfejsu programowania aplikacji.

Konfiguracja pierwsza lokalnej topologii fizycznej środowiska badawczego

Pierwsza spośród lokalnych topologii fizycznych środowiska badawczego zastosowana została w celu badania wpływu wykorzystania odmiennych systemów bazodanowych na wydajność interfejsów programowania aplikacji.

W ramach niniejszej topologii, wyróżnić należy dwa interfejsy programowania aplikacji (tj. utworzone z wykorzystaniem technologii C#/NET oraz NodeJS/Express), komunikujące się z jednym z pięciu serwerów bazodanowych (tj. MySQL Server, PostgreSQL Server, Microsoft SQL Server, SQLite oraz MongoDB). W określonym momencie czasu, utrzymywane jest tylko jedno aktywne połączenie pomiędzy jednym z API a jednym z serwerów bazodanowych. Ponadto, wewnątrz lokalnej sieci komputerowej, wyróżnić należy urządzenie komunikacyjne którym jest router, a także trzy urządzenia końcowe. Pierwszy z hostów pełni rolę "dyrygenta testu", który dostarcza informacje o konfiguracji testowej bezpośrednio do dwóch pozostałych urządzeń końcowych. Te urządzenia z kolei, odpowiedzialne są za generowanie żądań zgodnie ze zdefiniowanym natężeniem, częstotliwością, a także czasem trwania ewaluacji. Każde z oddzielnych urządzeń fizycznych połączone jest z urządzeniem komunikacyjnym poprzez łącze przewodowe, o tej samej przepustowości (tj. 1Gb/s), a także identycznej kategorii przewodu (tj. kategoria 6).

Na ilustracji 4.1 przedstawiono schemat pierwszego wariantu lokalnej topologii fizycznej środowiska badawczego.



Rys. 4.1: Konfiguracja pierwsza lokalnej topologii fizycznej środowiska badawczego

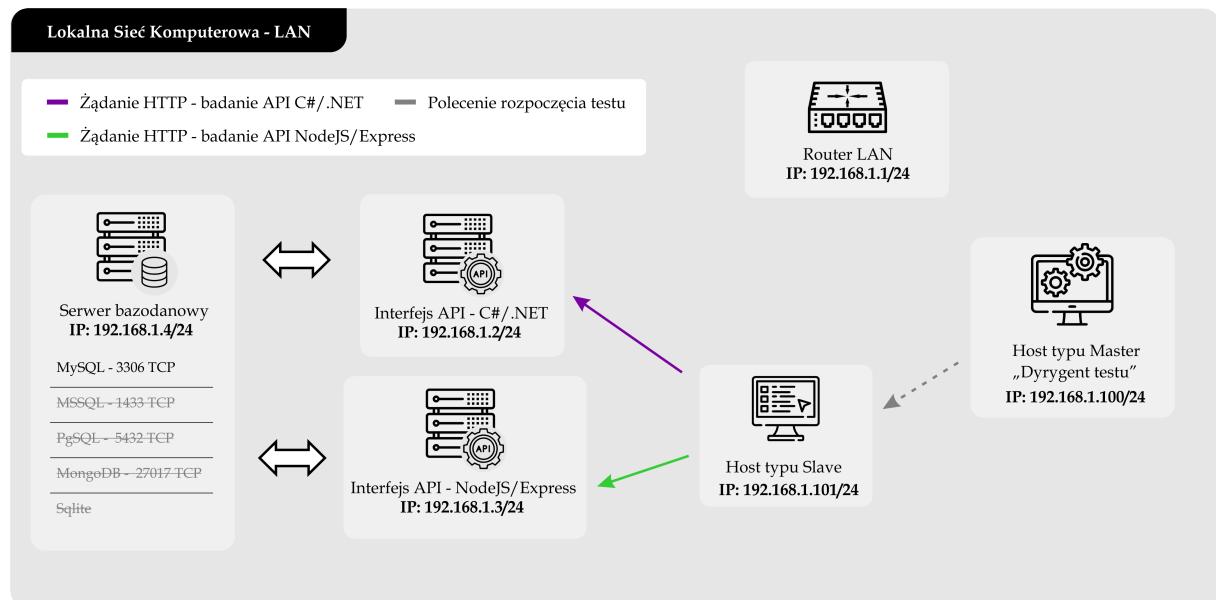
Konfiguracja druga lokalnej topologii fizycznej środowiska badawczego

Druga topologia fizyczna środowiska badawczego, dotycząca ewaluacji w obrębie lokalnej sieci komputerowej, zbudowana została na potrzeby badań zaimplementowanych mechanizmów programowania współbieżnego, a także przechowywania odpowiedzi na żądania w ramach pamięci podręcznej.

W omawianej topologii wskazać należy dwa interfejsy programowania aplikacji, których implementacja dokonana została w porównywanych dwóch technologiach programistycznych. Obie usługi sieciowe, komunikują się z pojedynką instancją serwera bazodanowego - w przypadku badania pamięci cache, bądź też nie odwołując się do niego wcale - w przypadku badania

efektywności operacji współbieżnych. Spośród urządzeń komunikacyjnych, wykorzystywanych do przeprowadzenia ewaluacji, wyróżnić należy jedno urządzenie typu Master (tzw. Dyrygent testu), a także jednego hosta typu Slave (tzw. generator żądań). Pierwszy z komputerów ma za zadanie przechowywać konfigurację wykonywanego badania, a także dostarczać komendy związane z rozpoczęciem i charakterystyką testu. Drugi host pełni rolę maszyny wytwarzającej i wysyłającej żądania protokołu hipertekstowego, zgodnie z koncepcją nakreślona przez uznany plan ewaluacji. Wszystkie urządzenia znajdują się w obszarze pojedynczej, przewodowej lokalnej sieci komputerowej. Analogicznie do konfiguracji pierwszej, każde łącze przewodowe charakteryzuje się tym samym standardem oraz przepustowością.

Na ilustracji 4.2 przedstawiono schemat drugiego wariantu lokalnej topologii fizycznej środowiska badawczego.



Rys. 4.2: Konfiguracja druga lokalnej topologii fizycznej środowiska badawczego

Konfiguracja trzecia lokalnej topologii fizycznej środowiska badawczego

Trzecia lokalna topologia fizyczna środowiska badawczego przystosowana została w celu umożliwienia realizacji badań dotyczących wydajności obsługi operacji asynchronicznych.

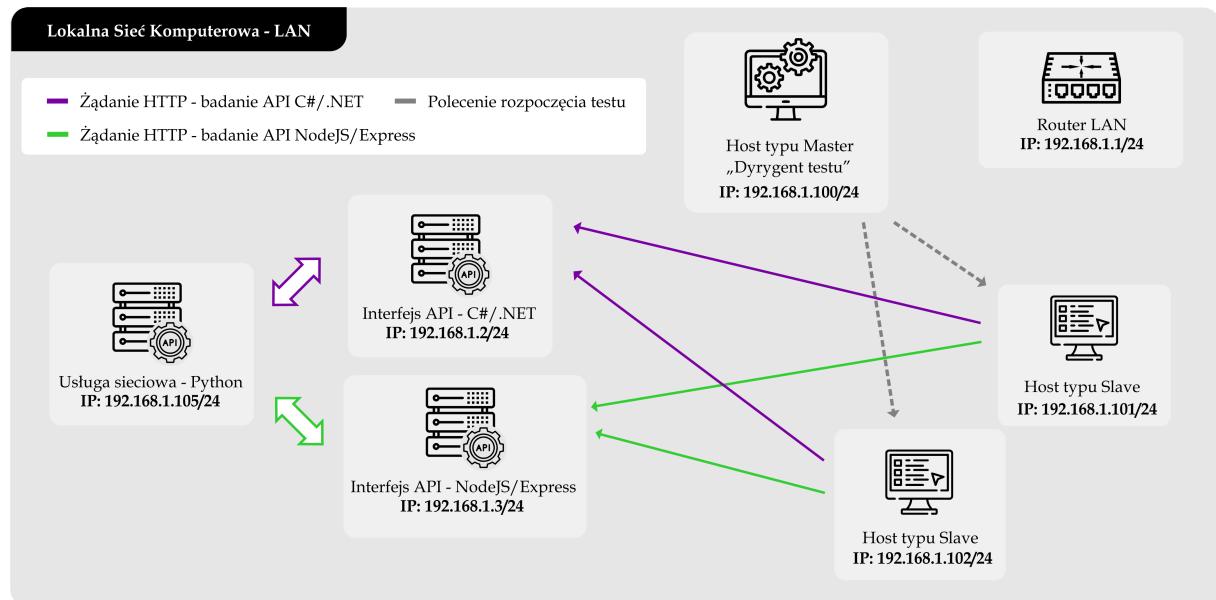
W schemacie tym, zauważać można wystąpienie trzech interfejsów programowania aplikacji. Analogicznie do topologii opisanych powyżej, dwa spośród trzech API zaimplementowane są w technologiach będących przedmiotem analizy tej pracy. Trzecia usługa sieciowa, służy do udostępniania zgromadzonych w niej danych, w związku z czym posiada ona tylko i wyłącznie punkty końcowe obsługiwane z wykorzystaniem metody GET. Punkty styku ostatniego z interfejsów dostarczają funkcjonalności pobierania danych o zróżnicowanym rozmiarze.

Ponadto, zbieżnie do konfiguracji pierwszej lokalnego środowiska badawczego, wyszczególnić możemy dwa urządzenia końcowe w roli generatorów żądań, oraz jednego hosta działającego w trybie "Dyrygenta testu". Należy podkreślić, że ewaluacje dotyczące każdej z technologii, zarówno w scenariuszach badawczych wykorzystujących tę, jak i pozostałe topologie, wykonywane są w odrębnych chwilach czasu. Implikuje to fakt, że połączenie pomiędzy hostem badającym, interfejsem badanym, a także interfejsem pomocniczym jest aktywne tylko dla aktualnie badanego rozwiązania technologicznego.

Poza interfejsami programowania aplikacji oraz urządzeniami końcowymi wskażać należy urządzenie sieciowe, jakim jest przewodowy router LAN, który łączy wszystkie elementy topo-

logii w ramach pojedynczej sieci LAN. Standard oraz przepustowość wykorzystywanych łączy pozostaje niezmienna względem pierwszej oraz drugiej z topologii fizycznych.

Na ilustracji 4.3 przedstawiono schemat trzeciego wariantu lokalnej topologii fizycznej środowiska badawczego.



Rys. 4.3: Konfiguracja trzecia lokalnej topologii fizycznej środowiska badawczego

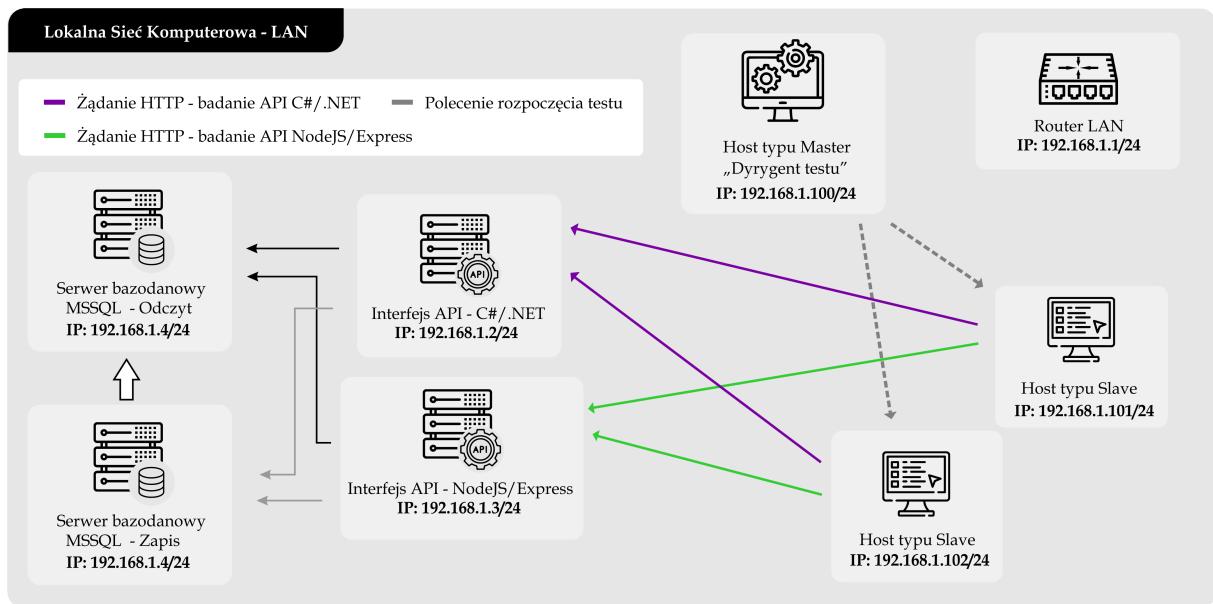
Konfiguracja czwarta lokalnej topologii fizycznej środowiska badawczego

Ostatnia z lokalnych topologii fizycznych środowiska badawczego zbudowana została w kontekście przeprowadzenia badania efektywności interfejsów programowania aplikacji w związku z zastosowaniem wzorca podziału odpowiedzialności.

Wykorzystanie tego wzorca, umożliwia separację zarówno modeli danych, jak i również fizycznych struktur bazodanowych. Dlatego też, w ramach omawianej topologii zdecydowano się na wprowadzenie dwóch oddzielnych serwerów baz danych. Serwer bazy danych wykorzystywany do zapisu implementuje operację replikacji transakcyjnej dzięki czemu, po wykonaniu zapisu do jednego źródła danych, zapisane informacje są automatycznie przenoszone do drugiej z baz. W ramach połączenia z serwerem bazodanowym wykorzystywany do odczytu, po stronie interfejsów programowania aplikacji dokonano przystosowania modelu danych.

Poza serwerami danych, w ramach topologii wyróżnić należy dwa porównywane interfejsy programowania aplikacji, a także zbiór trzech urządzeń końcowych. Dwa spośród nich pełnią rolę generatorów żądań, natomiast trzeci host pracuje w trybie "dyrygenta testu". Wszystkie z urządzeń połączone są w obrębie lokalnej sieci komputerowej do urządzenia sieciowego którym jest router LAN.

Na ilustracji 4.4 przedstawiono schemat czwartego wariantu lokalnej topologii fizycznej środowiska badawczego.



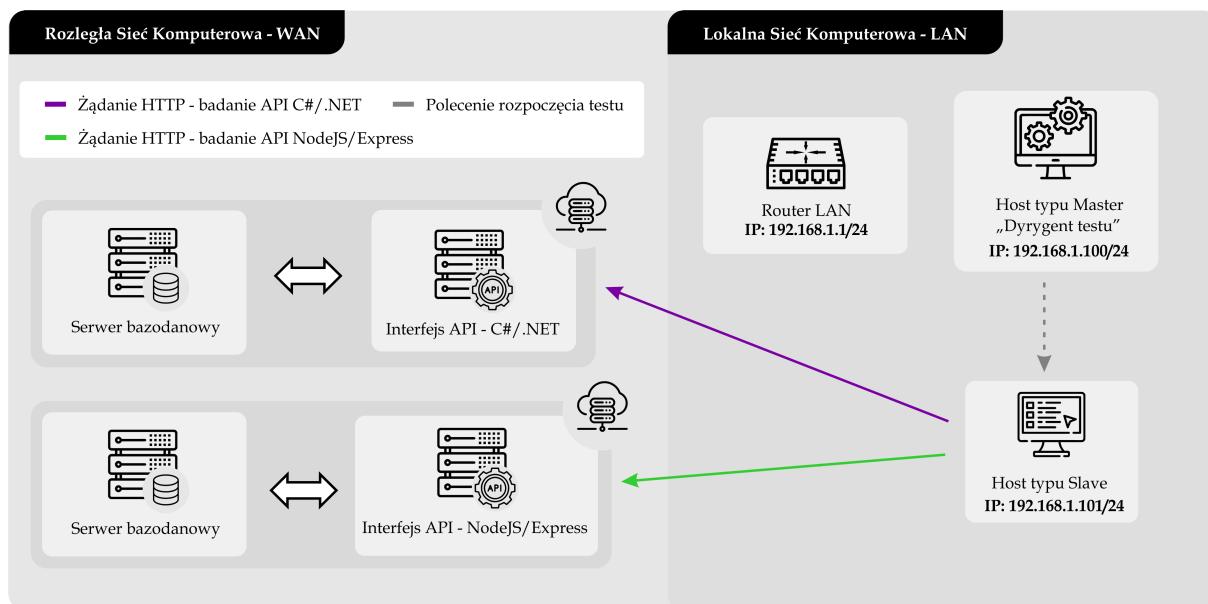
Rys. 4.4: Konfiguracja czwarta lokalnej topologii fizycznej środowiska badawczego

Konfiguracja pierwsza rozległej topologii fizycznej środowiska badawczego

Konfiguracja topologii fizycznej dla rozproszonego środowiska badawczego, stworzona została w celu wykonania badań dotyczących efektywności działania interfejsów programowania aplikacji uruchamianych na odmiennych platformach chmurowych.

W tym przypadku, wyodrębnić należy dwa obszary, zawierające urządzenia w kontekście których przeprowadzana jest ewaluacja. Pierwszy obszar to lokalna sieć komputerowa, w której ulokowane zostały urządzenia końcowe odpowiedzialne za przechowywanie konfiguracji badania, a także generowanie żądań w kierunku API. W drugim obszarze zaś, nazwanym rozległą siecią komputerową uruchomione są platformy chmurowe, wewnętrznych których działają interfejsy programowania aplikacji oraz serwery bazodanowe. Systemy internetowe zaimplementowane w odrębnych technologiach, przechowywane są na oddzielnego platformach chmurowych i nie są one ze sobą w żaden sposób skomunikowane. Zauważać należy również fakt, że obie platformy chmurowe znajdują się w różnych lokalizacjach geograficznych. Stwierdzenia te, wymuszają modyfikację sposobu pozyskiwania pomiarów żądań, poprzez przeniesienie odpowiedzialności za wyliczenie czasu wykonywania operacji na interfejsy programowania aplikacji. Pomimo tego, niezbędnym jest posiadania urządzenia generującego żądania zgodnie z określona charakterystyką. Dlatego też, wewnątrz obszaru lokalnego wskazać możemy dwa urządzenia końcowe, których rolą, podobnie do urządzeń końcowych zawartych we wszystkich poprzednich topologii fizycznych, jest generowanie żądań, oraz koordynowanie przeprowadzanego badania.

Na ilustracji 4.5 przedstawiono schemat pierwszego wariantu rozległej topologii fizycznej środowiska badawczego.



Rys. 4.5: Konfiguracja pierwsza rozległej topologii fizycznej środowiska badawczego

4.2. Budowa interfejsów programowania aplikacji

Ogólna funkcjonalność interfejsów programowania aplikacji

Niezależnie od rozważanej technologii, interfejsy programowania aplikacji implementowane w celu przeprowadzania badań zawartych w ramach niniejszej pracy magisterskiej, dostarczają funkcjonalności obsługi lokalu restauracyjnego. W obrębie omawianych aplikacji zdefiniować można konta użytkowników należących do jednej z pięciu ról pracowników. Wyróżnić należy stanowiska: administratora systemu, zarządcy przedsiębiorstwem, kierownika sali, kucharza, a także kelnera. Użytkownicy dysponujący jedną z wymienionych ról mogą korzystać z określonego zestawu funkcjonalności, który determinowany jest poziomem uprawnień zależnych od stanowiska. Przed wywołaniem funkcjonalności systemu, każdy z użytkowników musi zostać uwierzytelniony, a także autoryzowany. Uwierzytelnienie polega na odwołaniu się do punktu końcowego logowania i podaniu swoich danych poświadczania. Jeżeli dane poświadczania wskazują na konkretnego użytkownika API, w odpowiedzi zwracany jest token uwierzytelniający, który musi zostać następnie dołączony do nagłówka każdego z wywoływanych żądań. Proces autoryzacji odbywa się na poziomie funkcji warstwy pośredniczącej (*ang. Middleware*), tuż przed rozpoczęciem wykonywania kodu metody klasy kontrolera.

W kontekście funkcjonalności tworzonych usług sieciowych wymienić należy:

- definiowanie oraz zarządzanie strukturą lokalu restauracyjnego (zarządzanie pomieszczeniami lokalu, określanie układu stolików wewnętrz pojedynczego pomieszczenia)
- obsługa procesu zamówienia (zarządzanie rachunkami, danymi, statusami przetwarzania posiłków, a także dyspozycjami klientów)
- zarządzanie gospodarką magazynową lokalu restauracyjnego (obserwacja stanów ilościowych w kontekście produktu, obsługa danych produktu, definiowanie kategorii produktowych).

Dodatkowe funkcje systemu, nie związane z przeprowadzaniem operacji typu CRUD, takie jak wdrożenie metody wyznaczania trasy dla dostarczania zamówień poprzez implementację algorytmu dla symetrycznego problemu komiwojażera, opisane zostały w następnych sekcjach niniejszego rozdziału.

Interfejs API realizujący operacje CRUD stworzony z wykorzystaniem technologii C#/.NET

Omawiany interfejs programowania aplikacji zaimplementowany został w języku C#, z zastosowaniem środowiska uruchomieniowego .NET w wersji piątej. Usługa sieciowa, o której mowa w niniejszym podrozdziale, stanowi złożenie dwóch składowych. Pierwszą z nich jest program, którego zadaniem jest dostarczenie definicji oraz obsługa określonych poleceń klienta, wydanych w postaci żądań protokołu hipertekstowego. Ścisłe rzecz ujmując, to właśnie ten program należy określić terminem interfejsu programowania aplikacji. Jako drugą składową natomiast, wskazać należy serwer warstwy aplikacji, pozwalający na obsługę komunikacji pomiędzy usługą sieciową a urządzeniami z zewnątrz. Serwerem wykorzystanym w ramach zaimplementowanej usługi sieciowej jest otwartoźródłowe oprogramowanie Kestrel.

W celu powiązania metod operujących na wewnętrznym modelu danych z fizycznymi strukturami bazodanowymi wykorzystano maper obiektowo-relacyjny Entity Framework Core w wersji drugiej dla relacyjnych systemów baz danych, a także narzędzie Mongoose dla nierelacyjnej bazy danych MongoDB.

Utworzone w języku C# rozwiązanie złożone jest z pięci projektów powiązanych pomiędzy sobą zależnościami. Pierwszy z projektów zawiera klasy kontrolerów, dokumenty json globalnych właściwości usługi w określonych środowiskach, punkt startowy programu, a także klasę konfiguracji dla wszystkich definiowanych w obrębie całego rozwiązania usług. W projekcie tym, można uzyskać dostęp do struktur programistycznych, definiowanych we wszystkich pozostałych fragmentach rozwiązania.

