

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: SYSTEMY I SIECI KOMPUTEROWE

PRACA DYPLOMOWA
MAGISTERSKA

Analiza wydajnościowa interfejsów API w
technologiach C# oraz NodeJS

Performance analysis of C# and NodeJS
APIs

AUTOR:
Maciej Grzela

PROWADZĄCY PRACĘ:
dr inż., Michał Kucharzak
Katedra Systemów i Sieci Komputerowych

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	15
2.3. Testowanie usług sieciowych	21
2.4. Wykorzystywane technologie	21
2.5. Przegląd literatury	21
3. Opis problemu	28
4. Projekt i implementacja środowiska badawczego	29
4.1. Interfejs programowania aplikacji (API)	29
4.2. Uwierzytelnianie i autoryzacja użytkowników	32
4.3. Zarządzanie stanem aplikacji	34
4.3.1. Struktura mechanizmu zarządzania stanem zewnętrznym	34
4.3.2. Odświeżanie widoków w zależności od zmiany stanu	34
4.4. Aktualizacja treści „na żywo”	35
4.5. Aplikacja webowa	37
4.5.1. Przekazywanie dokumentów	37
4.5.2. Śledzenie dokumentów	40
4.5.3. Nadzorowanie dostawców	41
4.5.4. Powiadamianie klientów	45
4.5.5. Wyznaczanie trasy dostawy	47
4.5.6. Zarządzanie systemem	51
4.6. Aplikacja mobilna	54
4.6.1. Panel główny użytkownika	54
4.6.2. Wprowadzanie wpisów kilometrów	55
4.6.3. Wprowadzanie notatek handlowych	55
4.6.4. Cykliczna lokalizacja urządzenia	55
4.7. Skrypty archiwizujące	57
4.8. Interfejs użytkownika	58
5. Badania eksperymentalne	65
5.1. Uzyskane efekty pracy	65
5.2. Możliwości dalszego rozwoju	66

6. Podsumowanie	67
Literatura	68

Spis rysunków

2.1. Proces przetwarzania żądania wewnętrz interfejsu API	19
2.2. Proces uwierzytelnienia oraz autoryzacji użytkownika przed interfejsem API	20
2.3. Proces uwierzytelnienia oraz autoryzacji użytkownika przed interfejsem API	21
4.1. Widok panelu głównego aplikacji webowej	58
4.2. Widok panelu głównego aplikacji webowej (cz 2.)	58
4.3. Widok funkcjonalności przekazywania dokumentów pracownikowi	59
4.4. Widok funkcjonalności śledzenia dokumentów	59
4.5. Widok funkcjonalności przeglądania notatek pracowników handlowych	60
4.6. Widok funkcjonalności planowania trasy dostawy - wybór lokalizacji	60
4.7. Widok funkcjonalności planowania trasy dostawy - rezultat	61
4.8. Widok funkcjonalności komunikatora tekstowego	61
4.9. Aktywność główna aplikacji mobilnej a) lista dokumentów b) panel boczny	62
4.10. Aktywności wprowadzania danych a) zgłaszanie uwagi b) wpis kilometrów c) wpis notatki	62
4.11. Aktywności planowania trasy dostawy a) lista dokumentów b) wybór lokalizacji c) rezultat obliczeń	63
4.12. Aktywność komunikatora aplikacji mobilnej a) lista pracowników b) wymiana wiadomości	64

Spis listingów

4.1. Kod metody punktu końcowego przekazywania dokumentów pracownikowi	30
4.2. Kod metody serwisu przekazywania dokumentów pracownikowi	31
4.3. Kod metody repozytorium odpowiedzialnej za przekazywanie dokumentów pracownikowi	32
4.4. Kod metody punktu końcowego logowania pracownika	33
4.5. Kod metody klasy serwisu realizującego procedurę logowania	33
4.6. Kod metody inicjującej połączenie WebSocket dla komunikatora tekstowego	35
4.7. Kod metody wyboru uczestnika konwersacji tekstowej	36
4.8. Kod akcji utworzenia nowej wiadomości tekstowej	36
4.9. Kod metody serwera realizującej utworzenie nowej wiadomości tekstowej	37
4.10. Kod funkcji wykonywanej po zeskanowaniu barkodu pracownika	38
4.11. Kod funkcji wykonywanej po zeskanowaniu barkodu dokumentu	39
4.12. Kod funkcji przekazania dokumentów	40
4.13. Kod akcji nasłuchującej na zdarzenie dodania lokalizacji pracownika	41
4.14. Kod akcji pobierającej listę kilometrów dla pojazdów	42
4.15. Kod akcji pobierającej listę punktów lokalizacyjnych dla pracownika	43
4.16. Kod metody grupującej uzyskane punkty lokalizacyjne	43
4.17. Kod metody sortującej uzyskane punkty lokalizacyjne	43
4.18. Kod komponentu wyświetlającego mapę oraz znaczniki lokalizacji	44
4.19. Kod metody wykonującej funkcję typu „Fire And Forget”	45
4.20. Kod punktu końcowego odpowiedzialnego za wysyłanie wiadomości e-mail	46
4.21. Kod metody serwisu odpowiedzialnej za wysyłanie wiadomości e-mail	46
4.22. Kod metody odpowiedzialnej za wyznaczenie trasy dostawy dla zbioru dokumentów	48
4.23. Kod funkcji składowej wyboru populacji potomnej	48
4.24. Kod metody generującej zbiór osobników dominujących dla nowej populacji	49
4.25. Kod metody generującej zbiór pozostałych osobników dla nowej populacji	49
4.26. Kod metody generującej zbiór pozostałych osobników dla nowej populacji	50
4.27. Kod metody generującej zbiór pozostałych osobników dla nowej populacji	50
4.28. Kod funkcji składowej wykonującej operację mutacji osobnika	50
4.29. Kod metody repozytorium przygotowującej dane dla wykresu kilometrów	53
4.30. Kod metody pozwalającej na zarejestrowanie osobnego zadania	56
4.31. Kod wątku wykonującego zadanie lokalizacji z interwałem czasowym	56
4.32. Kod metody uzyskującej lokalizację urządzenia użytkownika	57

Spis tabel

2.1. Zbiór dozwolonych metod protokołu hipertekstowego	16
2.2. Zbiór najczęściej wykorzystywanych linii nagłówkowych w kontekście żądania protokołu hipertekstowego	17
2.3. Zbiór najczęściej zwracanych linii nagłówkowych w kontekście odpowiedzi protokołu hipertekstowego	18
2.4. Zbiór kodów statusu odpowiedzi protokołu hipertekstowego	18

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 Visual Networking Index). 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.

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.

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. API - 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 war-

stwy obsługi żądania, która zwraca ją określonymu 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.

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, obejmować ma zarówno aspekty dotyczące efektywności działania samego interfejsu programowania aplikacji, jak i rozwiązań wchodzących w skład tworzonego systemu. Wśród rozwiązań tych, wyróżnić należy mappery obiektowo-relacyjne, systemy bazodanowe, czy też mechanizmy zarządzania pamięcią podręczną. Niektóre spośród wymienionych elementów 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 będą w kontekście 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 tworzonych 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 kontekście 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 aspekcie realizacji badań, a także przegląd technologii informatycznych, wykorzystywanych w celu implementacji środowiska badawczego oraz wykonania pomiarów.

W ramach trzeciego z rozdziałów, zdefiniowano i omówiono każdy z aspektów problemu badawczego. Dzięki czemu, w kolejnej z sekcji pracy, możliwe było sformułowanie określonych scenariuszy badawczych.

Rozdział czwarty, stanowi o projekcie oraz implementacji środowiska badawczego, wykorzystawanego w celu dokonywania pomiarów...

Piąty z rozdziałów, ma na celu przedstawienie rezultatów wykonywanych badań. Rezultaty te, w ramach niniejszego rozdziału zostały zgrupowane względem zdefiniowanych uprzednio scenariuszy badawczych. Ponadto, dla uzyskanych pomiarów wykonano testy parametryczne, dzięki którym możliwa jest ocena istotności statystycznej zaobserwowanych różnic wynikowych. W ramach tego rozdziału, dokonano także analizy uzyskanych wyników.

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 (API) - OPIS Z INŻ.

Application Programming Interface

Zbiór zasad oraz procedur determinujący sposób komunikacji pomiędzy wieloma aplikacjami. Aplikacjami tymi mogą być zarówno programy klienckie (np. strona webowa), jak i serwery danych.

API wykonane w technologii REST - OPIS Z INŻ.

RESTful API

Interfejs programowania aplikacji opierający swoją budowę oraz sposób funkcjonowania o zbiór ustalonych reguł. Reguły te, dotyczą między innymi: struktury żądań wysyłanych od klienta do serwera, budowy zasobu odpowiedzi serwera, a także kodów statusów zwracanych z chwilą odpowiedzi w zależności od wykonanej akcji.

Kontroler - OPIS Z INŻ.