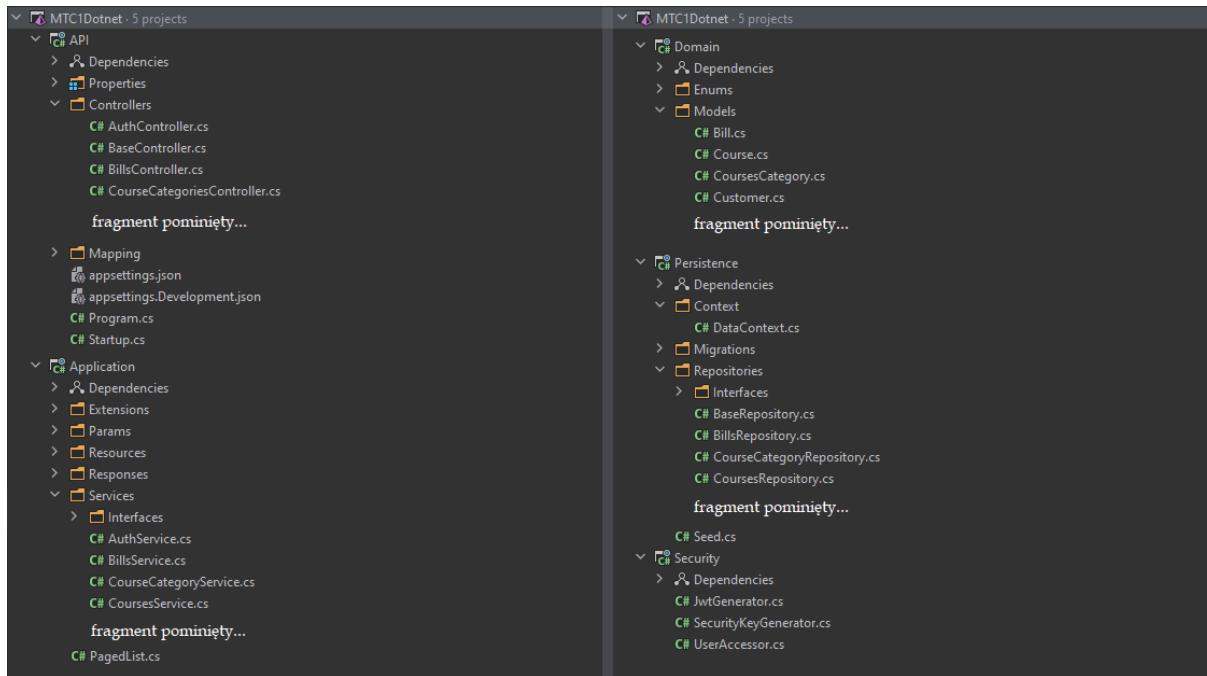
Drugi z projektów, stworzony został z myślą przechowywania kodu źródłowego związanego z przetwarzaniem logiki biznesowej programu. Zdefiniowane zostały tu klasy serwisów, metody rozszerzeń dla typów języka, obiekt parametrów paginacji, klasy transferu danych (*ang. Data Transfer Objects*), a także prodecury generowania opowiedzi dla użytkownika. Od projektu tego, zależne są wszystkie fragmenty rozwiązania z wyłączeniem pierwszego z omawianych.

Na równorzędnym poziomie hierarchii, zdefiniowany został projekt dotyczący obsługi operacji związanych z bezpieczeństwem interfejsu API. Wyróżnić możemy tutaj klasę generatora tokenu uwierzytelniającego, a także metodę dostępu do właściwości identyfikujących użytkownika poprzez zawartość kontekstu żądania http.

Kolejnym fragmentem rozwiązania stworzonego w języku C# jest projekt dostępu do danych. W projekcie tym, wskazać należy klasę kontekstu danych, zbiór migracji modelu danych do schematów bazodanowych, klasę propagacji danych początkowych, a także listę metod struktur repozytoriów, które pozwalają na wykonywanie bezpośrednich operacji na wykorzystywanym zbiorze informacji. Omawiany fragment, posiada zależność względem projektu logiki biznesowej.

Ostatnim projektem rozwiązania jest przestrzeń przechowywania klas modelu danych. Fragment ten, nie wprowadza zależności względem jakiegokolwiek z pozostałych projektów i pełni rolę rdzenia aplikacji. Każda z omówionych powyżej przestrzeni kodu źródłowego posiada dostęp do elementów modelu, jednakże jakiekolwiek operacje na tych danych, wykonywane są tylko i wyłącznie z poziomie klas repozytoriów.

Na ilustracji 4.6 przedstawiono strukturę obiektów wewnętrz poszczególnych fragmentów rozwiązania. Niektóre spośród elementów każdego projektu zostały pominięte na niniejszym rysunku w celu zachowania czytelności omawianej treści.



Rys. 4.6: Struktura obiektów wewnętrz rozwiązań systemu internetowego zaimplementowanego w języku C#

Interfejs API realizujący operacje CRUD stworzony z wykorzystaniem technologii NodeJS/Express

Druga z реализациj interfejsu programowania aplikacji napisana została w języku JavaScript, zgodnym ze specyfikacją językową ECMAScript 6. Do implementacji funkcjonalności omawianej usługi sieciowej, wykorzystano bibliotekę ExpressJS w wersji czwartej, a całość oprogramowania uruchomiono na serwerze wykorzystującym natywne moduły platformy uruchomieniowej NodeJS.

W celu powiązania metod operujących na wewnętrznym modelu danych z fizycznymi strukturami bazodanowymi wykorzystano maper obiektowo-relacyjny Prisma w wersji pierwszej dla relacyjnych systemów baz danych, a także narzędzie Mongoose dla nierelacyjnej bazy danych MongoDB.

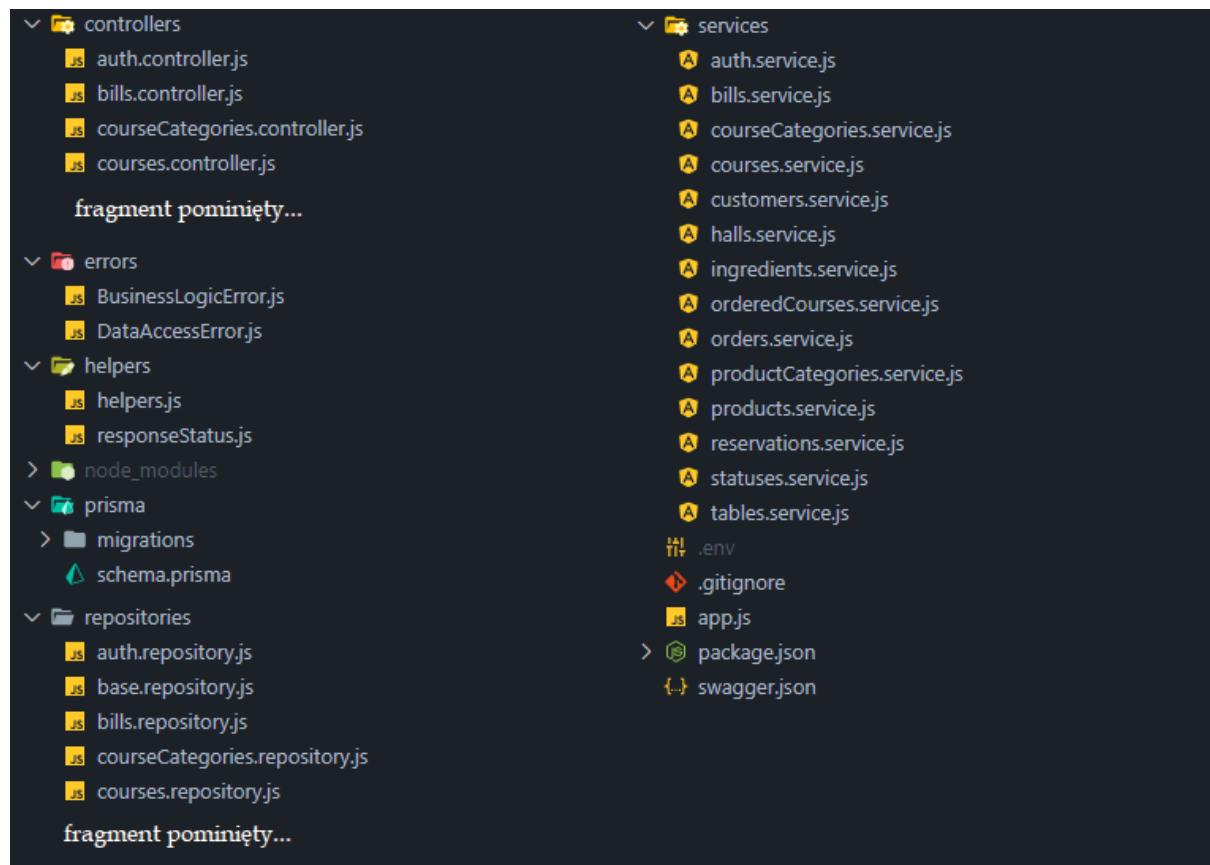
W przeciwieństwie do rozwiązania utworzonego przy wykorzystaniu technologii firmy Microsoft, schemat struktur programistycznych interfejsu programowania aplikacji opartego o JavaScript cechuje się znaczco niższym poziomem skomplikowania. Całość oprogramowania przechowywana jest w ramach pojedynczego projektu, wewnątrz którego podział struktur programistycznych ze względu na odpowiedzialności dokonany został na poziomie folderów.

Punktem startowym aplikacji, a także miejscem definicji podstawowej konfiguracji elementów usługi sieciowej jest skryptowy plik o nazwie *app.js*. W pliku tym, określono wartości dla zmiennych globalnych dotyczących ścieżki podstawowej w aplikacji, czy też klucza tajnego wykorzystywanego do wyliczania tokenu uwierzytelniania. Ponadto, określono funkcje pośredniczące wykonujące się przed rozpoczęciem zadanej funkcji modułu kontrolera. Plik *app.js*, zawiera także informacje dotyczące lokalizacji modułów, odpowiadających za obsługę punktów końcowych dla określonej ścieżki wywołania.

Poza plikiem startowym interfejsu programowania aplikacji wskazać należy foldery, przechowujące moduły odpowiedzialne za poprawną pracę całości aplikacji. Pierwszy z folderów o nazwie *controllers*, stanowi zbiór modułów zawierających funkcje obsługujące każdy z zaimplementowanych punktów końcowych. Funkcje te, odwołują się do fragmentów oprogramowania

realizujących operacje logiki biznesowej, które to fragmenty umieszczone są w folderze *services*. Analogiczne odwołanie ma miejsce w kontekście modułów serwisów a funkcji operujących na danych. Funkcje te, znaleźć można w folderze *repositories*. Ponadto, wyróżnić należy katalog *errors*, przechowujący kod źródłowy dotyczący przetwarzania błędów dla każdej z logicznych warstw interfejsu, katalog *helpers* w ramach którego zdefiniowane zostały funkcje pomocnicze, a także folder *prisma*, zawierający w sobie pliki migracji oraz dokument definicji schematu modelu danych.

Na ilustracji 4.7 przedstawiono strukturę obiektów wewnętrz poszczególnych fragmentów rozwiązania. Niektóre spośród elementów każdego projektu zostały pominięte na niniejszym rysunku w celu zachowania czytelności omawianej treści.



Rys. 4.7: Struktura obiektów wewnętrz rozwiązania systemu internetowego zaimplementowanego w języku JavaScript

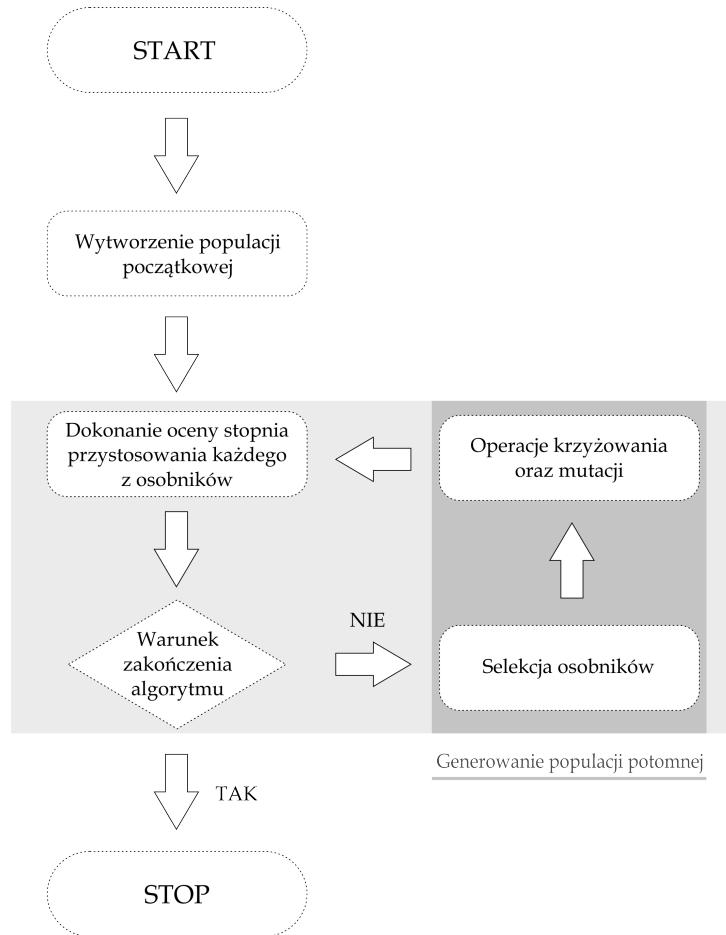
Algorytm metaheurystyczny dla symetrycznego problemu komiwojażera

W celu realizacji badania wydajności przetwarzania operacji współbieżnych zdecydowano się na implementację algorytmu metaheurystycznego przeznaczonego do rozwiązywania symetrycznego problemu komiwojażera. Zaimplementowany algorytm, sklasyfikować należy jako heurystykę z rodziny algorytmów genetycznych. Algorytmy te, zaliczane są do przestrzeni schematów ewolucyjnych, których główną koncepcją jest próba przeniesienia procesów, a także zachowania biologicznych obserwowanych w ramach dziedziny genetyki, w obszar definiowania oraz realizacji algorytmów.

Należyte przedstawienie procesu wykonania algorytmu genetycznego wymaga wprowadzenia odpowiedniej terminologii. Terminologia ta, podobnie jak wspomniane uprzednio obserwowane procesy biologiczne, została zaczerpnięta z dziedziny genetyki. Podstawowym pojęciem,

który należy wyróżnić w ramach wprowadzanych definicji jest chromosom. W kontekście algorytmu genetycznego jest on interpretowany jako pojedynczy element rozwiązania określonego problemu optymalizacyjnego (w tym przypadku, pojedyncza lokalizacja która ma zostać odwiezione przez osobę przemierzającą trasę). Wszystkie spośród permutacji zbioru chromosomów, przy założeniu, że liczba elementów permutacji równa jest wielkości zbioru, identyfikowane są jako osobniki, w ramach określonej populacji. Każdy z osobników populacji może zostać poddawany modyfikacji jego kodu genetycznego (zmiany jego chromosomów), poprzez zdefiniowane operacje krzyżowania osobników, a także ich mutacji. Ponadto, w celu wyboru najlepszych spośród osobników, wykonywany jest proces selekcji. Proces ten, powiązany jest ściśle z metryką oceny jakości osobnika względem populacji nazywaną funkcją przystosowania.

Na rysunku 4.8 zilustrowano schemat blokowy algorytmu genetycznego.



Rys. 4.8: Schemat blokowy algorytmu genetycznego

Implementowane rozwiązanie omawiane w ramach niniejszej pracy dyplomowej cechuje się realizacją funkcji krzyżowania osobników z wykorzystaniem dwóch punktów podziału (tzw. krzyżowanie dwupunktowe), a także dokonaniem mutacji w zależności od przyjętego prawdopodobieństwa. Ponadto, metodę selekcji oparto na technice ruletkowej, wzbogacając technikę tą, o mechanizm zapobiegania zjawisku przedwczesnej zbieżności poprzez kontrolę liczby osobników dominujących w populacji.

W tabeli 4.1 wymieniono hiperparametry zaimplementowanej metaheurystyki genetycznej, a także opisano ich znaczenie w kontekście schematu działania algorytmu.

Niezależnie od środowiska, w którym implementowany został opisany powyżej algorytm, sposób uruchamiania, przekazywania danych wejściowych, a także format danych wyjściowych jest identyczny. Omawiany algorytm metaheurystyczny dostępny jest z poziomu punktu końco-

Tab. 4.1: Wykaz hiperparametrów zaimplementowanego algorytmu genetycznego

Nazwa parametru	Opis	Wartość domyślna
Prawdopodobieństwo mutacji osobnika	Parametr definiujący częstość wystąpienia zdarzenia modyfikacji pojedynczego chromosomu w obrębie osobnika	0,01
Prawdopodobieństwo krzyżowania osobników	Parametr definiujący częstość wystąpienia zdarzenia wymiany kodu genetycznego pomiędzy określonymi osobnikami	1
Rozmiar populacji	Liczba osobników poddawanych ewolucji, spośród których ostatecznie wybierane jest rozwiązanie najlepsze	0,75 * liczba wszystkich rozpatrywanych lokalizacji
Liczba osobników dominujących w ramach pojedynczej populacji	Rozmiar podgrupy w obrębie pojedynczej populacji, skupiającej w sobie osobników o najwyższych wartościach funkcji przystosowania	0.25 * rozmiar populacji
Czas wykonania algorytmu	Przedział czasu określany przez liczbę sekund, w ramach którego wykonywana będzie główna pętla algorytmu	60

wego określonych interfejsów programowania aplikacji jako żądanie protokołu hipertekstowego dla ścieżki `/api/algorithms/roadPlan` oraz metody HTTP POST. Format danych wejściowych określony został jako tablica obiektów notacji JSON, zawierających właściwości definiujące szerokość i długość geograficzną, a także adres dla pojedynczej lokalizacji. Rezultatem otrzymywanym w ramach odpowiedzi protokołu hipertekstowego jest obiekt zawierający informacje dotyczącą ilorazu uzyskanego wyniku względem wyniku optymalnego, a także liczbę przeprowadzonych w pętli głównej algorytmu. Czas wykonania pojedynczego żądania dla omawianego punktu końcowego wynosi co najmniej 60 sekund.

W odniesieniu do szczegółów implementacyjnych tyczących się języka C# oraz środowiska uruchomieniowego .NET, wspomnieć należy o wykonaniu kodu źródłowego głównej pętli algorytmu w ramach osobnego wątku. Po zdefiniowaniu wartości hiperparametrów heurystyki, inicjowana jest nowa instancja klasy `Thread`, która przyjmuje w ramach konstruktora metodę wykonawczą (w tym przypadku metodę pętli głównej algorytmu). Następnie, obiekt klasy `Timer`, odlicza czas 60 sekund, w ramach których pętla programu może być wykonana. Po upłynięciu czasu wykonania kodu, wątek algorytmiczny łączony jest z głównym wątkiem wykorzystywanym do przetwarzania żądania, a rezultaty działania algorytmu formułowane są do postaci ciała odpowiedzi protokołu HTTP.

W ramach interfejsu programowania aplikacji zaimplementowanego w języku JavaScript, sposób przetwarzania współbieżnego różni się w sposób znaczący względem języka C#. Język JavaScript jest rozwiązaniem które dostarcza mechanizmy przetwarzania tylko i wyłącznie w obrębie pojedynczego wątku procesora. Dlatego też, wspominając o tej właśnie technologii programistycznej nie można stwierdzić, że istnieje możliwość współbieżnego wykonania kodu źródłowego. Przepływ sterowania od momentu wywołania punktu końcowego uwzględnia inicjalizację serwisu realizacji algorytmów, zdefiniowanie hiperparametrów programu, rozpoczęcie wykonania pomiaru czasu, a także uruchomienie pętli głównej algorytmu. Pomiar czasu wykonywany jest za pomocą natywnej struktury programistycznej dostarczanej zarówno w ra-

mach środowiska NodeJS, jaki i implementowanej przez silniki obsługi języka JavaScript w przeglądarkach internetowych (tj. interfejs *Performance*).

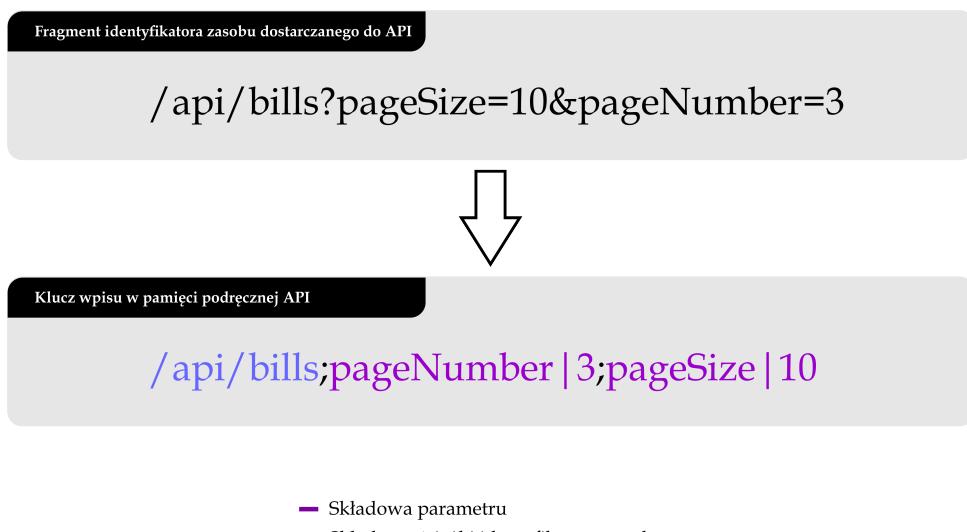
Mechanizmy obsługi pamięci podręcznej

Implementacja mechanizmów obsługi pamięci podręcznej wykonana została w oparciu o otwartoźródłowe rozwiązanie *Redis*, dostarczające zarówno strukturę danych pełniącą rolę magazynu wpisów pamięci cache, jak i interfejsy programistyczne służące do ustanowienia połączenia, a także zarządzania zawartością tej struktury. Magazyn pamięci podręcznej Redis uruchomiony został jako autonomiczna usługa sieciowa poprzez wykonanie obrazu *bitnami/redis* w obrębie kontenera *Docker*.

Niezależnie od badanej technologii programistycznej, zaimplementowane mechanizmy pamięci podręcznej są niezależne od określonego punktu końcowego i mogą zostać połączone z dowolną metodą klasy kontrolera, której wynikiem działania jest obiekt odpowiedzi, specyficzny względem wybranego języka. Wykonanie kodu źródłowego związanego z obsługą mechanizmów pamięci podręcznej odbywa się wewnątrz metod pośredniczących (ang. *Middleware*). Metody te, uzyskują dostęp zarówno do treści żądania wygenerowanego przez aplikację kliencką, jak i odpowiedzi będącej rezultatem działania metody klasy kontrolera. Implikuje to możliwość definiowania identyfikatorów wpisów w pamięci podręcznej dla każdego z wywołanych żądań.

Klucze poszczególnych elementów przechowywanych w pamięci cache składają się ze składowej ścieżki, oraz składowych parametrów. Każdy z fragmentów identyfikatora wpisu rozdzielony jest znakiem średnika, natomiast w obrębie składowej parametru klucz odseparowany jest od wartości znakiem kreski pionowej. Aby zapewnić jednoznaczność wpisu niezależnie od kolejności w jakiej aplikacja kliencka definiuje parametry wewnętrz ciała żądania, poszczególne składowe właściwości są sortowane w kolejności rosnącej według nazwy parametru.