Controller

Klasa, której zadaniem jest obsłuzenie żądania aplikacji klienckiej, weryfikacja jego poprawności, a następnie wywołanie kodu logiki biznesowej w ramach struktur serwisów. Po otrzyman-

niu rezultatu obliczeń z warstwy logiki biznesowej, odpowiedzialnością metody kontrolera jest zwrócenie przetworzonego zasobu do systemu klienta.

Serwis - OPIS Z INŻ.

Service

Klasa, zawierająca metody odpowiedzialne za realizację logiki biznesowej w ramach interfejsu programowania aplikacji. Obiekt tej struktury danych, wywoływany jest bezpośrednio przed metody klas kontrolerów.

Repozytorium - OPIS Z INŻ.

Repository

Struktura danych, wykorzystywana do komunikacji interfejsu API z serwerem bazodanowym. Metody, w ramach klas repozytoriów, operują na modelu danych, przechowywanym w ramach API, a następnie, odwzorowują ten model za pomocą narzędzia ORM, na fizyczną zawartość bazodanową.

Mapper obiektowo-relacyjny (ORM)

Object-relational mapper (ORM)

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.

Wielowątkowość - OPIS Z INŻ.

Multithreading

Technika programowania, zakładająca wykorzystanie wielu odrębnie wykonywanych procesów w ramach jednej aplikacji. W przypadku interfejsu API implementującego tę technikę, każdy z punktów końcowych stanowi osobny wątek, będący częścią składową pojedynczego procesu. Dzięki temu, aplikacja jest dostępna, niezależnie od wywołanych, niekiedy długo trwających zadań.

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 usługi sieci Web - OPIS Z INŻ.

Endpoint

Metoda klasy kontrolera, uruchamiana w momencie określenia żądania klienta. Do każdego z punktów końcowych, przypisany jest adres wywołania, zbiór wymaganych parametrów, a także obsługiwany typ metody protokołu hipertekstowego. Dzięki temu, bazując na strukturze otrzymanego żądania, interfejs API jest w stanie stwierdzić który z punktów końcowych powinien zostać wywołany.

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 dopuszczalnych 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 zwrocanego 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 (JSON) - OPIS Z INŻ.

JavaScript Object Notation

Niezależny od języka programowania format prezentacji, definicji oraz wymiany danych w formie obiektów. Powszechnie stosowany jako sposób generowania ciała żądania wysyłanego do interfejsu API, a także odpowiedzi od niego uzyskiwanej.

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 obciążeniowe

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 tekstową określającą ciało żądania (*ang. Request body*). Taki zbiór informacji, pozwala na jednoznaczna 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.
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 wykorzystywanym 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.

Po wykonaniu kodu programu przypisanego do określonego rodzaju polecenia generowanego przez aplikacje 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 ustrukturzowana 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 charakterystyce 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 dokumentów RFC (*ang. Request For Comments*) o numerach 7230 do 7237, powinno ono posiadać charakter informacji pomocniczej, bądź też pozostać puste. 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 zauważać, że w ramach systemu tego, występuje zjawisko inercji, powodowane koniecznością realizacji zdefiniowanego w ramach API kodu programu. Na podstawie tego założenia, ewaluację działania oraz

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 mniejszych 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
cookie	Zbiór informacji pozwalających na wprowadzenie oraz utrzymanie stanowego charakteru transmisji	Zestaw par klucz-wartość, gdzie klucz jest wartością tekową, 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

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 niemiejszej 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 - wysłosowane 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

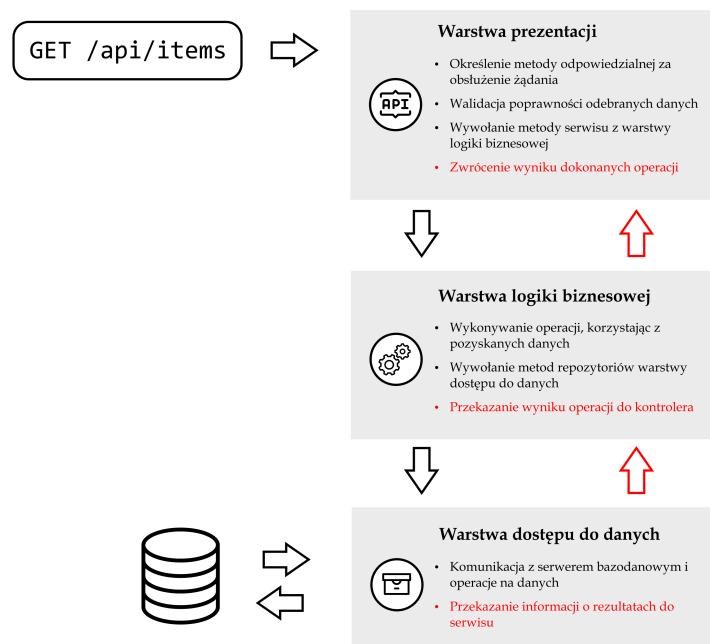
wydajności interfejsu programowania aplikacji przeprowadzić można poprzez wprowadzanie określonego wejścia (tj. generowanie żądania) oraz obserwację wartości zwróconej na wyjściu (tj. uzyskana odpowiedź).

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, złożone 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. Na ilustracji 2.1 przedstawiono przepływ informacji wewnątrz interfejsu API, od momentu wygenerowania żądania do chwili uzyskania odpowiedzi.



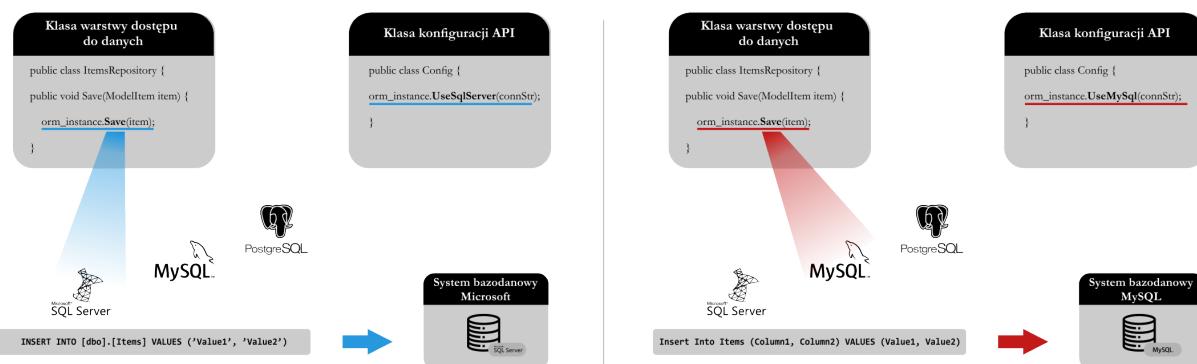
Rys. 2.1: Proces przetwarzania żądania wewnątrz interfejsu API

Mapowanie obiektowo-relacyjne oraz podejście Code-First

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 rzutowane 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.

Uwierzytelnienie oraz autoryzacja

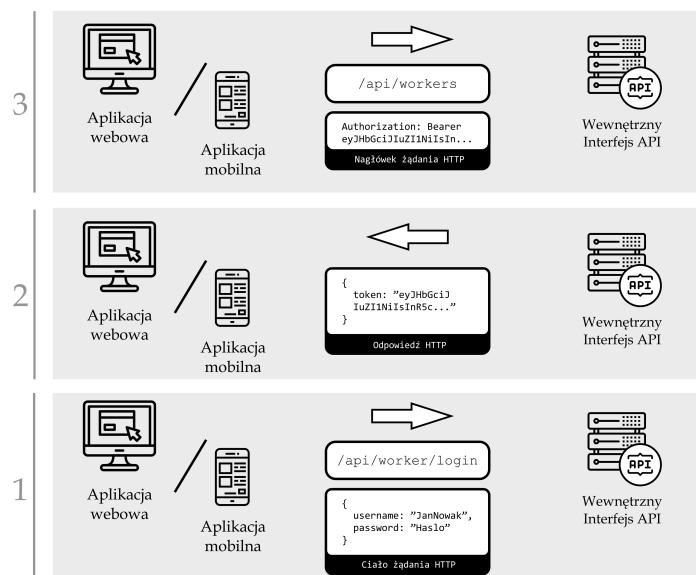
Proces uwierzytelnienia oraz autoryzacji użytkownika odwołującego się do interfejsu programowania aplikacji, przedstawić należy w trzech następujących krokach.

Pierwszym z nich, jest wygenerowanie żądania odwołującego się do punktu końcowego odpowiedzialnego za obsługę uwierzytelnienia wewnątrz API. Żądanie to, musi posiadać ciało, zawierające informacje poświadczające o konkretnymi 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, zwrocona 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.