Na ilustracji 4.9 zaprezentowano format klucza wpisu przechowywanego w pamięci podręcznej.



Rys. 4.9: Format klucza wpisu przechowywanego w pamięci podręcznej

Wdrożone zostały dwa rodzaje mechanizmów obsługi pamięci podręcznej. Pierwszy z nich, zakłada stały czas przechowywania wpisu w magazynie, a także unieważnienie elementu w momencie wywołania metody aktualizacji danych powiązanych z encją opisującą określony ele-

ment. Mechanizm ten, stanowi klasyczną implementację pamięci podręcznej stosowaną w większości produkcyjnych interfejsów programowania aplikacji.

Drugi mechanizm natomiast, stanowi autorskie podejście twórcy pracy do zarządzania wpisami pamięci podręcznej poprzez ustalenie zmiennego czasu przechowywania elementu w zależności od częstości wywołania punktu końcowego, a także liczby unieważnień. W podejściu tym, wyróżnić należy niezależną strukturę danych, przechowującą liczbę wykonień dla określonego endpointu, a także liczbę wykonień tych punktów końcowych, których działanie prowadzi do unieważnienia wpisu. Przedstawiona struktura jest aktualizowana przy każdym żądaniu wygenerowanym przez klienta i wraz z postępującym działaniem interfejsu programowania aplikacji zawiera coraz bardziej dokładną charakterystykę wywołań poszczególnych obszarów API. Dane te, przechowywane są wewnątrz pamięci programu interfejsu programowania aplikacji, natomiast autor zostawia dowolność co do ich synchronizacji z zewnętrznym źródłem. Synchronizacja ta może być wykonywana poprzez uruchamianie skryptów cyklicznych zadań (np. *cron*), bądź też w momencie uruchamiania oraz wyłączania API.

W momencie tuż przed stworzeniem wpisu i wysyłaniem go do magazynu pamięci podręcznej, w omawianym podejściu następuje kalkulacja czasu życia generowanego elementu. Na początku, obliczany zostaje współczynnik wykonania będący ilorazem liczby uruchomień rozpatrywanego punktu końcowego oraz maksymalnej liczby uruchomień dowolnego z endpointów. Kolejno, w sposób analogiczny tworzony zostaje współczynnik inwalidacji. W tym przypadku stanowi on iloraz wywołań punktów końcowych unieważniających rozpatrywane żądanie i maksymalnej liczby unieważnień dla dowolnego żądania. Oba współczynniki mogą przyjmować wartości z zakresu od zera do jeden. Czas ważności wpisu pamięci podręcznej stanowi iloczyn maksymalnej wartości czasu życia elementu oraz sumarycznego współczynnika wykonania i inwalidacji. Współczynnik ten, określany jest jako suma połowy wartości współczynnika wykonania, a także połowy odwrotności wartości współczynnika unieważnienia. Wprowadzenie takiej formuły prowadzi do uzyskiwania dłuższych czasów przechowywania wpisów dla żądań często wykonywanych oraz rzadko unieważnianych. Wraz ze spadkiem liczby wywołań punktu końcowego, bądź wzrostem liczby jego unieważnień czas ważności wpisu pamięci podręcznej jest zmniejszany. Tak zdefiniowana koncepcja prowadzić powinna do faworyzowania tych punktów końcowych, z których użytkownicy interfejsów korzystają najczęściej.

$$\text{exec_factor}(x) = \frac{\text{executions}(x)}{\max(\text{executions}(x) : x = 1..n)} \quad (4.1)$$

$$\text{invalid_factor}(x) = \frac{\text{invalidations}(x)}{\max(\text{invalidations}(x) : x = 1..n)} \quad (4.2)$$

$$\text{ttl}(x) = (\frac{1}{2} * \text{exec_factor}(x) + \frac{1}{2} * (1 - \text{invalid_factor}(x))) * \text{max_ttl} \quad (4.3)$$

gdzie:

$\text{executions}(x)$ - liczba wywołań punktu końcowego, którego wpis w pamięci podręcznej identyfikowany jest kluczem x

$\text{invalidations}(x)$ - liczba unieważnień wpisu pamięci podręcznej identyfikowanego kluczem x

$\text{execution_factor}(x)$ - współczynnik częstości wykonania określonego punktu końcowego

$invalidation_factor(x)$ - współczynnik częstości unieważnień wpisów dotyczących określonego punktu końcowego

$ttl(x)$ - czas ważności wpisu dodawanego do pamięci podręcznej

$max_ttl(x)$ - maksymalny czas ważności wpisu podręcznej względem którego wyliczany jest faktyczny czas ważności

Kod źródłowy służący do zarządzania elementami pamięci podręcznej zaimplementowany został w API C# za pomocą funkcji pośredniczących dołączanych do struktur programistycznych w formie atrybutu. Stworzona została klasa dziedzicząca po natywnym typie *Attribute*, a także implementująca interfejs *IAsyncActionFilter*. Implementacja niniejszego interfejsu wymaga zdefiniowania metody *OnActionExecutionAsync*, w ramach której napisano logikę działania obu mechanizmów obsługi cache. Wspomniana metoda, posiada dostęp do kontekstu żądania HTTP przez co możliwe było zarówno wykonanie operacji, tuż przed jak i tuż po uruchomieniu metody klasy konstruktora.

W przypadku platformy uruchomieniowej NodeJS oraz języka JavaScript, wymagane było utworzenie dwóch osobnych funkcji, a także rozdzielenie logiki zarządzania pamięcią podręczną, względem operacji wykonywanych przed oraz po uruchomieniu kontrolera. Interfejs programowania aplikacji tworzony z wykorzystaniem omawianych technologii, a także biblioteki ExpressJS, opiera swoje działanie na ciągu kolejno wywoływanych metod, z których ostatnią jest ta, która wysyła odpowiedź do klienta, korzystając z natywnego obiektu odpowiedzi. Taka zasada działania, wymusiła modyfikację kodu źródłowego metod klas kontrolerów w taki sposób, aby rezultat ich wykonania nie zwracał bezpośrednio odpowiedzi w stronę klienta, a przekazywał ją do kolejnej funkcji pośredniczącej. Stwierdzenie to jednak nie jest sprzeczne z informacją przytoczoną na początku tej sekcji. Chociaż zmieniony musiał zostać fragment kodu źródłowego metody kontrolera, to nie zmienia to faktu niezależności i odrębności logicznych operacji przeprowadzanych przez tę metodę.

Uogólniony przepływ sterowania wewnętrz interfejsów API implementujących pamięć podręczną, w kontekście odczytu danych, a także abstrahując od używanej technologii programistycznej, stanowi następującą sekwencję:

- Wywołania żądania protokołu HTTP przez aplikację kliencką
- Określenie metody klasy konstruktora obsługującej żądanie o wyspecyfikowanych składowych
- Wywołanie metody obsługi wpisu pamięci podręcznej dla operacji odczytu (tj. połączenie się z magazynem cache, sprawdzenie dostępności wpisu)
- Wywołanie metody klasy konstruktora w przypadku braku wpisu w magazynie cache
- Pominiecie metody klasy konstruktora w przypadku uzyskania wpisu z magazynu
- Zwrócenie odpowiedzi z jednego z dwóch źródeł danych

W aspekcie zapisu danych natomiast wyróżnić możemy poniższe kroki:

- Wywołania żądania protokołu HTTP przez aplikację kliencką
- Określenie metody klasy konstruktora obsługującej żądanie o wyspecyfikowanych składowych
- Wywołanie metody klasy konstruktora
- Ustalenie kolekcji kluczy wszystkich wpisów pamięci podręcznej powiązanych z modyfikowaną encją
- Unieważnienie wszystkich wpisów na podstawie kluczy uzyskanej kolekcji

- Zwrócenie odpowiedzi w kierunku aplikacji klienckiej

Obie powyższe sekwencje, w przypadku metody autorskiej wzbogacone zostały o aktualizację informacji o wykonaniu i unieważnieniu punktu końcowego, a także proces wyliczenia współczynników wykonania i unieważnienia.

Implementacja wzorca projektowego CQRS z uwzględnieniem replikacji pomiędzy źródłami danych

Implementacja wzorca projektowego podziału odpowiedzialności uwzględnia separację obsługi żądań dotyczących pozyskiwania danych (zwanych zapytaniami), a także żądań manipulacji danymi (zwanych komendami). W kontekście przeprowadzonego badania, niezależnie od omawianej technologii, wewnątrz każdego z interfejsów programowania aplikacji zdefiniowano dwa osobne zbiory encji pełniące role modeli danych. Każdy z modeli bazodanowych, odwołuje się do innego źródła danych, przez co wyróżnić możemy dwa zupełnie odseparowane środowiska zapisu oraz odczytu informacji.

Aby zachować zgodność danych pomiędzy modelami, a co za tym idzie, strukturami bazodanowymi które są odwzorowywane na podstawie tych modeli, skonfigurowano mechanizm replikacji transakcyjnej. W mechanizmie tym, wyróżnić możemy pojęcia strony publikującej (*ang. Publisher*), dystrybutora transakcji (*ang. Distributor*), a także subskrybenta (*ang. Subscriber*). Każde z tych pojęć tyczy się określonego systemu bazodanowego i określa jego rolę w procesie replikacji. Jeżeli replikacja wykonywana jest pomiędzy dwoma źródłami bazodanowymi (tj. tak jak w niniejszym przypadku), rolą dystrybutora przypisana może być do dowolnego z systemów bazodanowych i determinuje ona sposób generowania komunikatów replikacji. W kontekście omawianego rozwiązania, stroną publikującą i jednocześnie dystrybutorem był serwer bazodanowy wykorzystywany do operacji zapisu. W momencie wykonania zapytania modyfikującego dane w obrębie replikowanego systemu bazodanowego, zapytanie to jest wykonywane lokalnie, a następnie dokonane zmiany są transferowane do bazy danych subskrybenta (w tym przypadku bazy danych obsługującej operacje odczytu). Wprowadzenie takiego mechanizmu pozwoliło na dokonanie całkowitej separacji źródła danych przeznaczonego do zapisu danych, od tego, desygnowanego do ich odczytu.

Na ilustracji 4.10 zobrazowano zasadę działania zaimplementowanego mechanizmu replikacji transakcyjnej.

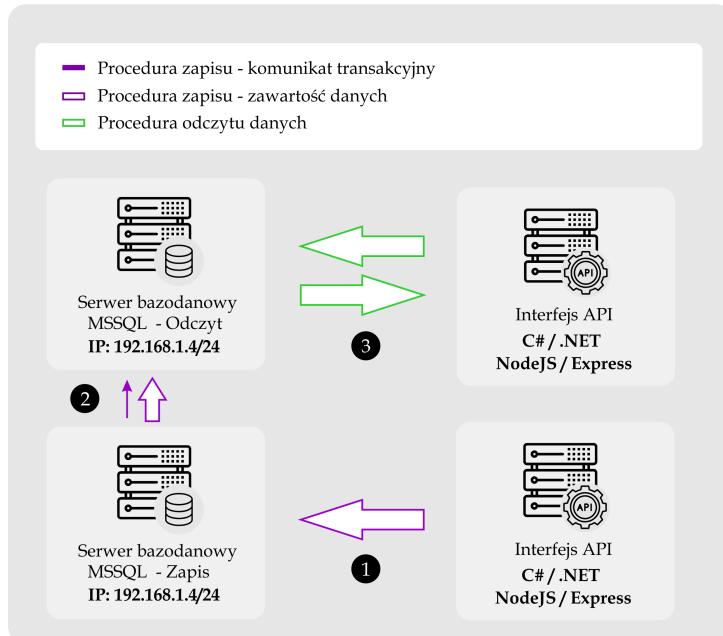
Zastosowanie omówionego mechanizmu pozwoliło na dokonanie implementacji interfejsów programowania aplikacji, które mogą zarówno uwzględniać dwa odrębne, zoptymalizowane względem typu operacji modele danych, jak i manipulować danymi bez ryzyka wystąpienia nadmiernego opóźnienia wynikającego z konieczności wykonania synchronizacji na poziomie API.

Podstawową intencją dokonania separacji środowisk zapisu i odczytu, a także wykorzystania wzorca projektowego podziału odpowiedzialności jest możliwość wdrożenia optymalizacji modelu odczytu danych. Optymalizacje te, zarówno dotyczące struktur bazodanowych, jak i rozwiązań programistycznych, mają na celu przyspieszenie procedury odwoływanego się, pozyskiwania oraz zwracania danych bezpośrednio do aplikacji klienckiej.

Zależnie od wykorzystywanej technologii, możliwe było wprowadzenie różnych rozwiązań optymalizacyjnych dotyczących zarówno struktur bazodanowych, jak i generowania zapytań w kierunku bazy danych.

W ramach interfejsu programowania aplikacji uruchamianego w środowisku .NET/C# wprowadzono następujące usprawnienia wydajnościowe:

- Kompilowane kwerendy (*ang. Compiled Queries*) - zapytania o dane linq, wykorzystywane przez mapper entity framework core stanowią intuicyjny i prosty w budowie interfejs komuni-



Rys. 4.10: Schemat wykonywania procedur zapisu oraz odczytu z wykorzystaniem replikacji transakcyjnej

kacji z systemem bazodanowym. Programista, w nieskomplikowany sposób może zbudować zaawansowaną kwerendę pozyskującą dane, odwołującą się jednocześnie do wielu struktur bazodanowych. Rozwiążanie to jednak, wymaga zbudowania przez bibliotekę entity framework core drzewa składniowego reprezentującego kwerendę i pozwalającego na jej przekształcenie do kodu języka bazy danych. Proces ten, wraz ze zwiększeniem się poziomu skomplikowania zapytania może trwać znaczowo dłużej i wpływać negatywnie na wydajność aplikacji. Dlatego też, zastosowano się na wykorzystanie podejścia kompilowanych kwerend, które pozwala na zbudowanie drzew składniowych dla zapytań jednokrotnie po uruchomieniu api, a następnie szybsze generowanie kodu języka bazy danych.

- Pula kontekstów bazodanowych (*ang. DbContext Pool*) - dla standardowej konfiguracji interfejsów API w ramach platformy .NET, instancja klasy kontekstu bazodanowego inicjalizowana każdorazowo dla pojedynczego żądania wysyłanego w kierunku API. Rozwiążanie to jest akceptowalne przy standardowym natężeniu otrzymywanych żądań ze względu na niewielki rozmiar instancji klasy, a także szybki czas jej ustanowienia. W przypadku zwiększonego natężenia ruchu sieciowego w kierunku API, wydajność może zostać z tego powodu obniżona. W celu zapobiegania spadku wydajności zastosowano mechanizm puli obiektów kontekstu bazodanowego. Mechanizm ten pozwala na utrzymywanie wielu obiektów kontekstu przez cały czas działania API i czasowe przydzielanie tych obiektów do obsługi nadchodzących żądań.
- Leniwe ładowanie (*ang. Lazy loading*) - domyślnym zachowaniem obiektu kontekstu bazodanowego jest wykonanie kwerendy tuż po jej deklaracji, a także pozyskanie wszelkich obiektów nawigacyjnych, które zostały w niej wymienione. Zachowanie to, powoduje pozyskanie określonych fragmentów danych, które zostaną wykorzystane dopiero w dalszej części przetwarzania programu. Mechanizm leniwego ładowania pozwala na pozyskiwanie danych dopiero wtedy, gdy instrukcje kodu źródłowego wskazują na konieczność ich wykorzystania.
- Rozdzielone zapytania (*ang. Split Queries*) - kod języka bazy danych, wygenerowany poprzez przetworzenie drzewa składniowego zapytania linq przez entity framework core, domyślnie uwzględnia pozyskiwanie encji zależnych (tj. encji identyfikowanych kluczem obcym) poprzez dokonanie lewostronnego iloczynu kartezjańskiego. Dzięki temu, niezależnie od tego,

ile encji zależnych programista będzie chciał uzyskać, operacja pozyskiwania danych spro-wadzona zostanie do pojedynczego zapytania. Wiąże się to jednak z możliwością uzyskania bardzo dużej liczby zduplikowanych danych. Dane te, są programowo filtrowane przez mapper obiektowo-relacyjny przed zwróceniem ich do kodu programu. Aby uniknąć konieczności redukcji zduplikowanych danych, przy założeniu wykonywania kwerend dołączających wiele encji zależnych wykorzystany został mechanizm rozdzielonych zapytań, który przekształca pojedyncze zapytanie z wielokrotnym lewostronnym iloczynem w serię kilku osobnych zapytań.

- Pula połączeń bazodanowych (*ang. Database Connections Pool*) - zdefiniowanie parametru obsługi wielu równolegle działających połączeń z serwerem bazodanowym.

Ponadto, wprowadzono wiele mniej znaczących usprawnień wydajnościowych, takich jak przekazywanie parametrów do zapytań LINQ zamiast stosowania stałych dosłownych, czy też redukcja wykorzystania zewnętrznych metod wewnętrz funkcji anonimowych zapytania.

W ramach interfejsu programowania aplikacji uruchamianego w środowisku NodeJS/Express wprowadzono następujące usprawnienia wydajnościowe:

- Wykorzystanie obiektu `include` do dołączania encji zależnych w sposób kontrolowany przez mapper obiektowo-relacyjny Prisma, tak aby zredukować możliwość wystąpienia problemu $n+1$ zapytań.
- Zdefiniowanie pojedynczej, globalnej instancji klienta mappera obiektowo-relacyjnego *PrismaClient*, w celu redukcji konieczności każdorazowego dokonywania operacji zajęcia i zwolnienia pamięci, a także czasu potrzebnego na nawiązanie połączenia pomiędzy klientem a instancją systemu bazodanowego.
- Pula połączeń bazodanowych - zdefiniowanie parametru obsługi wielu równolegle działających połączeń z serwerem bazodanowym.

Co więcej, wprowadzono również następujące optymalizacje wydajności ściśle dotyczące struktur bazodanowych:

- Zdefiniowanie indeksów oraz określenie pól jako unikalne w celu przyspieszenia procesu odwoływanego się do danych, a także ich filtracji oraz sortowania
- Ograniczenie przedziałów liczbowych dla pól numerycznych oraz maksymalnych długości dla ciągów tekstowych
- Przekształcenie określonych pól ciągów tekstowych z typu *NVARCHAR* wykorzystującego do kodowania pojedynczego znaku 2 bajty (kodowanie UTF16), do typu *VARCHAR* wykorzystującego do kodowania pojedynczego znaku jeden bajt.

4.3. Konfiguracja generycznych oraz dedykowanych platform chmurowych

W kontekście przeprowadzanych ewaluacji skonfigurowano trzy platformy chmurowe, w ramach których uruchomiono badane interfejsy programowania aplikacji. Wykorzystanie odmiennych platform chmurowych jako środowisk wdrożeniowych dla interfejsów API ma na celu dokonanie obserwacji zmiany wydajności działania systemów internetowych względem poziomu przystosowania usługi wdrożeniowej w odniesieniu do konkretnego interfejsu programowania aplikacji. Oznacza to, że ewaluowane zostały zarówno rozwiązania o wysokim stopniu ogólności (tj. dostarczające możliwości pełnego zarządzania systemem operacyjnym serwera, na którym wdrożone zostały aplikacje), jak i te, które bezpośrednio powiązane są z określoną technologią implementacyjną.

W ramach pierwszej z omawianych platform chmurowych, wykorzystano usługę typu *Infrastructure as a Service* w postaci wirtualnego serwera prywatnego udostępnionego przez do-

stawcę usług chmurowych *DigitalOcean*. Serwer ten, posiadał 25GB pamięci dyskowej, 16GB pamięci operacyjnej o dostępie swobodnym, a także ośmiordzeniową centralną jednostkę przetwarzania o architekturze 64-bitowej. Na omawianym serwerze zainstalowano dystrybucję systemu operacyjnego Linux o nazwie *Ubuntu Server* w wersji 22.04 LTS.

Niezależnie od technologii wdrażanego interfejsu programowania aplikacji, zdecydowano się na wykorzystanie tego samego serwera usługi WWW, którym w tym przypadku było oprogramowanie *Apache2*. Usługa ta, pełniła rolę odwróconego serwera pośredniczącego (ang. *Reverse-proxy*), pomiędzy serwerem interfejsu API a zlokalizowanymi na zewnątrz sieci urządzeniami klienckimi. Serwerem interfejsu API napisanego w języku C# było oprogramowanie *Kestrel*, natomiast w przypadku NodeJS API, usługą wykorzystywaną do hostowania aplikacji w środowisku lokalnym był framework *ExpressJS*. Oba, spośród wymienionych serwerów aplikacji uruchomione były w obrębie systemu operacyjnego w postaci programów typu demon działających w tle. Oprogramowanie poszczególnych interfejsów programowania aplikacji uruchomione było tylko na czas przeprowadzania ewaluacji i nigdy nie funkcjonowało w sposób równoległy.

Drugi z zastosowanych mechanizmów wdrożeniowych to rozwiązanie typu *Platform as a Service* udostępniane przez dostawcę usług chmurowych *Microsoft Azure*. W tym przypadku, programista zarządzać może uruchomionymi usługami, określić konfigurację każdej z nich, natomiast nie posiada on dostępu do warstw uruchamiania programów, czy też administrowania systemem operacyjnym. Aby zapobiec niereprezentatywności wyników badań przeprowadzanych w obrębie przedstawianego środowiska chmurowego, zdecydowano się na skorzystanie z opcji dynamicznego przydzielu zasobów sprzętowych do skonfigurowanych usług, z zastrzeżeniem maksymalnych wartości pojemności dyskowej do 25GB, pamięci RAM do 16GB, a także liczby wykorzystywanych rdzeni centralnej jednostki przetwarzania do ośmiu. Na omawianej platformie uruchomiono usługę interfejsu programowania aplikacji napisanego w języku C# z wykorzystaniem platformy .NET, a także system bazodanowy Microsoft SQL Server.

Odnosząc się do ostatniej spośród eksploatowanych platform chmurowych, zastosowano rozwiązanie typu *Platform as a Service* dostawcy usług *Heroku*. Rozwiązanie to, jest rekommendowanym sposobem wdrożenia aplikacji internetowych opartych o środowisko NodeJS. Analogicznie do platformy udostępnianej przez dostawcę DigitalOcean, w przypadku usługi Heroku istnieje możliwość doboru komponentów sprzętowych dostępnych w ramach fizycznej maszyny hostującej. W związku z tym faktem, a także w celu zapobiegania uprzywilejowaniu któregokolwiek z rozwiązań w kontekście przeprowadzanych badań, zastosowano te same wartości pojemności dyskowej, pojemności pamięci operacyjnej, a także liczby rdzeni procesora. Dostęp do systemu operacyjnego urządzenia serwerowego, zgodnie z typem przedstawianej usługi, nie jest przyznany użytkownikowi. Na omawianej platformie uruchomiono usługę interfejsu programowania aplikacji napisanego w języku JavaScript z wykorzystaniem platformy NodeJS/Express, a także nierelacyjny system bazodanowy MongoDB.

4.4. Konfiguracja narzędzia do realizacji badań

Narzędziem wykorzystanym do przeprowadzenia procesu ewaluacji wydajności interfejsów programowania aplikacji było otwartoźródłowe oprogramowanie Apache JMeter. Przytoczone rozwiązanie informatyczne dostarcza możliwości definiowania, konfiguracji, a także wykonywania testów wydajności usług sieciowych, z uwzględnieniem zmennego natężenia ruchu sieciowego, kontrolowanego przez użytkownika. Jedną z kluczowych zalet omawianego narzędzia jest możliwość kompozycji testów wydajnościowych o charakterze rozproszonym. Możliwość ta, została wykorzystana w ramach przeprowadzonych badań, dzięki czemu w czasie pojedynczej ewaluacji, żądania generowane były ze zmiennej liczby urządzeń klienckich.

Pierwszym etapem konfiguracji oprogramowania JMeter było przystosowanie wszystkich instancji tego narzędzia, ulokowanych w obrębie poszczególnych hostów sieci komputerowej. Przystosowanie to, sprowadzało się do edycji plików właściwości programu, a także skryptu programowania wsadowego uruchamiającego plik wykonywalny JMeter. Niezbędnym krokiem w kierunku ustanowienia komunikacji pomiędzy hostami było również wyłączenie systemowej usługi zapory sieciowej.

Implementacja ustawień programu JMeter dla komputera pełniącego rolę koordynatora testów sprowadzała się do modyfikacji zawartości pliku *jmeter.properties*. W pliku tym, wyszczególniono adresy internetowe hostów typu slave, przypisano numer portu dla serwera zdalnego wywoływania metod, wskazano lokalizację pliku przechowującego klucz szyfrowania komunikacji, a także ustalono opcję wykorzystania protokołu SSL do komunikacji za pomocą usługi RMI. Pozostała zawartość pliku nie uległa zmianie. Na listingu 4.1 pokazano fragment omaranego pliku właściwości zawierający zmodyfikowane wpisy.

Listing 4.1: Modyfikowane właściwości pliku konfiguracyjnego JMeter

```
remote_hosts=192.168.1.101, 192.168.1.102
server.rmi.port=1099
server.rmi keystore.type=JKS
server.rmi keystore.file=ssl_master_thesis_testing.jks
server.rmi keystore.password=master_thesis
server.rmi.ssl.disable=false
```

W analogiczny sposób zmodyfikowano zawartość plików konfiguracyjnych dla każdej z maszyn pełniących role generatorów żądań. W tym przypadku jednak, klucz *remote_hosts* zawierał tylko i wyłącznie adres IP hosta typu master, a plik klucza prywatnego, uprzednio wygenerowany na maszynie master, został rozdysytrybuowany pomiędzy maszynami slave. Po wykonaniu opisanych czynności, na każdym z hostów-generatorów uruchomiono plik wsadowy *jmeter-server*, nasłuchujący poleceń od urządzenia typu master.

Po realizacji czynności związanych z ustanowieniem transmisji pomiędzy maszynami, sporządzono plany testowe dotyczące poszczególnych scenariuszy badawczych. Wyróżnić należy dwa plany testowe, których określone warianty pozwoliły na przeprowadzenie każdej z ewaluacji określonych w ramach scenariuszy badawczych.

Plan testowy nr. 1

Pierwszy z planów testowych, stworzony został w celu obserwacji sposobu działania interfejsów programowania aplikacji względem zwiększającego się poziomu natężenia równolegle generowanego ruchu sieciowego. Podstawowym elementem schematu ewaluacyjnego jest grupa współbieżnie działających wątków (*ang. Concurrency Thread Group*). W przeciwieństwie do standardowej implementacji mechanizmu grupy wątków, dostępnej w ramach oprogramowania JMeter, grupa wątków współbieżnych pozwala na kumulowanie liczby aktywnych procesów w czasie testu, pozwalając na ich zakończenie dopiero w momencie określonym przez użytkownika. Moment ten, w kontekście niniejszych testów przypadał na chwilę zakończenia badania. Dzięki zastosowaniu takiego rozwiązania, możliwe było symulowanie zjawiska stopniowego wzrostu natężenia ruchu generowanego równolegle przez określoną liczbę użytkowników. Wybór standardowej grupy wątków, prowadziłby do finalnego uruchomienia określonej liczby procesów, jednakże nie pozwalałby na kontrolę momentu ich zakończenia, co prowadziłoby do zmiennego poziomu natężenia żądań, posiadającego charakterystykę losową, a także niepowiązanego z czasem trwania testu.

Kolejne składowe występujące w ramach planu testowego, podzielić należy na cztery grupy.

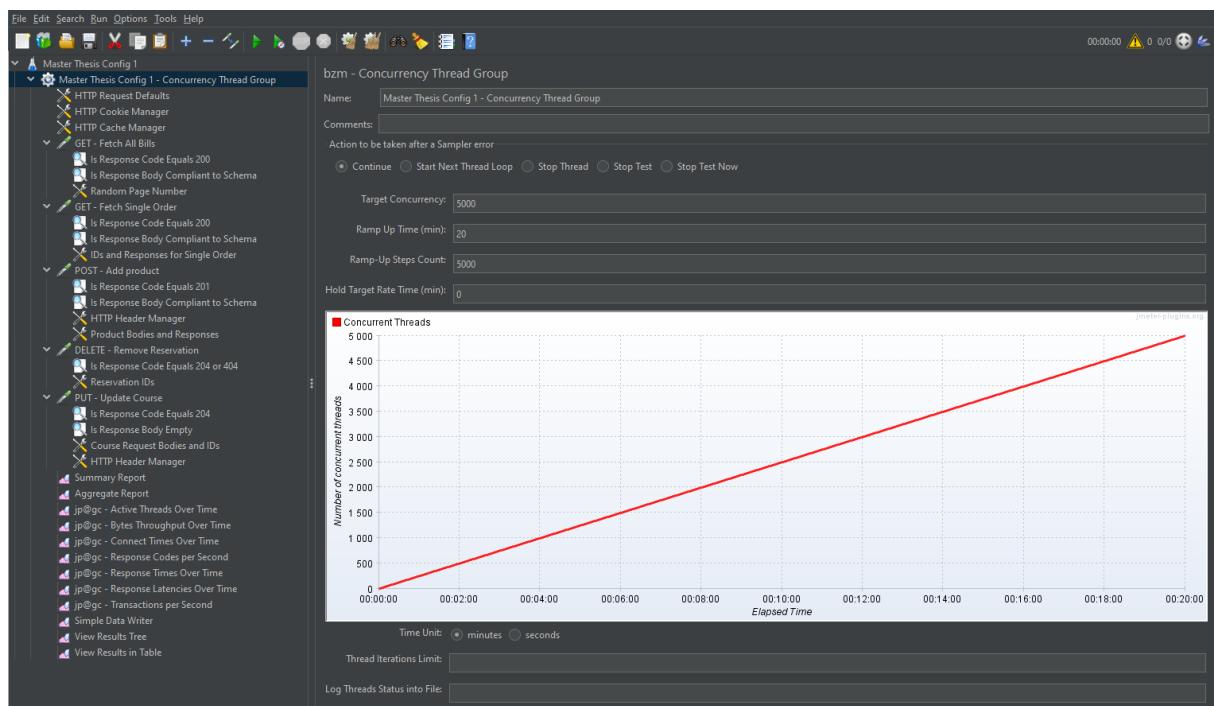
Pierwszą z nich są elementy konfiguracyjne (*ang. Config Elements*). W elementach tych, określono: adres IP interfejsu programowania aplikacji, port TCP, rodzaj protokołu hiperteksto-

wego, implementację klienta HTTP (wybrano domyślną - HttpClient4), a także czasy upływu limitu czasu połączenia (30000ms) oraz uzyskania odpowiedzi (120000ms).

Kolejna z grup składowych zawiera próbniki (*ang. Samplers*). Elementy te, dostarczają informacji o punktach końcowych API, do których wysyłane mają być żądania. Do każdego z próbników, można dodać indywidualny element konfiguracyjny, a także mechanizm weryfikacji poprawności realizacji żądania nazywany asercją (*ang. Assertions*). W przypadku niniejszego planu testowego, wprowadzono asercje dotyczące kodu odpowiedzi HTTP, a także jej schematu w postaci notacji JSON.

Ostatnie spośród składowych to nasłuchiwanie (*ang. Listeners*). Posiadają one postaci raportów, tabel, a także wizualizacji dotyczących przeprowadzonego testu.

Na ilustracji 4.11 przedstawiono zastosowane elementy składowe pierwszego planu testowego.

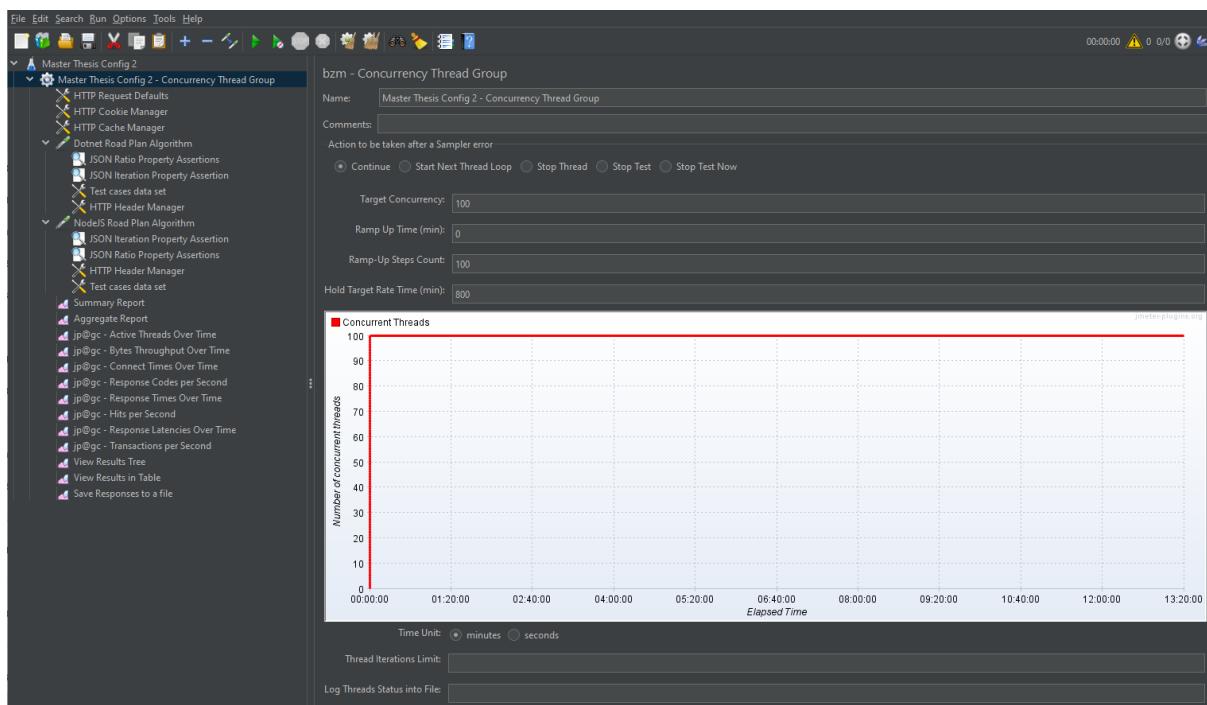


Rys. 4.11: Konfiguracja pierwsza planu testowego JMeter

Plan testowy nr. 2

Drugi z planów testowych, został przygotowany w kontekście wywoływania punktów końcowych o stałym czasie odpowiedzi, realizujących operacje charakteryzujące się wysokim kosztem obliczeniowym. W tym przypadku, zdefiniowano współbieżną grupę wątków o stałej liczebności równej 100. Czas podtrzymania każdego ze stu procesów określony został na 68000 sekund, co wynikało z założenia najbardziej pesymistycznego scenariusza badań, wykorzystujących ten właśnie plan. Ponadto, wprowadzono elementy konfiguracyjne dotyczące komunikacji z API z wykorzystaniem protokołu internetowego, a także, analogiczne względem pierwszego planu komponenty nasłuchujące. Spośród składowych próbników, wymienić należy dwa moduły dotyczące wywołania punktów końcowych interfejsów programowania aplikacji dla porównywanych technologii. Każdy z modułów zawierał asercje dotyczące występowania określonych właściwości w ramach odpowiedzi JSON, a także zbiory danych pomocniczych oraz element konfiguracyjny nagłówka żądania.

Na ilustracji 4.12 przedstawiono zastosowane elementy składowe drugiego planu testowego.



Rys. 4.12: Konfiguracja druga planu testowego JMeter

Plan testowy nr. 2 - wariant 1

Wariant pierwszy drugiego planu testowego wykorzystany został w kontekście obsługi punktów końcowych realizujących operacje asynchroniczne, poprzez odwołania do zewnętrznej usługi. Wprowadzone modyfikacje względem drugiego planu testowego obejmują zastosowanie innych elementów próbników (w tym przypadku elementów odwołujących się do punktów końcowych ewaluacji operacji asynchronicznych), a także wprowadzenie alternatywnej wartości czasu wykonania testu wynoszącej dziesięć minut

Plan testowy nr. 2 - wariant 2

Drugi z wariantów planu testowego dostosowano względem ewaluacji wydajności mechanizmów pamięci podręcznej. Liczba wątków w obrębie grupy ograniczona została do trzydziestu, a czas przeprowadzenia badania ustanowiono na 20 minut. Ponadto, wprowadzono komponenty próbników specyficzne w odniesieniu do badania mechanizmów cache. Należy także wspomnieć o wprowadzeniu próbnika pełniącego rolę mechanizmu unieważnienia wpisów pamięci podręcznej. Próbnik ten, wywoływany był tylko i wyłącznie w określonych momentach testu. Takie zachowanie, osiągnięto poprzez wprowadzenie elementu kontrolera przepływności (*ang. Throughput Controller*), determinującego liczbę wywołań próbnika, a także licznika czasu (*ang. Constant Timer*), wskazującego na moment wywołania.

Rozdział 5

Przeprowadzone badania

W niniejszym rozdziale przedstawiono oraz omówiono badania przeprowadzone w celu ewaluacji wydajności interfejsów API implementowanych z wykorzystaniem porównywanych technologii. Każde z wykonanych badań oparte jest o scenariusz testowy określony w ramach sekcji 3.2, a także dotyczy odmiennych aspektów działania usługi sieciowej interfejsu programowania aplikacji. W kolejnych sekcjach tego rozdziału w sposób szczegółowy opisano podjęte czynności badawcze, zwizualizowano rezultaty każdej z ewaluacji, dokonano analizy statystycznej, a także sformułowano wnioski.

5.1. Wpływ zastosowanego systemu bazodanowego na efektywność działania interfejsu API

W ramach badania zobserwowano zmianę wydajności działania interfejsów programowania aplikacji względem skomunikowanego z nim systemu bazodanowego. Wykorzystano cztery najpopularniejsze relacyjne systemy baz danych (tj. MySQL, SQL Sever, PostgreSQL oraz SQLite), a także jeden system nierelacyjny (tj. MongoDB). Interfejsy programowania aplikacji komunikujące się z poszczególnymi systemami bazodanowymi poddawane były coraz to większemu obciążeniu, poprzez zwiększenie liczby procesów generujących żądania.

Pierwszą czynnością wykonaną w ramach ewaluacji było spełnienie warunków początkowych dotyczących podjęcia czynności badawczych. Warunki te, uwzględniały realizację testów funkcjonalnych gwarantujących poprawność działania punktów końcowych interfejsu programowania aplikacji. Ponadto, przeprowadzenie tego rodzaju testów pozwoliło na określenie progiów tolerancji oraz frustracji wykorzystywanych w kontekście późniejszego wyliczania wartości wskaźnika wydajności aplikacji APDEX.

Ewaluacja funkcjonalna, polegała na uruchomieniu testu uwzględniającego 30 współbieżnie pracujących wątków oprogramowania JMeter, które wysyłyły żądania w kierunku punktów końcowych API w czasie 10 minut.

Poczynione zostały dwa następujące założenia:

- generowanie żądań przez 30 współbieżnych procesów jest interpretowane jako funkcjonowanie interfejsu programowania aplikacji w standardowych warunkach pracy
- ogólna ocena wydajności systemu, a także progi uwzględniane w ramach wskaźnika APDEX generowane są na podstawie tylko tych punktów końcowych, które poddawane są ewaluacji

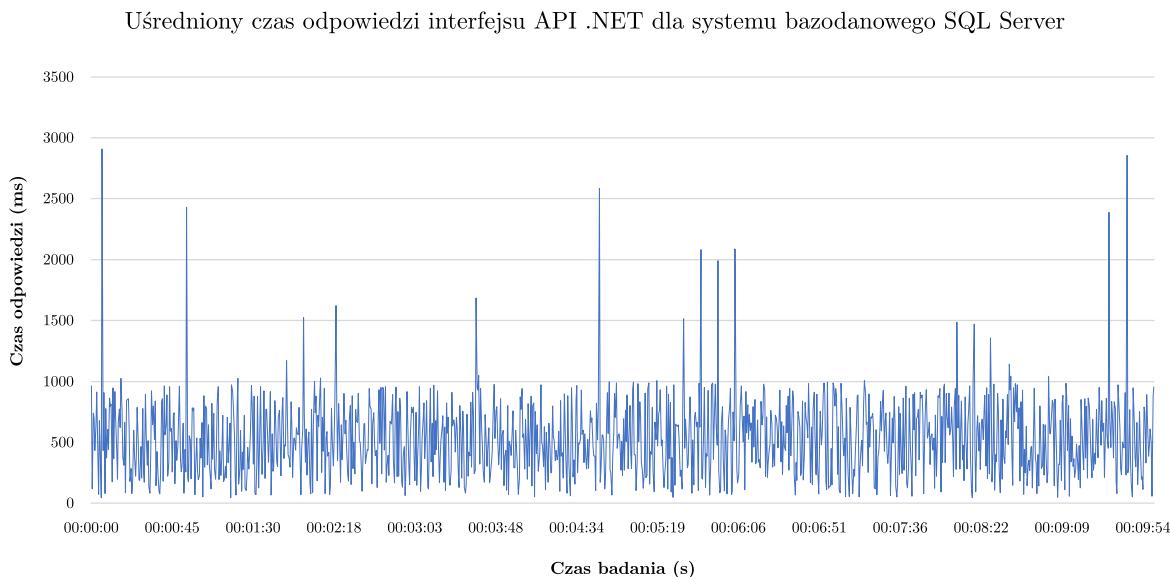
Zdecydowano się na wybór pięciu punktów końcowych obsługujących różne metody protokołu hipertekstowego, po to, aby weryfikować każdy rodzaj działania uwzględnianego w ramach funkcjonalności interfejsu API.

W tabeli 5.1 wyszczególniono każdy z wykorzystywanych punktów końcowych interfejsów programowania aplikacji, a także opisano sposób jego działania.

Tab. 5.1: Wykaz punktów końcowych wykorzystywanych w badaniu wpływu wykorzystanego systemu bazodanowego na działanie API

Identyfikator zasobu	Dostarczana zawartość	Rodzaj metody HTTP	Opis działania
/api/bills	pageSize - rozmiar strony (tj. liczba rekordów) - ustalony na stałe jako 30 pageNumber - numer strony - generowany zgodnie z jednostajnym rozkładem prawdopodobieństwa	GET	Zwrócenie listy encji identyfikujących rachunki wygenerowane w restauracji. Zarówno rozmiar strony, jak i liczba pomijanych rekordów zadane są jako parametr, a poszczególny element listy, zawiera zarówno dane encji podstawowej, jak i każdej z encji zależnych, powiązanych z nią kluczem obcym.
/api/orders/:id	id - identyfikator zamówienia wskazujący każdorazowo zamówienie istniejące w bazie danych	GET	Zwrócenie pojedynczej encji identyfikującej określone zamówienie poznane w ramach pobytu w restauracji. Zwrócony element zawiera zarówno dane encji podstawowej, jak i każdej z encji zależnych, powiązanych z nią kluczem obcym.
/api/products	saveProductBody - zasób zapisany w formacie JSON, dostarczający informacje o wartościach właściwości nowo tworzonego produktu - każdy z identyfikatorów encji zależnych wskazuje na istniejący obiekt	POST	Dodanie pojedynczej encji do zbioru danych z poprzedzającą ją walidacją danych dostarczonych w ramach ciała żądania.
/api/courses/:id	id - identyfikator posiłku zawartego w ramach zamówienia wskazujący każdorazowo istniejącą encję bazodanową updateCourseBody - zasób zapisany w formacie JSON, dostarczający informacje o wartościach właściwości modyfikowanego obiektu posiłku	PUT	Modyfikowanie encji bazodanowej dotyczącej posiłku zawartego w ramach zamówienia z uprzednią weryfikacją poprawności danych ciała żądania.
/api/reservations/:id	id - identyfikator rezerwacji, wskazujący na istniejący bądź nieistniejący zasób	DELETE	Usunięcie pojedynczego obiektu rezerwacji, bądź zwrócenie informacji o jego nie znalezieniu w systemie bazodanowym.

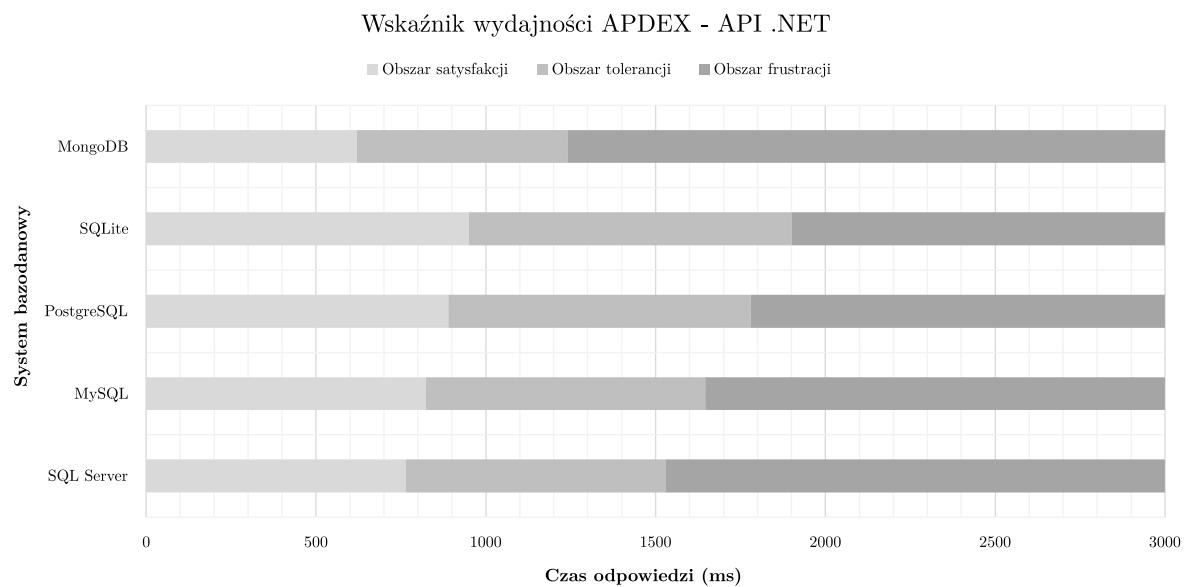
Na wykresie 5.1 przedstawiono zagregowane względem omówionych punktów końcowych, czasy odpowiedzi na żądanie w przypadku interfejsu programowania aplikacji zaimplementowanego z wykorzystaniem środowiska C#/.NET, a także komunikującego się z bazą danych SQL Server.