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



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

Segregacja zapytań oraz komend w kontekście odwołań do źródła danych

Konwencja REST

2.3. Testowanie usług sieciowych

2.4. Wykorzystywane technologie

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 [1] 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 [2]. W ramach tej pracy zawarto obszerne wprowadzenie do platformy NodeJS uwzględniające ponadto kwestie obsługi operacji wejścia/wyjścia, wykonywania natwnego 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 uwzględniają metodologię REST, zawarte zostały w pozycji literaturowej [3]. 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 rozwiąza niem ORM jest biblioteka Entity Framework Core, która przedstawiona została w pozycji [4]. 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 przetwarzać 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 przeciwieństwie do platformy .NET, zastosować możemy zdecydowanie większą liczbę bibliotek pełniących rolę maperów obiektowo-relacyjnych. Biblioteki te, zostały opisane w pozycjach [5] i [6]. Pozycja [5] pełni rolę całosciowego wprowadzenia do tematyki tworzenia interfejsów API, korzystając z platformy NodeJS, frameworka ExpressJS oraz nierelacyjnej bazy danych MongoDB. Rodział 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 [6] 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 [7] 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 [8]. 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 zuniifikowanych 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 [9] 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 oprogramowania. Uzgłaębiono 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), dostarczając 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 [10] 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, uzgłaębiono 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 [11], 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 [12]. 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 validacji 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 [13]. Pozycja ta, stanowi normę międzynarodowej organizacji normalizacyjnej (ang. International Organization for Standardization) dotyczącą weryfikacji jakości oprogramowania. Uzgłaębiono 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 [14] 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 [15] oraz [16] przedstawiono pełen zakres funkcjonalności dostępnych w ramach narzędzia Apache JMeter. Pierwsza z prac (tj. [15]), skupia się na wykorzystaniu narzędzia w celu wykonywania testów wydajności usług sieciowych, natomiast druga z pozycji (tj. [16]), 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 [15], 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 [16] 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 [17] 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, narysowano pojęcie reprezentacji danych przesyłanych za pomocą protokołu 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 żądanie, grupując je w sposób semantyczny. Dla każdego z przedstawionych kodów statusu narysowano 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 [18] pozwala na poszerzenie wiedzy dotyczącej protokołu hipertekstowego w bardziej praktycznym kontekście. Podobnie jak w dokumencie [17], 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 [18] 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 połączniowa, 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 [19]. 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 literaturowych, 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 [20] 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 [21]. 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

Rozdział 4

Projekt i implementacja środowiska badawczego

W niniejszym rozdziale opisana została implementacja kluczowych funkcjonalności systemu. W ramach opisu, przedstawiono elementy kodu źródłowego programu wraz z ich wyjaśnieniami.

W pierwszych dwóch podrozdziałach, omówiona została zasada działania interfejsu programowania aplikacji. Uzakazano w nich między innymi pełny przepływ wykonywanych operacji w momencie wywołania punktu końcowego API.

W podrozdziale trzecim, przeanalizowano koncepcję mechanizmu zarządzania stanem aplikacji oraz aktualizacji treści bez przeładowywania strony. Koncepcje ta, są określone w sposób identyczny, zarówno dla aplikacji mobilnej jak i webowej.

Następne części odnoszą się kolejno do funkcjonalności aplikacji webowej oraz mobilnej. Niektóre spośród tych funkcjonalności są wspólne dla obu aplikacji, dlatego też przedstawione zostaną tylko w kontekście danego elementu systemu.

Ostatni z podrozdziałów stanowi o implementacji graficznego interfejsu użytkownika. Poza kodem źródłowym widoków systemu, zaprezentowane są w nim również ilustracje przedstawiające gotowy interfejs.

4.1. Interfejs programowania aplikacji (API)

Zgodnie z informacjami wprowadzonymi w podrozdziale ??, interfejs programowania aplikacji API, zbudowany został w oparciu o architekturę trójwarstwową. W strukturze tej, po wysłaniu żądania z wykorzystaniem protokołu hipertekstowego, zostaje ono obsłużone przez określoną metodę klasy kontrolera. Metoda ta, w celu ustalenia odpowiedzi na żądanie klienta, komunikować się musi z klasami niższych warstw modelu, określonymi jako serwisy.

Wewnątrz klas serwisów, wykonywane są operacje z dziedziny logiki biznesowej. W operacjach tych, wymagane są najczęściej dane, które metoda serwisu mogłaby przetwarzanie. Aby te dane uzyskać, podobnie jak metody kontrolerów, tak i metody warstwy logiki biznesowej odwoływały się muszą do klas warstw niższych. Takie klasy, w kontekście zrealizowanego oprogramowania nazywane są repozytoriami.

Mechanizm odwołań do klas warstw niższych w celu pozyskiwania niezbędnych do przetwarzania informacji umożliwia proste rozdzielenie odpowiedzialności w ramach realizowanych funkcji. Ponadto, pozwala on na implementację elastycznego mechanizmu obsługi błędów oraz łatwiejsze wykrywanie potencjalnych problemów.