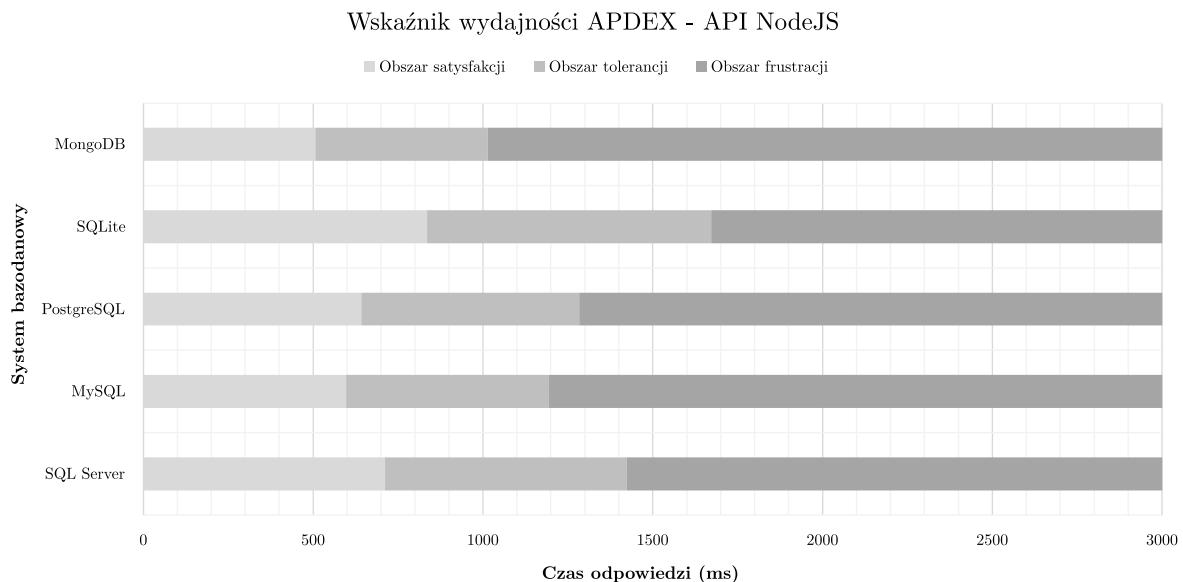
Rys. 5.1: Czas odpowiedzi na żądanie dla konfiguracji .NET/SQL Server w kontekście testu funkcjonalnego

Na podstawie analogicznych danych, dla każdej pary interfejs API - system bazodanowy, zdefiniowano progi tolerancji oraz frustracji stanowiące punkty odniesienia przy kalkulacji wartości wskaźnika APDEX. Wartości te, ustalono poprzez rosnące posortowanie zbioru czasów odpowiedzi API, a następnie dokonanie symetrycznego podziału dwupunktowego.

Uzyskane przedziały satysfakcji, tolerancji oraz frustracji względem każdego systemu bazodanowego oraz technologii tworzenia API zostały zobrazowane na wykresach 5.2 oraz 5.3.



Rys. 5.2: Obszary satysfakcji, tolerancji oraz frustracji wskaźnika APDEX względem poszczególnych systemów bazodanowych dla API zaimplementowanego w C#



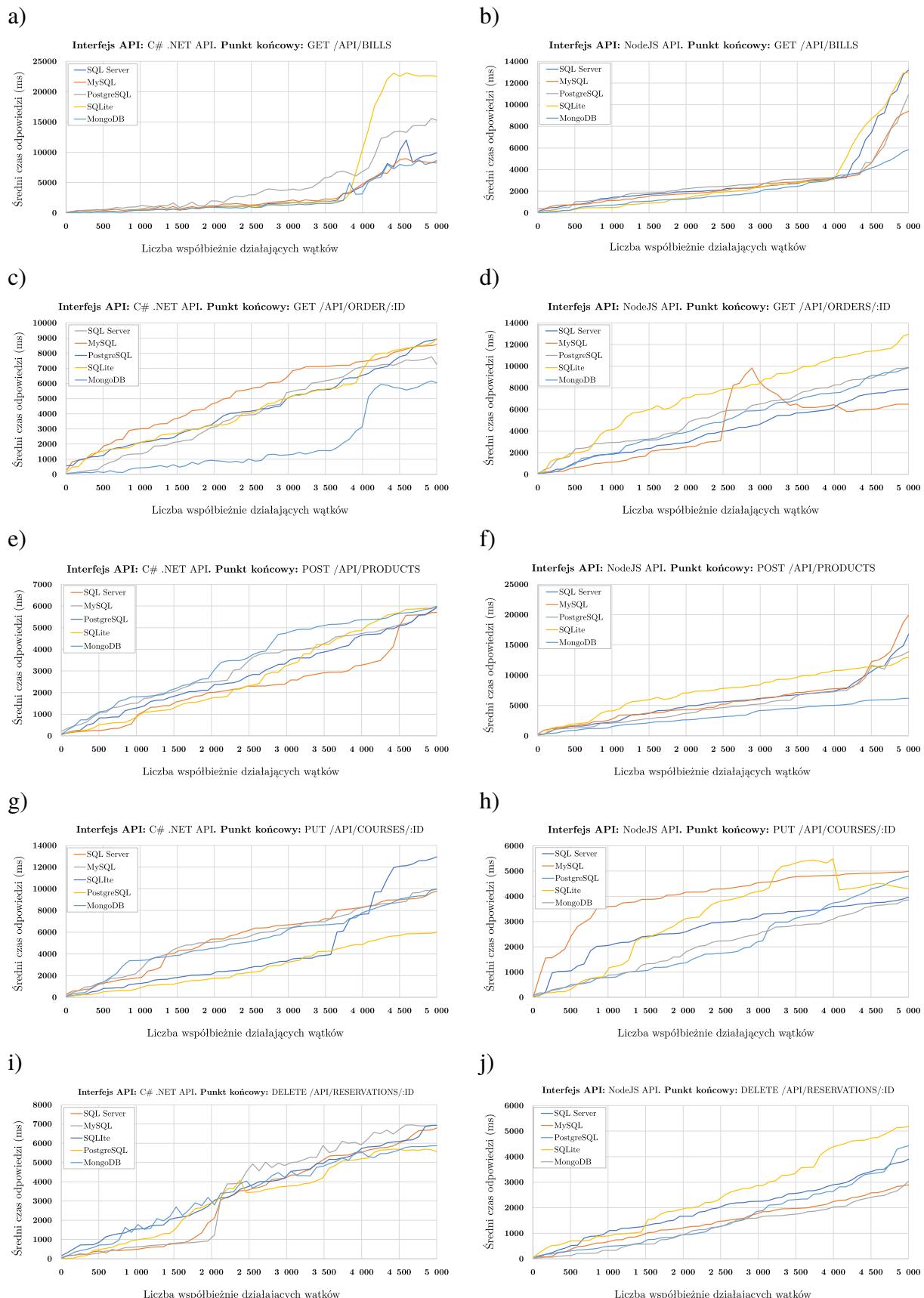
Rys. 5.3: Obszary satysfakcji, tolerancji oraz frustracji wskaźnika APDEX względem poszczególnych systemów bazodanowych dla API zaimplementowanego w JavaScript

Po spełnieniu warunków początkowych badania zrealizowano faktyczne czynności badawcze. Wykorzystując lokalną topografię fizyczną 4.1 rozpoczęto generowanie żądań protokołu hipertekstowego wytwarzanych przez dwa równolegle pracujące hosty sieciowe. Procedura badawcza trwała 20 minut i polegała na stopniowym zwiększeniu liczby współbieżnie pracujących procesów oprogramowania testującego rozpoczynając od jednego procesu a kończąc na pięciu tysiącach. Nowe wątki oprogramowania Apache JMeter uruchamiane były w stałych odstępach czasu, co implikuje niezmienność długości przedziału czasowego pracy w obrębie ustalonej liczby wątków. Czas ten, wynosi 240ms. Przy uwzględnieniu minimalnej zaobserwowanej średniej wartości natężenia generowanych żądań, wynoszącej 152 zapytania w ciągu sekundy, liczliwość zbioru próbek czasu zapytania w odniesieniu do dowolnego z 4990 poziomów przyrostu ruchu wyniosła co najmniej 36 elementów. Pierwsze 10 poziomów dotyczących liczby klientów generujących komunikaty nie pozwoliło na zarejestrowanie co najmniej 30 próbek, przez co elementy te, nie były brane pod uwagę w czasie opracowywania uzyskanych wyników. Opisane w niniejszym akapicie czynności zostały wykonane względem interfejsów programowania aplikacji wykorzystujących porównywane technologie, uwzględniając komunikację z każdym z pięciu systemów bazodanowych.

Na wykresach 5.4 a) do 5.4 j) zaprezentowano uśrednione względem liczbę pracujących wątków, czasy odpowiedzi na żądanie generowane w kierunku ewaluowanych interfejsów API.

Analizując wyniki uzyskane dla operacji pobierania kolekcji obiektów, zauważać można niewielkie różnice w wartości średnich czasów odpowiedzi, występujące w przedziale do około 4000 użytkowników. Różnice te, nie faworyzują ani deprecjonują któregokolwiek z rozwiązań.

5.1. Wpływ zastosowanego systemu bazodanowego na efektywność działania interfejsu API



Rys. 5.4: Średnie czasy odpowiedzi na żądanie względem liczby procesów generujących oraz systemu bazodanowego

Dla tego właśnie przedziału, jedyną zauważalną anomalię dotyczącą zmiany wydajności dostrzec można w kontekście interfejsu programowania aplikacji C# .NET korzystającego z silnika bazy danych PostgreSQL. W tym przypadku znaczące różnice pomiędzy wydajnością omawianego systemu, a efektywnością drugiego najgorszego rozwiązania odnotowano już dla około **720** wspólnie pracujących wątków. Dla tej właśnie metryki, omawiana różnica wynosi **374ms**. Wraz ze wzrostem liczby użytkowników o tysiąc, zauważać możemy różnicę **759ms**, a o dwa tysiące - **2326ms**. W momencie zaprzestania obserwacji tendencji nieznacznych dysproporcji pomiędzy systemami baz danych (moment ten możemy aproksymować do chwili działania 3750 jednocześnie działających procesów-generatorów) dywergencja czasów odpowiedzi osiągnęła wartość **4855ms**.

Dysproporcje wskazujące na niezaprzeczalną wyższość określonych rozwiązań nad pozostałymi dostrzegalne są dla rezultatów otrzymywanych w wyniku wspólnej pracy więcej niż czterech tysięcy procesów oprogramowania Apache JMeter.

Odwołując się do interfejsu zaimplementowanego w technologii NodeJS, zauważyciemy fakt, że spośród rozwiązań relacyjnych, najniższy skok wartości odnotowały rozwiązania MySQL oraz PostgreSQL. Ponadto, skok ten nastąpił zdecydowanie później, niż miało to miejsce w przypadku pozostałych rozwiązań. Wartym odnotowania są wyniki zaobserwowane dla nirelacyjnego systemu bazodanowego. W przypadku silnika bazy danych MongoDB, średnie czasy odpowiedzi były względnie niskie nie tylko do chwili uruchomienia czterech tysięcy wątków, ale również po tym czasie nie odnotowano gwałtownego wzrostu monitorowanej metryki. W momencie generowania maksymalnego natężenia ruchu sieciowego, zmierzony średni czas odpowiedzi wyniósł **5846ms**.

W przypadku rozwiązania zdefiniowanego na bazie technologii Microsoft, zauważalna jest bardzo niska wydajność obsługi komunikacji z rozwiązaniem SQLite (maksymalna zmierzona wartość to **22515ms**). Ponadto, silnik bazy danych PostgreSQL, analogicznie do obszaru niższego natężenia ruchu, notuje wyniki gorsze od swoich relacyjnych oraz nirelacyjnych odpowiedników w granicach od **2111ms** do **5373ms**.

W kontekście pobierania pojedynczego wyniku, wzrosty wartości średniego czasu odpowiedzi posiadają charakterystkę przybliżoną do charakterystyki liniowej. Wyjątkami są tutaj systemy bazy danych MongoDB dla interfejsu napisanego w języku C#, oraz silnik MySQL dla NodeJS API. W obu przypadkach pojawiają się gwałtowne wzrosty wartości czasu odpowiedzi. Odnosząc się do MongoDB zaobserwować możemy zmianę metryki wydajności od **1549ms** do **5945ms** na przestrzeni przyrostu wątków od liczby **3748** do **4253**. Zmiana ta, utrzymuje się do końca przeprowadzania testu. Dla interfejsu programowania aplikacji uruchamianego w środowisku NodeJS, oraz dla systemu bazodanowego MySQL, anomalia dostrzegana jest dla **2465** wątków a jej amplituda to **6721ms**. Co ciekawe, przekraczając poziom natężenia wynoszący **2890** wątków, średni czas odpowiedzi zaczyna spadać, aby w punkcie **4080** stanowić minimum w odniesieniu do pozostałych systemów baz danych.

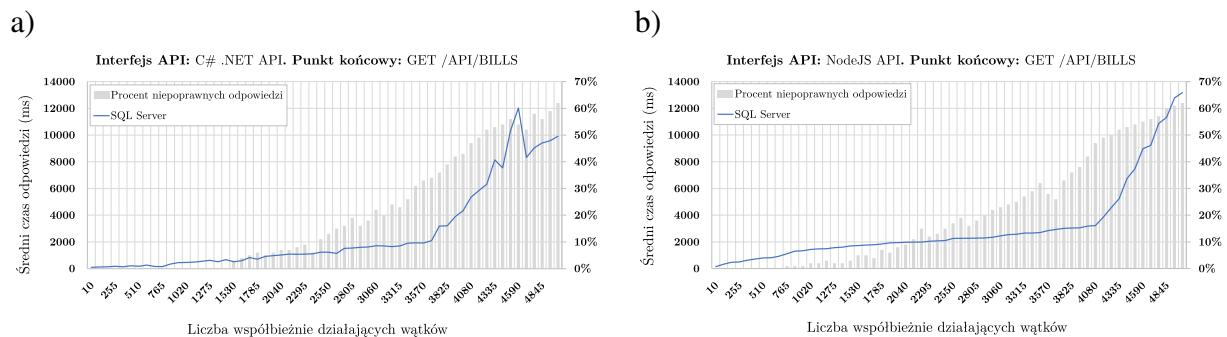
Poddając pod analizę procedurę zapisu danych do bazy, zgromadzone wyniki wykazują przewagę rozwiązania opartego o język C#, niezależnie od systemu bazodanowego. Ponadto, liniowa charakterystyka wzrostu metryk dla interfejsu JavaScript zakłócona została pojawieniem się nagłych spadków wydajnościowych. Analogicznie do punktu końcowego pobierania listy obiektów, najmniej znaczący spadek odnotować należy w odniesieniu do nirelacyjnego systemu MongoDB. W tym przypadku zmiana nastąpiła od wartości **5261ms** dla **4247** wątków do wartości **6203ms** dla **5000** wątków.

W aspekcie dwóch ostatnich procedur realizowanych przez punkty końcowe interfejsów programowania aplikacji, podobnie jak w przypadku funkcjonalności pozyskiwania pojedynczej encji, zaobserwować możemy przybliżone do liniowych, zmiany monitorowanego parametru wydajności. Spośród interesujących anomalii zaobserwowanych w ramach czterech ostatnich wykresów wyróżnić należy spadek wydajnościowy silnika bazodanowego SQLite w przypadku

modyfikacji encji za pomocą interfejsu API .NET. Spadek ten, identyfikowany jest poprzez wzrost średniego czasu odpowiedzi o **8018ms**, w przedziale czasowym w którym uruchomiono dodatkowe **847** wątków. Niestandardowym zachowaniem cechują się również systemy bazodanowe SQL Server oraz MySQL w odniesieniu do operacji usuwania pojedynczego rekordu poprzez API implementowane w języku C#. Interfejsy korzystające z obu mechanizmów przechowywania danych, do momentu osiągnięcia odpowiednio **1725** oraz **1955** wspólnie pracujących wątków generowania żądań, cechowały się wyjątkowo niskim średnim czasem odpowiedzi (tj. odpowiednio **827ms** oraz **922ms**).

Podsumowując, zarówno wskazanie wyższości jednej z technologii niezależnie od systemów baz danych, jak i wyróżnienie pojedynczego systemu bazy danych w kontekście realizowanej operacji nie jest możliwe biorąc pod uwagę kształt przeprowadzonego badania. Możliwym jest jednak zaobserwanie niskiego poziomu kompatybilności pomiędzy interfejsem API napisanym w języku C# oraz systemem baz danych SQLite. Ponadto, relatywnie wysoką wydajnością względem innych rozwiązań bazodanowych cechuje się system nierelacyjny MongoDB. W pięciu na dziesięć porównaniach wydajnościowych, przedstawionych na omawianych wykresach, to właśnie ten silnik baz danych najdłużej utrzymuje najniższą wartość średniego czasu odpowiedzi względem pozostałych rozwiązań.

Przedstawione powyżej wyniki nie dostarczają jednakże pełnego obrazu faktycznej wydajności ewaluowanych usług. Należy pamiętać, że uzyskana odpowiedź na żądanie nie musi być zawsze pozytywna, przez co nie zawsze niesie ona ze sobą informacje pożądaną dla klienta. Na wykresach 5.5 a) oraz 5.5 b) zestawiono ze sobą informacje o średnim czasie odpowiedzi na żądanie, a także o procencie żądań zakończonych niepomyślnie. Poprzez niepomyślne zakończenie żądania, rozumiano zarówno uzyskanie odpowiedzi o niepoprawnym kodzie i treści ciała, jak i zgłoszenie wyjątku protokołu hipertekstowego związanego z odmową aplikacji względem realizacji zapytania. Choć zobrazowano tu tylko i wyłącznie przypadek pojedynczej operacji oraz jednego systemu bazodanowego, analogiczne zachowanie zaobserwować można dla wszystkich pozostałych operacji i jest ono specyficzne dla interfejsu programowania aplikacji.



Rys. 5.5: Średnie czasy odpowiedzi na żądanie oraz procent niepoprawnych odpowiedzi względem liczby procesów generujących

Analizując przedstawione wykresy należy zwrócić uwagę zarówno na moment pojawiania się błędów obsługi żądania, jak i na stopień korelacji błędów z czasem przetwarzania zapytania. W przypadku interfejsu API opartego o technologię .NET widzimy, że procent błędów jest silnie powiązany ze zwiększającym się przedziałem czasowym obsługi komendy, a także intensyfikacją ruchu sieciowego. Oznacza to, że serwer w momencie kolejkowania żądań do przetworzenia, nie bierze pod uwagę ich liczby i stara się przetwarzać każde, jakie do niego dotrze. W związku z odmową realizacji funkcjonalności ze strony API, serwer zgłasza wyjątek protokołu hipertekstowego. Z racji konieczności wykonania pracy w kontekście każdego żądania, niezależnie od niewielkiego prawdopodobieństwa jego pomyślnej realizacji, wydłuża się czas uzyskania odpowiedzi na zapytanie klienckie.

Odnosząc się do interfejsu zaimplementowanego w technologii NodeJS, zaobserwować możemy odmienne zachowanie. Pierwszym aspektem na jaki należy zwrócić uwagę jest pojawienie się błędów obsługi żądania zdecydowanie wcześniej (tj. przy znacząco mniejszej liczbie uruchomionych wątków testowych). Oznacza to, że usługa nie jest w stanie radzić sobie na tyle dobrze z dostarczonym ruchem, jak robi to interfejs napisany w języku C#. Jednakże, wraz z rosnącym błędem procentowym nie zmienia się czas obsługi żądania. Oznacza to, że komponent serwerowy w ramach API nie przetwarza każdego z zapytań jakie zostanie do niego dostarczone, a także to, że w pewien specyficzny względem swojej charakterystyki sposób, API dokonuje wyboru tych spośród otrzymanych żądań, które mają zostać natychmiastowo odrzucone. Taki model przetwarzania komunikatów pozwala na utrzymanie niskiego poziomu czasu odpowiedzi, jednakże należy wziąć pod uwagę fakt, że nie musi być ona jednoznaczna z poziomem wydajności działania API.

5.2. Wpływ zastosowanej technologii na wydajność realizacji operacji współbieżnych

Niniejsze badanie, przeprowadzone zostało w celu zaobserwowania różnic dotyczących sposobu przetwarzania długo trwających operacji. Potencjalnie różnice te, związane mogą być z odmienną implementacją wewnętrznych mechanizmów przetwarzania współbieżnego, a także diametralnie innym podejściem obu technologii do zagadnienia wielowątkowości. Zdecydowano się na implementację algorytmu rozwiązującego symetryczny problem komiwojażera, bazującego na metaheurystyce genetycznej. Elementy charakterystyczne względem omawianego rozwiązania przedstawiono w sekcji 4.2. Napisane programy, zawarte są wewnątrz warstwy logiki biznesowej interfejsów API, a punktem wprowadzania danych dla algorytmów są określone punkty końcowe. W odniesieniu do przekazywanych danych wejściowych, wyróżnić należy ciało żądania, zawierające tablicę obiektów notacji JSON. Obiekty te, identyfikują określone lokalizacje poprzez wprowadzenie parametrów długości oraz szerokości geograficznej. Ponadto, przekazywany zostaje parametr czasu trwania głównej pętli algorytmu (tj. jak długo program powinien dokonywać kalkulacji), a także wartość wyniku optymalnego, względem której program powinien porównać uzyskany rezultat.

Odnosząc się do protokołu badawczego, zdecydowano się na wykonanie pomiarów jakości uzyskanego rozwiązania, a także liczby iteracji głównej pętli implementowanego algorytmu. Metryki te, zostały zebrane w kontekście uruchomienia programu dla piętnastu odmiennych zbiorów danych testowych, dostępnych w ramach otwartej biblioteki TSPLib [28]. Dla każdego ze zbiorów danych, trzydziestokrotnie powtórzono wywołanie algorytmu, w obrębie każdego z czterech różnych czasów wykonywania iteracji głównej programu. Czasy te to: 15s, 30s, 45s oraz 60s. W momencie przeprowadzenia badania, a także przez cały okres jego trwania, uruchomionych było 30 współbieżnie pracujących wątków będących generatorami żądań.

Przed rozpoczęciem realizacji omówionego protokołu przeprowadzono ewaluację funkcjonalną, stanowiącą warunek początkowy podjęcia badań. Ewaluacja ta, polegała na pięciokrotnym odwołaniu się punktów końcowych api obu technologii, wprowadzając jako dane wejściowe, zbiory współrzędnych dostępnych w ramach biblioteki TSPLib. Warto zaznaczyć, że są to zbiory inne, niż te wykorzystane następnie w faktycznym badaniu. Kryterium akceptacji warunku początkowego, było pomyślne wykonanie algorytmu w każdym ze zdefiniowanych przypadków funkcjonalnych, a także zwrócenie poprawnej odpowiedzi w czasie zgodnym z parametrem czasu wykonania algorytmu. Oba wymienione kryteria zostały w czasie ewaluacji funkcjonalnej spełnione.