Na listingach 4.1, 4.2 i 4.3, określone zostały fragmenty kodu źródłowego metod, dla klas z poszczególnych warstw architektury aplikacji. Wszystkie z listingów są ze sobą powiązanie, ponieważ stanowią fragmenty realizacji odpowiedzi na to samo żądanie klienta.

Poniżej przedstawiono sposób obsługi żądania typu POST dla punktu końcowego `/api/documents/forwards/worker`, odpowiedzialnego za realizację przekazania dokumentu pracownikowi.

Po wysłaniu żądania HTTP, dla określonego w akapicie powyżej adresu punktu końcowego, wykonany zostanie kod metody z listingu 4.1. W pierwszym wierszu tej metody, określona została adnotacja (*ang. Data Annotation*) dotycząca typu obsługiwanej żądania, oraz fragmentu jego adresu. Metoda jest asynchroniczna (tj. uruchamiana jest jako oddzielny wątek programu i cechuje się sekwencyjnością realizacji operacji niezależnie od czasu ich trwania).

Funkcja punktu końcowego przyjmuje jako argument zasób (*ang. Resource*). Element ten, definiuje parametry związane z nadawcą, odbiorcą oraz przekazywanym dokumentem. Wartość zasobu, można przekazać metodzie endpoint'u poprzez zdefiniowanie ciała żądania HTTP (*ang. HTTP Body*). W przypadku realizowanego API, ciało żądania posiada formę obiektu JSON.

Na początku metody, przekazany argument (zasób) poddawany jest walidacji. Walidacja może przebiec niepomyślnie w sytuacji, w której elementy obiektu JSON będą posiadały niepoprawne typy danych, lub wymagane wartości nie zostaną w nim zdefiniowane. Po pomyślnym przejściu procedury walidacji, obiekt zasobu oraz zagnieżdżony w nim zasób dokumentu odwzorowywane są na obiekty klas modelu danych. Pozwala to, na przekazanie ich jako parametry do metody klasy warstwy niższej - serwisu.

Po uzyskaniu wartości zwrotnej przez serwis, w metodzie kontrolera sprawdzana jest jej poprawność. W zależności od wyniku tego sprawdzenia, wysyłana jest odpowiedź HTTP sygnalizująca błąd wykonywanej operacji lub jej powodzenie. Zgodnie ze standardem REST (*ang. Representational State Transfer*), wykorzystywanym przez system jako zbiór zasad dotyczących działania API, w przypadku poprawności metody punktu końcowego dla typu operacji POST, zwieracana jest pusta zawartość (*ang. No Content*).

```

1 [HttpPost("worker")]
2 public async Task<IActionResult> ForwardToWorkerAsync([FromBody]
3     ↪ ForwardToWorkerResource res) {
4
5     if (!ModelState.IsValid) {
6         return BadRequest(ModelState.GetModelErrorMessagesInfo());
7     }
8
9     var documentsForward = mapper.Map<ForwardToWorkerResource, DocumentsForward>(res);
10    var document = mapper.Map<ForwardDocumentResource, ForwardDocument>(res.Document);
11    var result = await documentsForwardService.ForwardToWorker(documentsForward,
12        ↪ document);
13
14    if (!result.Success)
15        return BadRequest(result.Message);
16
17    return NoContent();
18 }
```

Listing 4.1: Kod metody punktu końcowego przekazywania dokumentów pracownikowi

W linii dziewiątej listingu 4.1 metoda kontrolera API wywołuje metodę serwisu. Linia ta, jest kluczowym fragmentem kodu źródłowego punktu końcowego, ponieważ właśnie w metodzie serwisu wykonywane są operacje związane z faktycznym przekazaniem dokumentu pracownikowi. Omawiana w następnym akapicie funkcja serwisu przedstawiona jest na listingu 4.2.

Do metody przekazania dokumentu, dostarczane są dwa argumenty. Pierwszym z nich jest obiekt zasobu, natomiast drugi stanowi zasób faktury. Elementy te, zgodnie z informacjami przedstawionymi uprzednio, zostały wcześniej przekształcone z obiektowej notacji JavaScript na klasy modelu danych.

Początkowo, weryfikowane jest istnienie dokumentu w systemie na podstawie jego identyfikatora. Jeżeli dany rachunek nie został znaleziony, jest on generowany w oparciu o dane zawarte w zasobie dokumentu.

Następnie, weryfikowane są informacje związane z nadawcą oraz adresatem faktury. Dla prawidłowego przekazania dokumentu, dane obu pracowników (tj. ich identyfikatory) muszą być prawidłowe. W sytuacji wykrycia nieprawidłowości dla danych o pracownikach, metoda serwisu zwraca sterowanie do metody kontrolera, przekazując nowy obiekt odpowiedzi (*ang. Response*). Gdy w obiekcie tym, przypisana zostaje właściwość wiadomości błędu (*ang. Message*), logiczny atrybut powodzenia wykonania funkcji (*ang. Success*) przyjmuje wartość `false`. Pozwala to, na określenie poprawności realizacji zadania serwisu wewnątrz metody kontrolera.