Następnie, przygotowano lokalną topografię fizyczną nr. 4.1 oraz rozpoczęto generowanie żądań. Już na tym etapie, a jeszcze przed otrzymaniem wyników działania algorytmu, zaobser-

wowano interesujące zachowanie dotyczące obsługi dugo trwających żądań. Analizując czasy rozpoczęcia oraz zakończenia pracy dla poszczególnych wątków Apache JMeter, a także całkowity czas trwania badania dla każdego z interfejsów programowania aplikacji, dostrzeżono pełną sekwencyjność przetwarzania zapytań w przypadku API zaimplementowanego w technologii JavaScript / NodeJS, a także częściowe zrównoleglenie operacji w kontekście technologii C# .NET. W konsekwencji tego, ewaluacja usługi sieciowej opartej o NodeJS trwała 18 godzin i 52 minuty, podczas gdy badanie interfejsu napisanego w języku programowania C# - 5 godzin i 27 minut. Tak znacząca dysproporcja wynika ze sposobu zarządzania wykonaniem zapytań poprzez zastosowanie podejścia wielowątkowości, a także pracy z wykorzystaniem pojedynczego wątku.

Kiedy żądanie zacznie być przetwarzane przez interfejs programowania aplikacji NodeJS, jest ono wykonywane do momentu: uzyskania wyniku, przekroczenia dozwolonego czasu realizacji zapytania (*ang. timeout*), bądź też przekazania tokenu przerwania (*ang. cancellation token*). Jeżeli w czasie obsługi żądania pojawi się następne, musi ono zostać przekazane do kolejki, po to, aby stać się aktywnym po zakończeniu przetwarzania poprzedniej wiadomości.

Analizując wewnętrzne mechanizmy usługi sieciowej implementowanej w języku C# zauważycemożemy odmienne, niż zaprezentowane powyżej podejście. W związku z faktem utworzenia nowego wątku dla głównej pętli algorytmu, wątek podstawowy programu nie jest obciążony koniecznością realizacji jakichkolwiek dodatkowych operacji i oczekuje na uzyskanie wyników z procesu potomnego. Jeżeli w czasie oczekiwania pojawi się przychodzące żądanie, wątek główny zapisuje kontekst wywołania dla obecnego zapytania, przechodzi do realizacji kolejnego z nich, a po jego zakończeniu, bądź w momencie wyczekiwania na zakończenie, przełącza się do kontekstu poprzedniego zadania, aby zweryfikować jego status. Dzięki zastosowaniu takiego podejścia, które możliwe jest tylko w sytuacji dostępności wielu wątków w ramach pojedynczego programu, usługa sieciowa nie jest blokowana względem innych klientów.

Po ukończeniu obsługi wszystkich wygenerowanych żądań, przez oba systemy internetowe, uśredniono serie każdych 30 rezultatów, uzyskanych względem różnych zbiorów testowych oraz odmiennych czasów wykonania. Opracowane wyniki, przedstawiono w tabelach 5.2 oraz 5.3.

Analizując zgromadzone rezultaty, zauważycię należycią przewagę interfejsu programowania aplikacji zaimplementowanego w technologii C# .NET, w kontekście wartości współczynnika jakości rozwiązania. Na 60 uśrednionych wartości tego parametru, interfejs API NodeJS notuje wyniki lepsze zaledwie w 15 przypadkach. Ponadto, zauważycię, że niezależnie od technologii, czas wykonywania algorytmu nie zawsze musi przekładać się na uzyskanie lepszego rozwiązania. Niedeterministyczna charakterystyka algorytmu genetycznego jest powodem powstawania się gorszych rozwiązań, pomimo pracy algorytmu przez dłuższy czas. Odnosząc się do wartości liczby iteracji głównej pętli programu, różnice cechujące się określona tendencją nie są zauważalne. Wynika to z dwóch następujących faktów. Po pierwsze, kod źródłowy C#, z chwilą translacji do języka pośredniego (*ang. intermediate language*), ulega wewnętrznej optymalizacji przeprowadzanej bezpośrednio przez mechanizmy języka. Po drugie, mechanizmy wykorzystywane do przetwarzania list w C# dostępne w ramach biblioteki LINQ, wprowadzają dodatkowy narzut związany z koniecznością budowy wewnętrznej struktury danych związanej z przetwarzaną listą. Dlatego też, jakikolwiek wzrost efektywności związany ze wspomnianymi w pierwszym fakcie optymalizacjami, może zostać redukowany poprzez spadek wydajnościowy wprowadzany przez instrukcje przetwarzania list. Mechanizmy przetwarzania kolekcji w ramach języka JavaScript z kolei, nie wprowadzają dodatkowego opóźnienia w wykonywaniu kodu, jednakże na etapie interpretacji wydajność implementowanego programu nie ulega zmianie.

Odwołując się do różnic w kontekście opracowywanych kolekcji danych, zauważycię możliwą bardzo słabe rezultaty dla zbiorów **att48** oraz **att532**. Uzyskane w tych przypadkach długości najkrótszych tras są niemalże 5 krotnie większe, niż rozwiązania optymalne. Z drugiej strony,

Tab. 5.2: Wydajność realizacji algorytmu genetycznego dla problemu komiwojażera w zależności od czasu przetwarzania oraz technologii - współczynnik jakości rozwiązania

	C# / .NET				JavaScript / NodeJS			
	15s	30s	45s	60s	15s	30s	45s	60s
burma14	0,3462	0,6847	0,8475	0,8589	0,2983	0,5943	0,7636	0,7579
ulysses16	0,5299	0,3910	0,9270	0,6549	0,4817	0,2857	0,8297	0,5354
ulysses22	0,5271	0,4171	0,9312	0,7824	0,5488	0,4668	0,8728	0,6717
att48	0,0493	0,0611	0,3129	0,2623	0,0242	0,0238	0,2413	0,2903
berlin52	0,2316	0,2484	0,9718	0,8398	0,2239	0,3861	0,9202	0,7642
gr96	0,4798	0,1617	0,9274	0,7808	0,4874	0,1386	0,8606	0,7103
bier127	0,2313	0,1963	0,8473	0,8505	0,1162	0,1928	0,7906	0,7792
ch130	0,0947	0,1328	0,8072	0,8066	0,0575	0,1457	0,7521	0,6828
ch150	0,0784	0,1140	0,8584	0,7966	0,0812	0,1483	0,7900	0,7991
tsp225	0,2819	0,0988	0,7801	0,8311	0,3057	0,0826	0,8209	0,7533
att532	0,0194	0,0171	0,2226	0,2470	0,0116	0,0233	0,2650	0,1498
u574	0,5940	0,5454	0,6460	0,7872	0,4821	0,3782	0,5836	0,6573
u724	0,1620	0,0483	0,6529	0,7718	0,0943	0,0472	0,5747	0,6369
vm1084	0,0289	0,0273	0,5971	0,7938	0,0051	0,0381	0,5126	0,7962
d1291	0,1927	0,2930	0,5711	0,8421	0,1768	0,1016	0,5979	0,7438
Średnia ranga	3,0000	2,6667	7,0667	7,0000	2,1334	2,6667	6,0667	5,8000

Tab. 5.3: Wydajność realizacji algorytmu genetycznego dla problemu komiwojażera w zależności od czasu przetwarzania oraz technologii - liczba iteracji pętli algorytmu

	C# / .NET				JavaScript / NodeJS			
	15s	30s	45s	60s	15s	30s	45s	60s
burma14	71725	130514	201224	248972	71747	129134	205306	227232
ulysses16	68491	124929	174385	244773	68488	122619	192590	219521
ulysses22	33520	117580	164707	224099	63802	129397	176319	203753
att48	62550	120072	183853	233205	62843	134376	157470	198158
berlin52	72721	114196	190914	235500	72331	122215	182668	237310
gr96	62745	133156	172122	203419	63117	99062	162453	204787
bier127	66491	132354	179792	247307	66123	117603	188063	227924
ch130	62713	118125	174455	234333	62785	127119	159485	223465
ch150	66845	123024	179031	235174	66984	138917	185943	211357
tsp225	61923	124816	163056	202829	61479	126286	173199	224409
att532	65333	153374	188061	220558	65372	121134	173916	240112
u574	70850	133559	190742	231134	70639	106502	196297	248733
u724	69793	120363	205031	226060	69700	117866	200697	229731
vm1084	64393	111650	192909	225794	64545	122070	174608	227990
d1291	68067	120889	199096	235376	67843	140742	196741	238146

zaobserwować możemy rozwiązania znacząco bliskie optymalnym w odniesieniu do zbioru **gr96** dla interfejsu NodeJS oraz zbioru **bier127** dla interfejsu C# .NET.

W celu wykazania istnotnych statystycznie różnic dotyczących rezultatów zgromadzonych dla określonych konfiguracji badania, wykonano parowe testy statystyczne bazujące na teście Wilcoxona. Test ten, jest nieparametryczną odmianą procedury t-Studenta, zakładającą jako

hipotezę zerową zgodność wartości średkowych w odpowiadających sobie populacjach. Jako poziom istotności przyjęto wartość 0,05. Na podstawie tego właśnie testu, wyliczono średnie rangi dla poszczególnych czasów wykonania algorytmów w kontekście wartości współczynnika uzyskanego rezultatu do rozwiązania optymalnego.

Analizując przeprowadzoną ewaluację statystyczną, wykazać należy istotną wyższość rozwiązań uzyskanych przez interfejs programowania aplikacji języka C# w kontekście czasów wykonania równych 45s oraz 60s. W pozostałych przypadkach różnice nie są istotne statystycznie przy uwzględnieniu poziomu istotności 0,05. Macierz istotności statystycznej przedstawiona w tabeli 5.4.

Tab. 5.4: Macierz istotności statystycznej dla współczynników jakości rozwiązania pozyskanych w ramach badania wydajności obsługi operacji współbieżnych

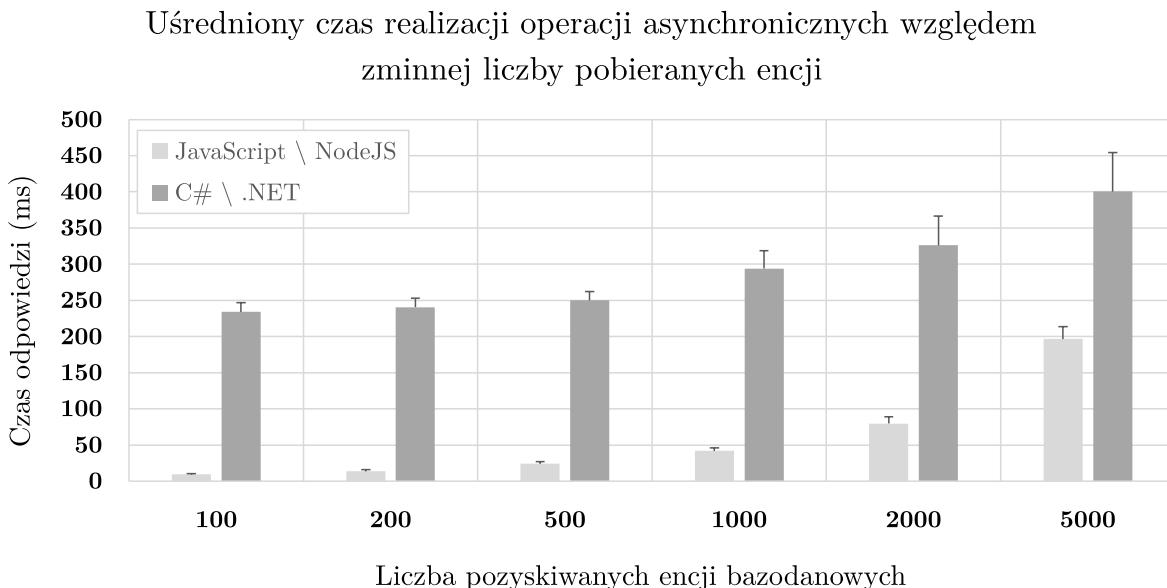
	15s (C#)	30s (C#)	45s (C#)	60s (C#)	15s (NodeJS)	30s (NodeJS)	45s (NodeJS)	60s (NodeJS)
15s (C#)	0	0	0	0	0	0	0	0
30s (C#)	1	0	0	0	1	0	0	0
45s (C#)	1	1	0	0	1	1	1	0
60s (C#)	1	1	1	0	1	1	1	1
15s (NodeJS)	0	0	0	0	0	0	0	0
30s (NodeJS)	0	0	0	0	1	0	0	0
45s (NodeJS)	1	1	0	0	1	1	0	0
60s (NodeJS)	1	1	0	0	1	1	0	0

5.3. Wpływ zastosowanej technologii na efektywność obsługi operacji asynchronicznych

W ramach omawianego badania dokonano analizy wydajności realizacji operacji asynchronicznych względem technologii wykorzystywanych do implementacji porównywanych interfejsów programowania aplikacji. Operacje te, znajdują szerokie zastosowanie wewnętrz kodu źródłowego usług sieciowych, będąc wykorzystywanym w celu uzyskania dostępu i zarządzania plikami, odwoływanie się do zewnętrznych źródeł danych, czy też synchronizowania egzekucji wybranych fragmentów kodu programu w czasie.

Zrealizowany eksperyment polegał na obserwacji czasów wykonania operacji asynchronicznych, względem zmiennej liczby encji pobieranych za ich pomocą. Elementy modelu danych, pozyskiwane były poprzez generowanie żądań protokołu hipertekstowego w kierunku zewnętrznej usługi sieciowej. Usługa ta, znajdowała się wewnątrz sieci lokalnej i została zaimplementowana intencjonalnie na potrzeby tego badania. Interfejs programowania aplikacji poddawany ocenie, korzystając z natywnego klienta protokołu hipertekstowego, wykonuje 30 iteracji, w ramach których pozyskuje kolejno 100, 200, 500, 1000, 2000, oraz 5000 encji bazodanowych. Dla każdej z wykonywanych operacji, odnotowany zostaje czas jej ukończenia, a także binarna wartość wskazująca na jej poprawność. W badaniu wykorzystano topologię fizyczną przedstawioną w sekcji 4.1, a także wariant planu testowego umówiony w punkcie 4.4. Wiąże się to z uruchomieniem grupy stu wątków w przedziale czasowym dziesięciu minut.

Pozyskane w ramach badania rezultaty uśredniono, a następnie zaprezentowano na wykresie 5.6. Zdecydowano się nie uwzględniać metryki procentowego błędu związanego z niepoprawnym wykonaniem żądań, ponieważ niezależnie od poddawanego analizie przypadku, był on równy zero.



Rys. 5.6: Uśredniony czas realizacji operacji asynchronicznych względem zmiennej liczby pobieranych encji

Zauważać należy znaczącą wyższość rozwiązania opartego o technologie JavaScript / NodeJS, względem oprogramowania utworzonego z wykorzystaniem C# .NET. Dla najmniejszej liczby pozyskiwanych encji, różnica średnich czasów odpowiedzi jest ponad 25 krotna. Wraz ze zwiększeniem liczebności pozyskiwanych obiektów modelu danych średni czas odpowiedzi dla NodeJS rośnie co prawda w szybszym tempie, niż ma to miejsce w kontekście rozwiązania uruchamianego na platformie .NET, jednakże nawet dla największej spośród liczb encji, dysproporcja wyników jest ponad dwukrotna. Warto zwrócić również uwagę na dyspersję poszczególnych rozwiązań względem przedstawionych średnich. Zaobserwować można zdecydowanie mniejsze odchylenia standardowe w odniesieniu do rozwiązania języka JavaScript, które zgodnie ze spodziewaną tendencją zwiększają się wraz z liczbą obiektów modelu danych. Dla dużych liczebności encji bazodanowych, rozwiązanie implementowane w języku C# w niewielkiej liczbie przypadków uzyskuje wyniki bliskie średniej.

Tak znacząca dyferencja w kontekście obu interfejsów programowania aplikacji, wynikać może z implementacji natywnych rozwiązań klienta protokołu hipertekstowego. W środowisku NodeJS, klient ten posiada niewiele opcji konfiguracyjnych, a jakiekolwiek bardziej zaawansowane żądania, realizowane są z wykorzystaniem zewnętrznych bibliotek. Ponadto, klient ten, jest częścią rdzennego modułu środowiska NodeJS, obsługującego zoptymalizowaną komunikację hipertekstową. Rozwiązanie służące do komunikacji HTTP dla języka C# jest mechanizmem dostarczonym przez biblioteki standardowe języka programowania, a nie samego środowiska uruchomieniowego. Oznacza to, że rozwiązanie dla tej właśnie technologii musi posiadać bardziej generyczną charakterystykę, aby móc być zastosowanym w dowolnym z przypadków użycia języka.

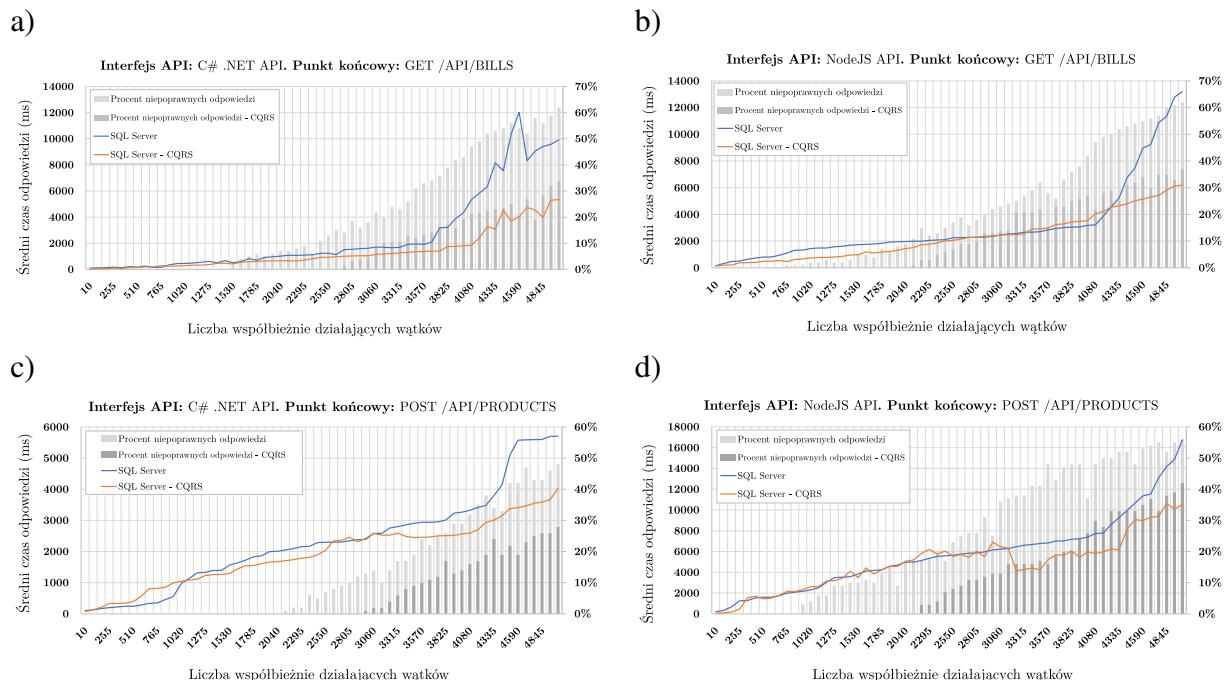
5.4. Wpływ implementacji wzorca projektowego CQRS na wydajność obsługi żądania

Celem niniejszego badania była obserwacja wpływu wdrożenia wzorca projektowego podziału odpowiedzialności, a także usprawnień wydajnościowych względem odseparowanych modeli

encji, na wydajność obsługi żądań dla ocenianych interfejsów programowania aplikacji. Wdrożenie wzorca projektowego miało na celu odseparowanie operacji dotyczących odczytu danych, od tych, dokonujących ich modyfikacji. Separacja ta, występowała zarówno na poziomie logicznym (tj. modelu danych), jak i fizycznym (tj. systemu bazodanowego). Dzięki temu, możliwe było wprowadzenie optymalizacji wydajnościowych względem modelu odczytu. Aby zachować spójność informacji pomiędzy bazami danych, wprowadzono ponadto mechanizm replikacji transakcyjnej, działający w warstwie systemu bazodanowego. Po wysłaniu wiadomości modyfikującej dane, określony rekord jest zmieniany w bazie zapisu, a następnie wysyłany jest komunikat synchronizacji zmian w kierunku bazy odczytu. Szczegóły implementacyjne dotyczące omawianego badania przedstawiono w sekcji 4.2.

Zastosowany protokół badawczy posiada analogiczną strukturę do tego, który został zaprezentowany w badaniu wpływu systemów bazodanowych. Oznacza to, że w przedziale dwudziestu minut, stopniowo zwiększano liczbę współbieżnie pracujących wątków oprogramowania testowego, osiągając szczytową wartość równą 5000. W tym przypadku jednak, poddano analizie tylko pojedynczy system bazodanowy, który wspiera usługę mechanizmu replikacji transakcyjnej - tj. Microsoft SQL Server. Kluczowym aspektem badania jest próba obserwacji czy, a także w jaki sposób wprowadzony model architektoniczny oraz usprawnienia wydajnościowe wpłyną na zmniejszenie się średniego czasu odpowiedzi, a także procentu błędnych odpowiedzi w stosunku do generowanych wiadomości. W badaniu zastosowano topologię fizyczną przedstawioną w sekcji 4.1, a także plan testowy wyszczególniony w punkcie 4.4. Ogół przeprowadzonych operacji jest zgodny z wyspecyfikowanym scenariuszem badawczym opisany w tabeli 3.4.

Na wykresach 5.7 a) do 5.7 d) przedstawiono kolejno porównanie procedur odczytu oraz zapisu, zarówno przed jak i po zastosowaniu omówionych modyfikacji.



Rys. 5.7: Wydajność działania interfejsów API dla operacji odczytu oraz zapisu, przed i po zastosowaniu wzorca podziału odpowiedzialności

Dla każdego z przypadków zauważać należy zarówno obniżenie się poziomu średniego czasu odpowiedzi, jak i procentowego błędu żądań niepoprawnych. Zachowanie takie, jest zrozumiałe ze względu na silną korelację obu metryk. Dostrzec należy relatywnie większą zmianę średniego czasu odpowiedzi w kontekście odczytu danych, niż ma to miejsce dla procedury ich zapisu. Wynika to z faktu, że dla zadania odczytu, dokonano usprawnień nie tylko związanych z

komunikacją z bazą danych (m.in. zdefiniowanie puli połączeń), ale również dotyczących formułowania zapytań, czy też wewnętrznej struktury poszczególnych pól danych. Wpływ pierwszego z wymienionych w poprzednim zdaniu aspektów widoczny jest szczególnie w odniesieniu do zmiany procenta niepoprawnych żądań. Tyczy się to zarówno procedury odczytu, jak i zapisu. Co więcej metryka ta, wpływa na średni czas odpowiedzi. Czym więcej kanałów komunikacji z serwerem bazodanowym posiada interfejs API, tym więcej żądań będzie mógł on obsłużyć bez konieczności czekania na zwolnienie połączenia.