Kolejno, pobierane są wszystkie przekazania przypisane do dokumentu. Jeżeli ich liczba, jest większa od zera, należy sprawdzić, czy osoba chcącą przekazać dokument, aktualnie znajduje się w jego posiadaniu. W tym celu, uzyskiwany zostaje ostatni element listy przekazań. Aby zweryfikować przynależność dokumentu do pracownika, identyfikator odbiorcy w ostatnim z przekazań musi być zgodny z numerem pracownika stanowiącego nadawcę dla nowego przekazania. W przeciwnym razie, generowany jest obiekt odpowiedzi i przekazywane zostaje sterowanie do metody kontrolera.

W ostatnim fragmencie kodu źródłowego funkcji serwisu, definiowany jest nowy obiekt przekazania dokumentu. Po jego utworzeniu, wywoływana zostaje metoda klasy repozytorium zapisująca przekazanie do kontekstu bazy danych. Aby zmiana danych w obrębie kontekstu, odwzorowana została w bazie danych wykonana musi zostać metoda `CommitTransactionAsync` dla instancji klasy `UnitOfWork`. Klasa ta, jest implementacją wzorca projektowego *Jednostka pracy*, opisanego w rozdziale ??.

Instrukcją kończącą metodę serwisu jest zwrócenie nowego obiektu odpowiedzi. W tym przypadku, obiekt ten zamiast tekstem wiadomości błędu, jest inicjalizowany strukturą przekazania dokumentu. Dzięki temu, atrybut poprawności wykonania metody przyjmuje wartość `true`.

```

1 public async Task<Response<DocumentsForward>> ForwardToWorker(DocumentsForward for
    ↪ ward, ForwardDocument document) {
2
3     var existingDocument = await documentService.GetDocumentAsync(document.Id);
4     Document documentToForward = null;
5     if (!existingDocument.Success) {
6         var result = await documentService.SaveAsync(document);
7
8         if (!result.Success) {
9             return new Response<DocumentsForward>("Błąd zapisu dokumentu:{...}");
10        }
11        documentToForward = result.Type;
12    }
13    else {
14        documentToForward = existingDocument.Type;
15    }
16
17    var existingRemitter = await workerService.GetWorkerAsync(forward.RemitterId);
18    var existingRecipient = await workerService.GetWorkerAsync(forward.RecipientId);
19
20    if (!existingRemitter.Success)
21        return new Response<DocumentsForward>("Nadawca o id:{...} nie został
    ↪ znaleziony");
22
23    if (!existingRecipient.Success)
24        return new Response<DocumentsForward>("Odbiorca o id:{...} nie został
    ↪ znaleziony");

```

```

25
26     var forwardsForDocument = documentToForward.DocumentsForwards.ToList();
27     if (forwardsForDocument.Count != 0) {
28         var lastForward = forwardsForDocument.Last();
29         if (lastForward.RecipientId != existingRemitter.Type.Id)
30             return new Response<DocumentsForward>("Nie jesteś właścicielem dokumentu!"
31             );
32     var forwardToSave = new DocumentsForward {
33         Remitter = existingRemitter.Type,
34         Recipient = existingRecipient.Type,
35         Document = documentToForward,
36         Confirmed = true,
37         Created = DateTime.Now
38     };
39     await documentsForwardRepository.SaveAsync(forwardToSave);
40     await unitOfWork.CommitTransactionAsync();
41     ...
42     return new Response<DocumentsForward>(forward);

```

Listing 4.2: Kod metody serwisu przekazywania dokumentów pracownikowi

W kodzie źródłowym metody serwisu, występuje odwołanie do funkcji klasy repozytorium. Funkcje tej klasy, mają za zadanie operować na kontekście bazy danych (tj. odwzorowaniu danych z bazy w kodzie programu).

Na listingu 4.3 pokazane zostało ciało metody repozytorium. Element ten, odpowiedzialny jest za zapisanie do kontekstu bazodanowego nowego przekazania dokumentu.