Dla procedury odczytu danych, po wprowadzeniu opisywanych w niniejszym badaniu usprawnień, pierwszy błąd związany z niepoprawną obsługą zapytania, pojawił się przy natężeniu o **1273** wątki wyższym dla api napisanego w technologii C# .NET, oraz o **1527** wątków wyższym w przypadku NodeJS API. W kontekście metryki czasu wykonywania zapytania klienta, w momencie szczytowym zauważać można różnicę **4577ms** dla rozwiązania C# oraz **6987ms** dla systemu opartego o NodeJS.

Odnosząc się do procedury zapisu danych, przedział liczby wątków oprogramowania testowego, w ramach którego nie wystąpiły błędne żądania, rozszerzył się o **1190** procesów w przypadku interfejsu napisanego w języku JavaScript, a także o **765** procesów w przypadku api implementowanego w C#. Doprowadziło to do spadku średniego czasu odpowiedzi w momencie szczytowym o **6225ms** dla pierwszej z wymienionych technologii, a także o **1677ms** dla drugiej z nich.

Analizując powyższe rezultaty, należy także spojrzeć na nie przez pryzmat ilości zmian, jakie zostały wprowadzone dla poszczególnych technologii. Mówiąc o interfejsie programowania aplikacji języka JavaScript, modyfikacje sprowadzały się do usprawnień na poziomie bazy danych, a także w kontekście momentu uruchamiania instancji mapera obiektowo-relacyjnego. Pomimo stosunkowo niewielkiej liczby zaimplementowanych adaptacji, wydajność rozwiązania wzrosła w sposób znaczący. Lekko odmienną tendencję zauważać można dla usługi sieciowej napisanej w języku C#. W tym przypadku wprowadzono nie tylko analogiczne względem konkurenta poprawki, ale także dodatkowo skorzystano z mechanizmów poprawy wydajności, specyficznych względem tylko tej technologii. Co prawda implementacja określonych modyfikacji wpłynęła pozytywnie na efektywność działania interfejsu, to zmiana ta, nie jest tak znacząca, jak dla rozwiązania opartego o technologię JavaScript / NodeJS.

5.5. Porównanie efektywności obsługi zapytań dla odmiennych mechanizmów pamięci podręcznej

W przedstawionym badaniu dokonano porównania wydajności obsługi żądań klienckich, dotyczących pozyskiwania obiektów danych, w odniesieniu do wykorzystania zaimplementowanych mechanizmów pamięci podręcznej. Wydajność w ramach przeprowadzonej ewaluacji rozumiana była poprzez czas odpowiedzi na żądanie. Warto zaznaczyć, że nie uwzględniono parametru dotyczącego liczby niepoprawnych zapytań ponieważ dla żadnej z próbek, nie uzyskano nieprawidłowej odpowiedzi.

Zdecydowano się na wdrożenie dwóch mechanizmów obsługi pamięci podręcznej. Działanie pierwszego z nich, nazywanego mechanizmem statycznym, polega na zapamiętywaniu odpowiedzi w kontekście żądania o określonych właściwościach (tj. identyfikator zasobu, parametry zapytania), a także przetrzymywaniu jej w buforze przez stały, określony czas. Jeżeli zapytanie charakteryzujące się znaną uprzednio strukturą, zostanie odebrane w tym właśnie czasie, odpowiedź nie zostanie pobrana z bazy danych, a dostarczona bezpośrednio z omawianego bufora. Drugi mechanizm jest autorskim pomysłem twórcy niniejszej pracy i uwzględnia zmienność czasu przechowywania wpisu w odniesieniu do częstości wywoływań, a także unieważnienia danych przechowywanych w pamięci cache. Mechanizm ten, wykorzystuje ponadto osobną struk-

turę danych, która jest aktualizowana przy wywołaniu dowolnego punktu końcowego, a gromadzone w niej dane, dostarczają informacji niezbędnych do wyliczania czasu zapamiętywania wpisu w cache. Niezależnie od omawianego mechanizmu, punkty końcowe odpowiedzialne za modyfikację danych, dla których wpisy przechowywane są w pamięci podręcznej, uruchamiają procedurę unieważnienia określonych obiektów encji. Szczegóły implementacyjne dotyczące obu mechanizmów pamięci podręcznej przytoczone zostały w sekcji 4.2.

Odoszcząc się do procedury badawczej, wykonane zostało 20 eksperymentów, w ramach których 30 współbieżnie pracujących wątków generowały żądania w kierunku dwóch punktów końcowych. Pierwszy z nich, dotyczył pobrania listy encji o stałym rozmiarze, natomiast drugi - uzyskania pojedynczego obiektu danych. Dla każdej z ewaluacji, przeprowadzanej w kontekście określonego interfejsu programowania aplikacji, zdefiniowano pięć zdarzeń unieważnienia danych (tj. momentów wywołania punktu końcowego modyfikującego encje). Ponadto zaznaczyć należy, że czas trwania pojedynczego eksperymentu był stały i wynosił 20 minut. Co więcej, wprowadzona została celowa dysproporcja pomiędzy liczbą żądań pozyskujących listę obiektów, a liczbą zapytań, których rezultatem jest zwrócenie tylko jednego z nich. Stosunek ten, wynosił dwa do jednego. Taka struktura badań, wynika z charakterystyki dotyczącej sposobu wyliczania czasu przechowywania wpisu w pamięci podręcznej w odniesieniu do autorskiego mechanizmu. Promuje on, te spośród żądań, których częstotliwość wywołań jest większa.

Dla statycznego mechanizmu pamięci cache zastosowano czas życia wynoszący 60s, podczas gdy czasem referencyjnym mechanizmu autorskiego było 120s. Co więcej, wewnętrzna struktura wykorzystywana w rozwiązaniu autorskim, została wygenerowana na podstawie liczby żądań wysłanych w czasie realizacji eksperymentu ze stałym czasem przechowywania wpisu. Dzięki temu, punkt końcowy dotyczący pobierania listy elementów, był postrzegany jako element wywoływany z większą częstotliwością.

Badanie przeprowadzono zgodnie z koncepcją zarysowaną w scenariuszu badawczym 3.5, wykorzystując konfigurację lokalnej topologii fizycznej przedstawionej w sekcji 4.1.

W tabeli 5.5 zaprezentowano elementy charakterystyczne dla każdego z dwudziestu eksperymentów, a także rezultaty jakie w kontekście nich odnotowano.

Tab. 5.5: Wydajność metod obsługi pamięci podręcznej względem technologii oraz momentu unieważnienia wpisu

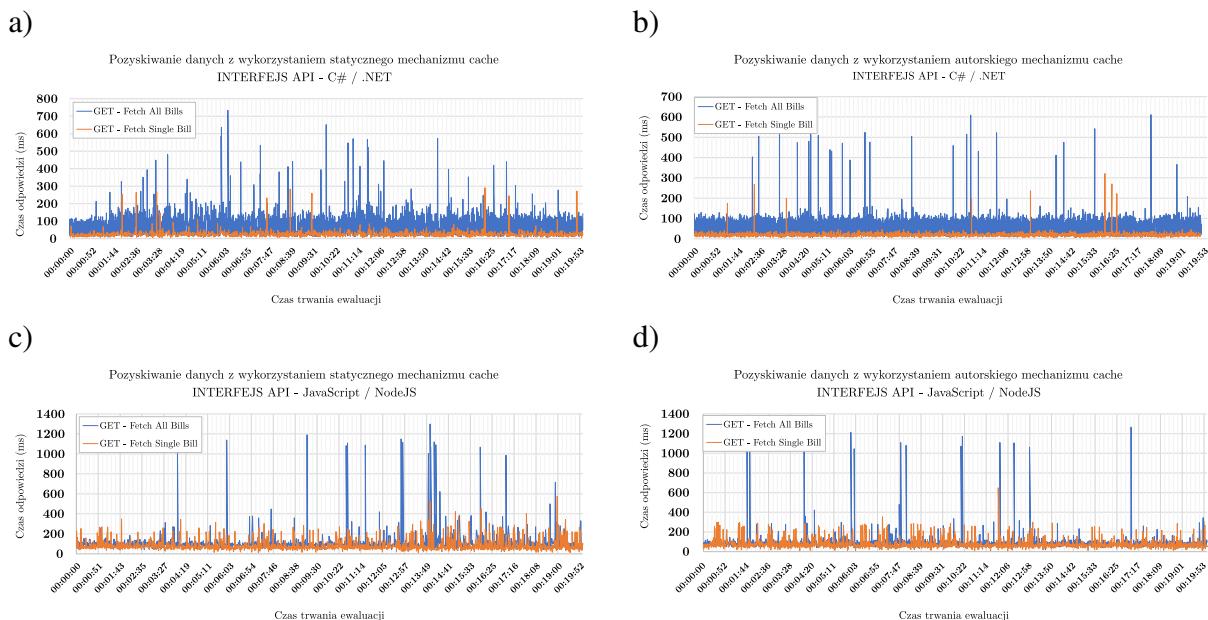
	Typ metody	Moment unieważnienia					Liczba żądań >500ms	Zysk względem rozwiązania alternatywnego
		#1	#2	#3	#4	#5		
C# / .NET	statyczna	06:40	10:00	13:20	15:00	19:40	10	-
	autorska	06:40	10:00	13:20	15:00	19:40	9	13,77 ms
C# / .NET	statyczna	00:32	03:45	06:52	07:49	16:40	8	20,16 ms
	autorska	00:32	03:45	06:52	07:49	16:40	12	-
C# / .NET	statyczna	02:44	06:30	14:27	16:55	19:42	14	-
	autorska	02:44	06:30	14:27	16:55	19:42	10	12,54 ms
C# / .NET	statyczna	01:15	05:29	08:31	14:48	17:45	15	-
	autorska	01:15	05:29	08:31	14:48	17:45	13	19,29 ms
C# / .NET	statyczna	02:01	07:32	12:00	13:27	15:56	16	24,85 ms
	autorska	02:01	07:32	12:00	13:27	15:56	20	-
JS / NodeJS	statyczna	06:40	10:00	13:20	15:00	19:40	17	-
	autorska	06:40	10:00	13:20	15:00	19:40	14	18,13ms
JS / NodeJS	statyczna	00:32	03:45	06:52	07:49	16:40	13	23,14 ms
	autorska	00:32	03:45	06:52	07:49	16:40	15	-
JS / NodeJS	statyczna	02:44	06:30	14:27	16:55	19:42	11	11,97 ms
	autorska	02:44	06:30	14:27	16:55	19:42	16	-
JS / NodeJS	statyczna	01:15	05:29	08:31	14:48	17:45	14	-
	autorska	01:15	05:29	08:31	14:48	17:45	8	21,91 ms
JS / NodeJS	statyczna	02:01	07:32	12:00	13:27	15:56	15	-
	autorska	02:01	07:32	12:00	13:27	15:56	13	14,35 ms

Moment unieważnienia rozumiany jest jako punkt na osi czasu, kiedy nastąpiło wywołanie żądania aktualizacji encji bazodanowych. Czasy odpowiedzi na żądanie oraz liczby żądań powyżej 500ms odnoszą się do punktu końcowego pozyskiwania kolekcji obiektów danych.

Zauważać należy, że autorskie rozwiązanie zakładające zmienny czas przechowywania wpisu w pamięci podręcznej, notuje zysk względem rozwiązania konkurencyjnego w sześciu przypadkach na dziesięć zdefiniowanych. Minimalna wartość zysku, w kontekście eksperymentów to **12,54ms** - interfejs programowania C# .NET, natomiast maksymalna to **21,91ms** - interfejs JavaScript / NodeJS. Niewątpliwie, zaobserwować można silną korelację pomiędzy wartością omawianego zysku a liczbą żądań, których czas trwania wynosi ponad 500ms. Bardzo prawdopodobnym jest, że dla większości z tych zapytań, realizowana jest komunikacja z bazą danych wskutek braku określonego wpisu w pamięci cache. Analizując przewagi rozwiązań w obrębie każdego z eksperymentów, zauważać można, że za każdym razem, gdy liczba żądań o czasie odpowiedzi powyżej 500ms jest wyższa, to przekłada się to na gorszy wynik średni.

Ponadto, wartym odnotowania jest fakt, że autorskie rozwiązanie zarządzania wpisami pamięci podręcznej, wykazuje lepsze rezultaty w tych przypadkach, kiedy zakresy pomiędzy momentami unieważnień są stosunkowo szerokie. W związku z uzyskaniem dłuższego czasu życia przez określony element gromadzony w pamięci podręcznej, żądanie pobierające dane z systemu bazodanowego może zostać opóźnione, a co za tym idzie, średni czas odpowiedzi może być niższy. Sytuacja staje się z goła odmienna, z chwilą częstej inwalidacji wpisu pamięci cache na przestrzeni czasu, co prowadzi do ograniczenia długości czasu jego życia.

Na wykresach 5.8 a) do 5.8 d) pokazano faktyczne czasy odpowiedzi, dla żądań generowanych w kontekście pierwszego z eksperymentów przedstawionych w tabeli.



Rys. 5.8: Porównanie czasów odpowiedzi na żądanie dla odmiennych mechanizmów pamięci podręcznej

W kontekście interfejsu programowania aplikacji zaimplementowanego w języku C# zauważać możemy znaczące wydłużenie się przedziałów czasowych, w ramach których nie występują żądania o wysokim czasie odpowiedzi. Trend ten, nie jest jednakże tak dobrze dostrzegalny w odniesieniu eksperymentu przeprowadzanego z wykorzystaniem interfejsu API napisanego w JavaScript. Ponadto, niezależnie od technologii, nie jest zauważalne spodziewane, stopniowe rozszerzanie się odcinków niskich czasów odpowiedzi, wraz z postępem ewaluacji. Może to być spowodowane nieadekwatnymi wartościami liczników unieważnień, które określone zostały na

podstawie poprzednich eksperymentów wykorzystujących mechanizm o stałym czasie przechowywania wpisu.

5.6. Zmienność wydajności api wdrożonego na generycznej oraz dedykowanej platformie chmurowej

Ostatnia ze zrealizowanych w ramach niniejszej pracy ewaluacji odnosi się do analizy zmiany wydajności pracy interfejsów programowania aplikacji w zależności od środowiska wdrożeniowego, w ramach którego usługa sieciowa jest hostowana. Celem niniejszego badania jest sprawdzenie, czy wykorzystanie dedykowanej rozwiązań platformy chmurowej, pozwala na uzyskanie lepszych rezultatów działania api, niż w przypadku wdrożenia systemu internetowego na infrastrukturze generycznej, którą jest wirtualny serwer prywatny.

Zdecydowano się na zastosowanie usługi w modelu infrastructure-as-a-service, która pełniła rolę środowiska produkcyjnego niezależnego od technologii implementacji api. W środowisku tym, wdrożono interfejsy programowania aplikacji napisane w językach C# oraz JavaScript, wykorzystując serwer HTTP Apache oraz usługę odwróconego proxy. Ponadto, każda z poddawanych analizie usług, komunikowała się z silnikiem bazodanowym znajdującym się wewnątrz wirtualnego serwera prywatnego. Dzięki temu, wyeliminować można było zjawisko zmienności czasu połączenia pomiędzy interfejsem a źródłem danych, która to zmienność wynika z niedeterministycznego charakteru łącza sieciowego. Aby zachować pełną generyczność omawianego rozwiązania zastosowano jeden z najpopularniejszych systemów bazodanowych jakim jest MySQL. System ten, choć w znaczącym stopniu wspierany przez aplikacje tworzone w językach C# oraz JavaScript, nie jest rozwiązaniem postrzeganym jako dedykowane względem platform .NET oraz NodeJS.

Jako podejście referencyjne względem infrastruktury generycznej, wdrożone zostały interfejsy programowania aplikacji na platformach Microsoft Azure oraz Heroku. Pierwsza z nich, jest usługą dedykowaną dla rozwiązań bazujących na platformie .NET oraz napisanych w języku C#. W tym przypadku, interfejs programowania aplikacji połączono z bazą danych Microsoft SQL Server. Druga z platform to rekomendowane rozwiązanie dla systemów internetowych tworzonych z wykorzystaniem środowiska uruchomieniowego NodeJS. Interfejs API wdrożony na platformie Heroku, skomunikowany został z nierelacyjną bazą danych MongoDB.

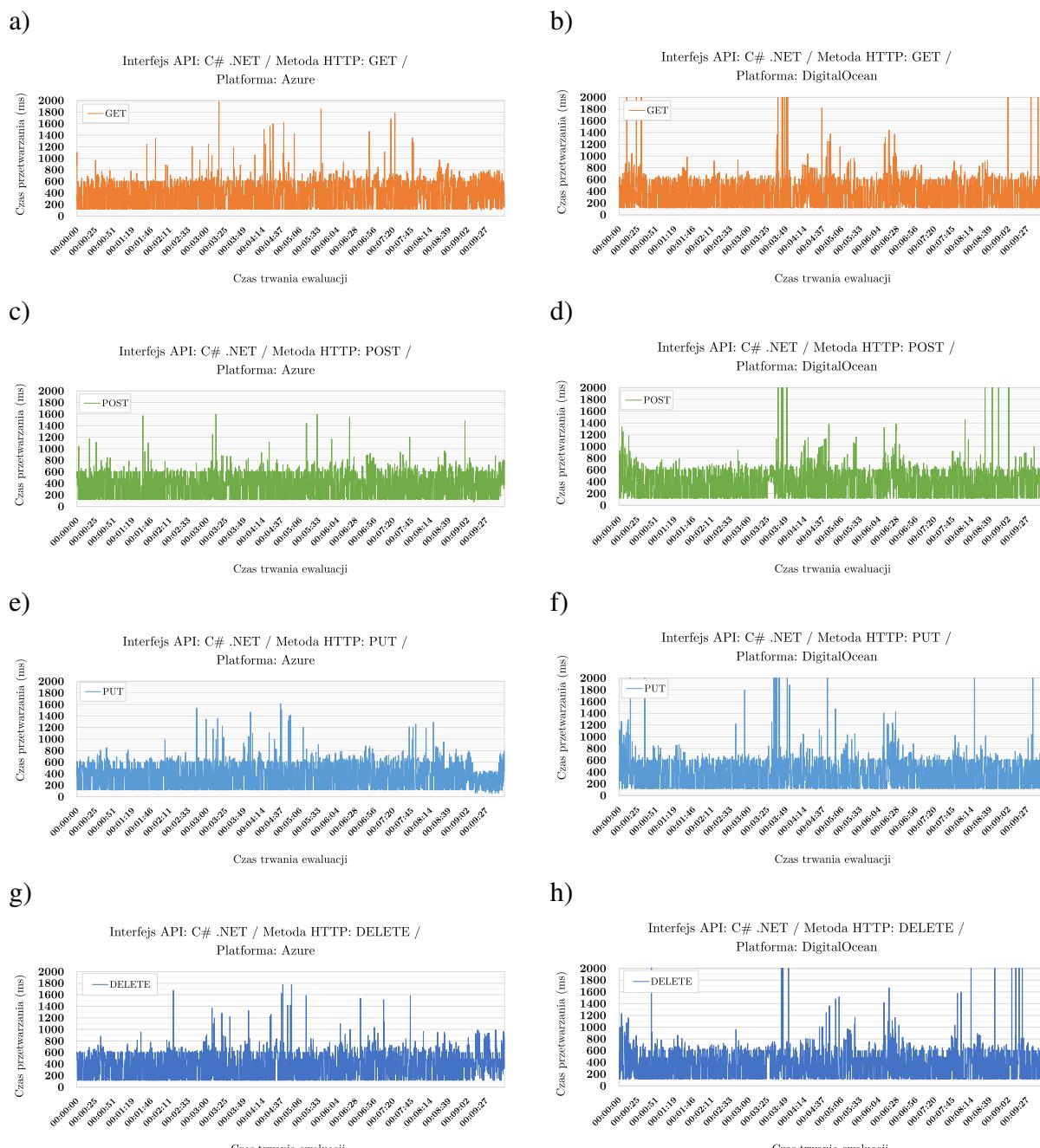
Odnosząc się do metryk gromadzonych w ramach niniejszego badania, wspomnieć należy o fakcie, że rezultaty wykonywanych operacji nie stanowią wartości czasu odpowiedzi na żądanie, a jedynie czas przetwarzania zapytania wewnątrz metody kontrolera interfejsu programowania aplikacji. Takie podejście, pozwala na uniezależnienie ewaluacji względem lokalizacji oraz szybkości łącza internetowego urządzeń generujących zapytania.

W przeprowadzonym badaniu generowano zapytania pod pięć punktów końcowych, które wykorzystywane były również w eksperymencie 5.1. Ich szczegółowa charakterystyka zawarta została w tabeli 5.1. Przed rozpoczęciem analizy, zestawiono rozproszoną konfigurację topologii fizycznej, która opisana została w sekcji 4.1, a także dokonano ewaluacji funkcjonalnej zaimplementowanych rozwiązań, poprzez realizację testów linii bazowej. Plan testowy obejmował uruchomienie 40 współbieżnie pracujących wątków oprogramowania Apache JMeter, które generowały ruch sieciowy przez 10 minut. Całość badania była przeprowadzona zgodnie ze zdefiniowanym uprzednio scenariuszem badawczym 3.6.

Kluczowymi wskaźnikami wydajności rozwiązań wdrażanych na platformach chmurowych są: czas realizacji pojedynczego zapytania, procent błędnych odpowiedzi w kontekście poprawnych zapytań, a także efektywność wykorzystania zasobów fizycznego sprzętu. Zdecydowano się na pominięcie ostatniego z przytoczonych wskaźników, ponieważ w odniesieniu do tak przygotowanego badania, nie dostarcza on informacji, które mogłyby być przesłanką do formułowa-

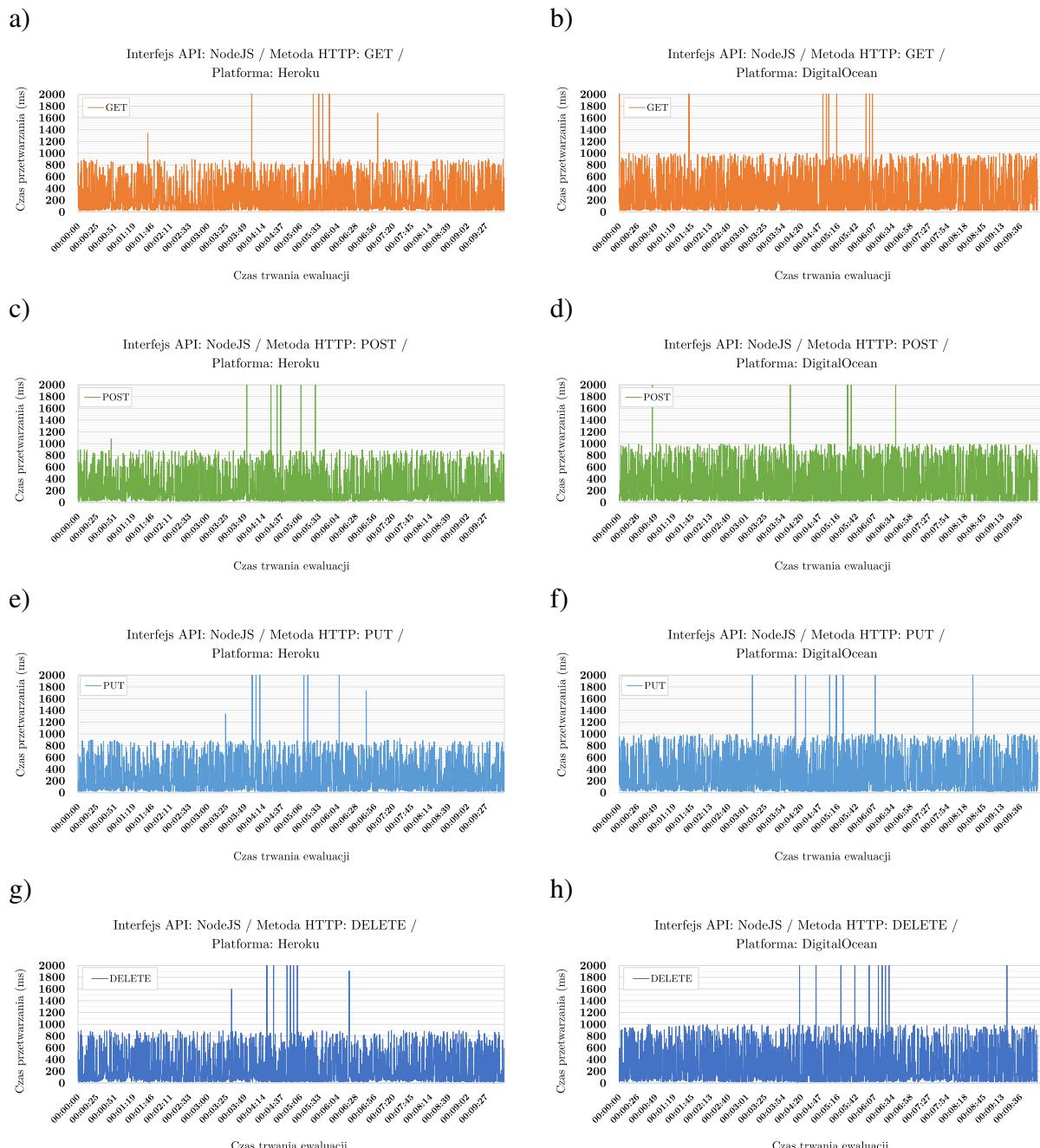
nia jakichkolwiek hipotez dotyczących omawianego problemu. W czasie generowania zapytań w kierunku interfejsów programowania aplikacji, niezależnie od platform wdrożeniowych oraz uruchomieniowych, konsumpcja fizycznych zasobów sprzętowych cechowała się relatywną stałością w czasie. Oznacza to, że każda z poddawanych analizie platform chmurowych, jest w stanie obsłużyć znaczco większy ruch, niż ten, generowany w ramach tego badania. Przeanalizowano natomiast, zarówno czas przetwarzania zapytania wewnątrz interfejsu, jak i liczbę błędnych zapytań w stosunku do wszystkich wygenerowanych.

Na wykresach 5.9 a) do 5.9 h) zawarto porównanie rezultatów pozyskanych w czasie ewaluacji interfejsu programowania aplikacji napisanego w języku C# i uruchamianego na platformach: generycznej (tj. DigitalOcean VPS), oraz dedykowanej (tj. Microsoft Azure).



Rys. 5.9: Wydajność działania mierzona czasem wewnętrznego przetwarzania operacji CRUD dla platform Azure oraz DigitalOcean - Interfejs API C# .NET

Wskazać należy nieznaczącą, jednakże występującą przewagę rozwiązania opartego o usługę platform-as-a-service. Jak możemy zauważyć, przewaga ta, występuje niezależnie od typu generowanego żądania klienta. Ponadto, w przypadku rozwiązania generycznego zaobserwowano pojawienie się większej liczby zapytań klienckich, których czasy odpowiedzi nie mieściły się w przedziale definiującym 97% wszystkich próbek (tj. od 200ms do 580ms) dla rozwiązania dedykowanego. Szczególnie trend ten, widoczny jest dla żądań metody HTTP typu PUT. Wartym odnotowania jest fakt niewystępowania jakiegokolwiek wyniku w przedziale od 0 do 119ms. Wyniki takie, były bez trudu osiągane w środowisku lokalnym dla małej liczby współbieżnie pracujących generatorów. Zjawisko to, jest specyficzne względem względem technologii C# .NET i nie uwidacznia się w przypadku interfejsów uruchamianych w NodeJS.

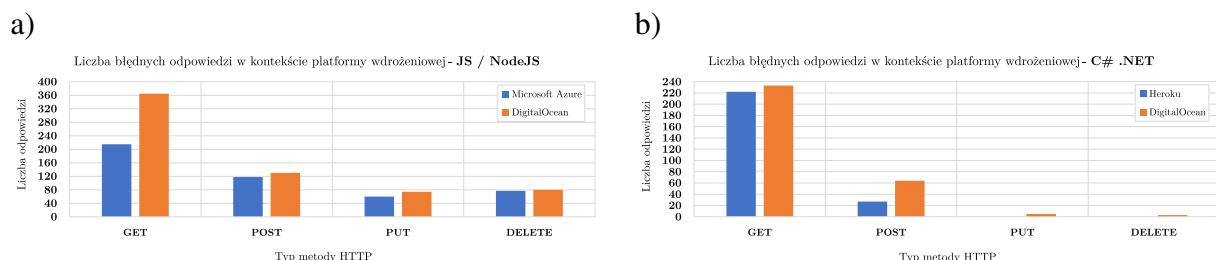


Rys. 5.10: Wydajność działania mierzona czasem wewnętrznego przetwarzania operacji CRUD dla platform Heroku oraz DigitalOcean - Interfejs API NodeJS

Kolejna z zaprezentowanych wizualizacji, dotyczy z kolei interfejsu API zaimplementowanego w oparciu o technologię NodeJS, który uruchomiony został na platformie Heroku (konfiguracja dedykowana), a także serwerze udostępnianym w ramach chmury DigitalOcean (konfiguracja generyczna). Zgromadzone czasy przetwarzania żądań dla odmiennych typów operacji pokazane zostały na wykresach 5.10 a) do 5.10 h).

Odnosząc się do analizowanej technologii, wzrost wydajności rozwiązania dedykowanego względem generycznego jest wydatny, i niezależnie od wybranego typu wykonywanej operacji, różnica wartości metryki go opisującą jest niemniejsza niż 237ms. Ponadto, w momencie w którym 99,98% wszystkich obserwacji należy do przedziału od 34ms do 900ms w kontekście rozwiązania dedykowanego, tylko 72,7% próbek badania dla rozwiązania generycznego, zawarte jest w analogicznym przedziale. Zauważać należy fakt braku zwiększenia się intensywności żądań dla czasów przetwarzania powyżej 1000ms, w odniesieniu do rozwiązania generycznego. Co prawda zapytania trwające powyżej 1000ms pojawiają się w tego typu konfiguracji, to podobnie do rozwiązania dedykowanego, są to obserwacje jednostkowe.

Koleijną metryką wziętą pod uwagę w ramach niniejszego badania jest liczba niepoprawnych odpowiedzi, które zostały zwrocone dla wygenerowanych zapytań. Podobnie jak dla wizualizacji zaprezentowanych powyżej, w tym przypadku, analiza również prowadzona będzie pod kątem obserwacji różnic wynikających z zastosowania odmiennych platform wdrożeniowych, a nie technologii implementacji interfejsów api. Na wykresach 5.11 a) oraz 5.11 b) przedstawione wartości odnotowane względem omawianej metryki.



Rys. 5.11: Liczba błędnych odpowiedzi względem typu żądania oraz platformy wdrożeniowej

W przypadku interfejsu implementowanego z wykorzystaniem platformy NodeJS oraz języka JavaScript różnice wydajnościowe ujawniają się jeszcze bardziej w momencie w którym brana pod uwagę jest liczba niepowodzeń realizacji zapytań klienckich. Niezależnie od typu operacji, rozwiązanie wdrożone na platformie Heroku, generowało mniejszą liczbę błędnych odpowiedzi. Przytaczając najbardziej skrajny przypadek (tj. żądania typu GET), liczba niezrealizowanych poprawnie zapytań dla platformy generycznej jest ponad dwa razy większa, od tej dla platformy dedykowanej.

Analogiczną tendencję zaobserwować możemy w przypadku interfejsu programowania aplikacji C# .NET. Tu również, niezależnie od metody protokołu hipertekstowego, rozwiązanie rekomendowane przez twórcę technologii posiada przewagę. Przewaga ta, najbardziej widoczna jest dla żądania typu PUT i wynosi ona 37 niepoprawnych odpowiedzi. Warto zaznaczyć, że całkowita liczba błędnie zrealizowanych żądań dla metody PUT oraz platformy dedykowanej wynosi 26.

Wyniki przeprowadzonego w tej sekcji badania posiadają najbardziej jednoznaczny charakter spośród wszystkich rezultatów otrzymanych w kontekście każdego z przeprowadzonych badań. Co więcej, rezultaty te, prowadzą do sformułowania popartego badaniami wniosku, że niezależnie od faktu, która z technologii implementacyjnych zostanie wybrana, a także jakiego typu operacje będą realizowane, system internetowy przygotowany do przejścia w fazę produkcyjną, powinien być wdrażany z wykorzystaniem narzędzi oraz infrastruktur dedykowanych względem

5.6. Zmienność wydajności api wdrożonego na generycznej oraz dedykowanej platformie chmurowej

określonej technologii. Usprawnienia wydajnościowe, wprowadzane przez twórców określonych środowisk chmurowych, pozwalają na korzystanie z usług sieciowych w sposób bardziej efektywny, a także generujący mniejszą liczbę błędów.

Rozdział 6

Podsumowanie

6.1. Uzyskane efekty pracy

Celem niniejszej pracy była ewaluacja wydajności interfejsów programowania aplikacji implementowanych w dwóch różnych technologiach, w odniesieniu do licznych aspektów dotyczących sposobów ich wykorzystania. Zdecydowano się, nie tylko na analizę efektywności podstawowych rodzajów operacji protokołu hipertekstowego, wchodzących w skład tak popularnych dzisiaj usług sieciowych, ale także wykorzystania mechanizmów programowania współbieżnego, technik obsługi żądań asynchronicznych, implementacji zaawansowanego wzorca projektowego, zastosowania rozwiniętych technik optymalizacji pozyskiwania danych, a także wdrożenia rozwiązań w kontekście środowisk chmurowych.

Zwrócenie uwagi na tak wiele aspektów dotyczących interfejsów programowania aplikacji miało na celu uświadomienie czytelnika, że tego typu systemy internetowe, wykorzystywane są powszechnie nie tylko do realizacji najpopularniejszych czterech, podstawowych operacji na danych. Wachlarz możliwości związanych z tworzeniem internetowych API jest znacznie szerszy, a fakt ten uwydatnia się wraz z rosnącym poziomem skomplikowania usług sieciowych, a także zadań, które są przed nimi stawiane.

Zastosowanie mnogości kontekstów, w których odnaleźć musiały się były przygotowane rozwiązania, miało też odmienny cel. Misją autora było dowiedzenie się czy którykolwiek z systemów opartych o dwie porównywane technologie wdrożeniowo-uruchomieniowe, wykazuje wysoką wydajność względem swojego konkurenta w którymkolwiek z obszarów prowadzonych badań. Jeżeli tak, to które z tych obszarów są faworyzowane przez konkretne technologie.

Wyniki przeprowadzonych badań umożliwiły rozwianie powyższych wątpliwości, a także uzyskanie dodatkowej wiedzy, która nawet dla osób posiadających doświadczenie w zakresie kompozycji oraz tworzenia interfejsów API, nie musi wydawać się oczywista. Zrealizowane eksperymenty uwidocznili niektóre zależności, zadając innym z kolei kłam. Przykładem potwierdzenia spodziewanej hipotezy, może być wykazana wyższość wydajności rozwiązań implementowanych na platformach dedykowanych względem platform generycznych. Kolejną egzemplifikacją, w ramach której, jeszcze przed przeprowadzeniem badania, sformułować można było silną hipotezę, była obserwacja wpływu zastosowania usprawnień wydajnościowych, separacji środowisk bazodanowych, a także wdrożenia wzorca podziału odpowiedzialności. Rezultaty badania systemów bazodanowych z kolei, mogą być doskonałym argumentem, na obalenie hipotezy wyższej efektywności komunikacji silników baz danych oraz interfejsów tworzonych na bazie technologii jednego producenta.

Odnosząc się do dodatkowej wiedzy, której chęć pozyskiwania wzmożona została poprzez ambicję wyjaśnienia pojawiających się w badaniach anomalii, wspomnieć należy o sposobie obsługi wielowątkowej w odniesieniu do wspólnie generowanych, długotrwałych żądań.

Obsługa ta, niemalże nie występuje w kontekście interfejsu języka JavaScript, natomiast jest wydatnie rozbudowana w przypadku usługi implementowanej w C# i uruchamianej na platformie .NET. Ponadto, ciekawym jest również fakt, jak bardzo prostota, tycząca się mechanizmów wywoływania operacji asynchronicznych, może nieść korzyść dotyczącą wydajności ich realizacji.

W niektórych przypadkach jednak, konwencjonalność rozwiązania nie idzie w parze z jego wydajnością. Potwierdzeniem tego właśnie stwierdzenia są przeprowadzone badania dotyczące podstawowego oraz autorskiego podejścia do realizacji mechanizmów pamięci podręcznej. W ramach pracy tej, zaimplementowano, a także zbadano zachowanie systemu cache uwzględniającego częstotliwość wywoływania punktu końcowego, a także liczbę unieważnień identyfikującego go wpisu. Zgromadzone rezultaty należy postrzegać jako obiecujące, jednakże wymagana jest zdecydowanie bardziej obszerna analiza uwzględniająca zmienność liczby momentów unieważnień, czy też wpływ dysproporcji parametrów w różnych chwilach obsługi żądań.

Wspomnieć należy również o przeprowadzonych w ramach niniejszego badania parowych testach statystycznych, które pozwoliły na wykazanie statystycznie istotnej przewagi określonych konfiguracji rozwiązań względem pozostałych z nich.

Bardzo ważnym jest również uwypuklenie pewnej tezy. Przeprowadzony zestaw badań nie wskazał, jednakże przede wszystkim nie miał wskazać, technologii niezaprzecjalnie lepszej. Technologia taka nie istnieje, a wynika to w głównej mierze z ilości obszarów, w kontekście których może ona zostać wykorzystana oraz badana. Dlatego też, dokument ten, może okazać się pomocny dla tych osób, którzy zainteresowani są oceną poziomu wydajności interfejsu dla konkretnej technologii oraz konkretnego sposobu jej wykorzystania.

6.2. Perspektywy rozwoju badań

Każdy z przytoczonych obszarów wykorzystania interfejsów programowania aplikacji, reprezentowany w niniejszej pracy poprzez odmienne badanie, może zostać z łatwością rozbudowany poprzez ewaluację dodatkowych metryk wydajnościowych, czy też zmianę konfiguracji środowiska badawczego. Zdecydowano się na wskazanie perspektyw rozwoju badań w odniesieniu do tych domen funkcjonalności API, które zostały poruszone w tym dokumencie.

Odwołując się do badania wpływu wykorzystania systemów bazodanowych w kontekście porównywanych technologii, jako perspektywę rozwoju wskazać należy przeprowadzenie ewaluacji wydajności dla przedziałów liczby wątków-generatorów o zmiennej długości. Ponadto, wzięte pod uwagę mogą być również te spośród systemów bazodanowych, w ramach których nie dostarczane jest wsparcie dla mapperów obiektowo-relacyjnych Entity Framework Core oraz Prisma.

W ramach badania realizacji operacji współbieżnych, wprowadzić można dodatkowe rodzaje algorytmów metaheurystycznych dla różnych problemów o wysokiej złożoności obliczeniowej. Interesującą perspektywą rozwoju tego badania, jest również implementacja odmiennych heurystyk dla symetrycznego problemu komiwojażera, a także porównanie ich wykonania dla interfejsów wspierających przetwarzanie wielowątkowe.

Badanie wydajności operacji asynchronicznych może zostać poszerzone o uwzględnienie różnych implementacji klientów protokołu hipertekstowego, a także konfiguracji poszczególnych ich parametrów.

Najwięcej perspektyw rozwoju badań, wyróżnić należy w kontekście ewaluacji porównywanych mechanizmów pamięci podręcznej. Zbadane mogą zostać między innymi odmienne metryki wpływające na zmianę czasu odpowiedzi na żądanie. Jako przykładową metrykę wskazać można narzut wydajnościowy wprowadzany przez metodę kalkulacji czasu życia wpisu w magazynie pamięci podręcznej. Ponadto, struktura wykonanego badania, mogłaby zostać dostosowana względem zmiennego czasu trwania testu, zmiennego natężenia ruchu sieciowego, czy też

deterministycznego charakteru wywoływania żądań unieważniających. Co więcej, zaproponowana przez autora metoda może zostać zmodyfikowana poprzez uzależnienie czasu przechowywania wpisu od dodatkowych parametrów, bądź też zmianę ich istotności względem wyliczania czasu życia elementu pamięci cache.

W kontekście wprowadzenia wzorca podziału odpowiedzialności, a także separacji modeli bazodanowych, badanie może zostać poszerzone o zastosowanie odmiennych mechanizmów replikacji, niż wykorzystana w tej pracy technika transakcyjna. W takim przypadku, należy zbadać w jakim czasie, od momentu dodania rekordu bazodanowego, będzie on dostępny w ramach źródła danych przeznaczonego do odczytu. Co więcej, należy pamiętać, że celem zdefiniowania specyficznej struktury obsługi żądania, była możliwość optymalizacji modeli danych. Dlatego też, interesującą perspektywą rozwoju tego badania, mogłoby być wskazanie, które z zaimplementowanych technik optymalizacji mają kluczowe znaczenie pod kątem wydajności, a które wpływają na nią jedynie nieznacznie.

Ostatnie ze zidentyfikowanych perspektyw rozwoju tyczą się obserwacji efektywności działania w odniesieniu do produkcyjnych środowisk chmurowych. Biorąc pod uwagę ten właśnie aspekt, możliwym jest dokonanie porównania dla większej liczby platform wdrożeniowych, czy też zbadanie, w jaki sposób usługi generyczne typu infrastructure-as-a-service mogą zostać zoptymalizowane, aby implementowane wewnętrz nich systemy internetowe, osiągały wydajność przybliżoną, lub wyższą względem rozwiązań dedykowanych.

Literatura

- [1] A. Barth. Rfc 6265-http state management mechanism. *Internet Engineering Task Force (IETF)*, strony 2070–1721, 2011.
- [2] D. Bermbach, E. Wittern. Benchmarking web api quality. *International Conference on Web Engineering*, strony 188–206. Springer, 2016.
- [3] G. Bierman, M. Abadi, M. Torgersen. Understanding typescript. *European Conference on Object-Oriented Programming*, strony 257–281. Springer, 2014.
- [4] J. Bogard. Mediatr library documentation. <https://github.com/jbogard/MediatR/wiki>. Dostęp z dnia: 2022-03-27.
- [5] V. Bojinov. *RESTful Web API Design with Node.js 10: Learn to create robust RESTful web services with Node.js, MongoDB, and Express.js*. Packt Publishing Ltd, 2018.
- [6] M. Casciaro, L. Mammino. *Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques*. Packt Publishing Ltd, 2020.
- [7] Cisco. Cisco annual internet report (2018–2023) white paper. *Cisco: San Jose, CA, USA*, 2018.
- [8] R. Fielding, J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content, 2014.
- [9] R. T. Fielding, J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, Czerw. 2014.
- [10] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, S. Reddy. *HTTP: the definitive guide*. O'Reilly Media, Inc.", 2002.
- [11] E. H. Halili. *Apache JMeter*. Packt Publishing Birmingham, 2008.
- [12] A. Hejlsberg, S. Wiltamuth, P. Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [13] B. Hetzel. *The complete guide to software testing*. QED Information Sciences, Inc., 1988.
- [14] S. Holmes. *Mongoose for Application Development*. Packt Publishing Ltd, 2013.
- [15] T. Hyttinen. .net core 3.1 &. net 5: Performance benchmarking in web api use. *Bachelor's Thesis*, 2021.
- [16] International Organization for Standardization. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, 2011.
- [17] S. H. Jensen, A. Møller, P. Thiemann. Type analysis for javascript. *International Static Analysis Symposium*, strony 238–255. Springer, 2009.

-
- [18] C. Kambalyal. 3-tier architecture. *Retrieved On*, 2(34):2010, 2010.
 - [19] B. Lakshmiraghavan. *Pro Asp. Net Web API Security: Securing ASP. NET Web API*. Apress, 2013.
 - [20] D. Laksono. Testing spatial data deliverance in sql and nosql database using nodejs ful-lstack web app. *2018 4th International Conference on Science and Technology (ICST)*, strony 1–5. IEEE, 2018.
 - [21] A. Mardan. *Express. js Guide: The Comprehensive Book on Express. js*. Azat Mardan, 2014.
 - [22] E. Miller. Introduction to software testing technology. *Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalog No. EHO*, strony 180–0, 1981.
 - [23] G. J. Myers, C. Sandler, T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.
 - [24] A. Neumann, N. Laranjeiro, J. Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 2018.
 - [25] D. Nevedrov. Using jmeter to performance test web services. *Published on dev2dev*, strony 1–11, 2006.
 - [26] K. Prasad. *Software Testing Tools: Covering WinRunner, Silk Test, LoadRunner, JMeter and TestDirector with case studies w/CD*. Dreamtech press, 2004.
 - [27] G. e. a. Rauch. Prisma orm library documentation. <https://www.prisma.io/docs/reference>. Dostęp z dnia: 2022-03-27.
 - [28] G. Reinelt. TSPLIB—A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3(4):376–384, November 1991.
 - [29] A. Roman. *Testowanie i jakość oprogramowania: modele, techniki, narzędzia*. Warszawa: Wydawnictwo Naukowe PWN, 2015.
 - [30] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, X. Xu. Web services composition: A decade’s overview. *Information Sciences*, 280:218–238, 2014.
 - [31] A. Singjai, U. Zdun, O. Zimmermann, M. Stocker, C. Pautasso. Patterns on designing api endpoint operations. *28th Conference on Pattern Languages of Programs (PLoP)*, October 2021.
 - [32] J. P. Smith. *Entity Framework core in action*. Simon and Schuster, 2021.
 - [33] A. Spillner, T. Linz. *Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant*. dpunkt. verlag, 2021.
 - [34] H. Subramanian, P. Raj. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs*. Packt Publishing Ltd, 2019.
 - [35] A. Torres, R. Galante, M. S. Pimenta, A. J. B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology*, 82:1–18, 2017.
 - [36] A. Troelsen, P. Japikse. *Pro C# 7: With. net and. net Core*, wolumen 1328. Springer, 2017.
 - [37] H. Wang, J. Z. Huang, Y. Qu, J. Xie. Web services: problems and future directions. *Journal of Web Semantics*, 1(3):309–320, 2004.

- [38] J. Webber, S. Parastatidis, I. Robinson. *REST in practice: Hypermedia and systems architecture*. Ó'Reilly Media, Inc.", 2010.
- [39] M. Zacharuk. Przedstawienie i rozwiązywanie problemu testowania wydajności aplikacji internetowej opartej na rest api w środowisku rozproszonym. *Bachelor's Thesis*, 2020.