Po wykonaniu wszystkich operacji, podobnie jak metody serwisów, tak i metoda repozytorium zwraca sterowanie do funkcji wywołująccej.

```

1 public async Task SaveAsync(DocumentsForward documentsForward) {
2     await context.DocumentsForwards.AddAsync(documentsForward);
3 }

```

Listing 4.3: Kod metody repozytorium odpowiedzialnej za przekazywanie dokumentów pracownikowi

4.2. Uwierzytelnianie i autoryzacja użytkowników

Mechanizm uwierzytelniania i autoryzacji użytkowników systemu skonstruowany został w oparciu o technologie .Net Core Identity oraz JSON Web Token. W ramach pierwszego z tych rozwiązań, udostępniane są metody obsługi użytkowników w systemie oraz mechanizmy tworzenia tokenu logowania. Drugie rozwiązanie, określa sposób autoryzacji użytkownika oraz format tokenu.

Pierwszą fazą procesu jest wysłanie przez klienta żądania HTTP pod adres punktu końcowego odpowiedzialnego za logowanie. W żądaniu tym, określone musi zostać jego ciało, zawierające dane uwierzytelniające.

Jeżeli dane te są poprawne, interfejs API w odpowiedzi zwraca wygenerowany token użytkownika.

Klient (tj. aplikacja webowa lub mobilna) wywołuje punkty końcowe API, załączając token autoryzujący w nagłówku komunikatu protokołu hipertekstowego. Token ten, będzie autoryzowany przez interfejs tak długo, jak długo będzie trwać jego ważność. Dla zrealizowanego oprogramowania czas ważności tokenu określony został na jeden dzień.

W poniższych akapitach, przedstawiona została implementacja omówionego procesu w ramach interfejsu programowania aplikacji. Na listingu 4.4 ukazano kod źródłowy metody Login klasy `WorkersController`, która pełni rolę punktu końcowego dla logowania w systemie.

Podobnie jak w przypadku punktu końcowego realizującego przekazania dokumentów, przed ciałem metody określony jest rodzaj żądania wywołującego oraz nazwa endpointu. Ponadto, dodana została adnotacja `AllowAnonymous`, informująca o możliwości wywołania punktu końcowego bez uprzedniej autoryzacji.

W ramach metody kontrolera, początkowo sprawdzana jest poprawność przekazanego jako parametr zasobu. Zasób ten, zawiera dane, za pomocą których użytkownik chce się uwierzytelnić. Aby dane te mogły być przetwarzane, muszą zostać odwzorowywane na instancję klasy modelu danych interfejsu API.

```

1 [AllowAnonymous]
2 [HttpPost("login")]
3 public async Task<ActionResult> Login([FromBody] LoginCredentialsResource login
    ↪ Resource) {
4     if (!ModelState.IsValid)
5         return BadRequest(ModelState.GetModelStateErrorMessagesInfo());
6
7     var loginCredential = mapper.Map<LoginCredentialsResource, LoginCredential>(login
    ↪ Resource);
8     var result = await workerService.Login(loginCredential);
9
10    if (!result.Success) return Unauthorized(result.Message);
11
12    var resource = mapper.Map<LoggedWorker, LoggedWorkerResource>(result.Type);
13    return Ok(resource);
14 }
```

Listing 4.4: Kod metody punktu końcowego logowania pracownika

Następnym krokiem jest odwołanie się do metody klasy serwisu, realizującej faktyczną procedurę uwierzytelniania. Kod źródłowy tej metody pokazany został na listingu 4.5. W zależności od wyniku zwróconego przez część składową serwisu, użytkownikowi przekazywany jest określony typ odpowiedzi HTTP.

Metoda klasy serwisu, obsługująca procedurę logowania przyjmuje jako parametr obiekt informacji uwierzytelniających. Na jego podstawie, z wykorzystaniem klasy `UserManager` dostępnej w ramach biblioteki .Net Core Identity, sprawdzane jest istnienie użytkownika oraz zgodność przekazanych przez niego danych.

W przypadku poprawności dostarczonych przez użytkownika informacji, tworzony zostaje obiekt parametrów pracownika, będący wartością zwracaną z metody serwisu. Obiekt ten, poza danymi pracownika, posiada także wygenerowany token autoryzujący.

Po odwzorowaniu zwróconej struktury na klasę zasobu, jest ona przekazywana jako odpowiedź na żądanie klienta.

```

1 public async Task<Response<LoggedWorker>> Login(LoginCredential loginCredential) {
2     var worker = await workerManager.Users
3         .Include(p => p.Vehicle)
4         .Include(p => p.JobPosition)
5         .Include(p => p.WorkersPrivileges)
6         .ThenInclude(p => p.Privilege)
7         .SingleOrDefaultAsync(w => w.UserName == loginCredential.UserName);
8
9     if (worker == null)
10        return new Response<LoggedWorker>("Pracownik: [...] nie został znaleziony");
11
12    var result = await signInManager.CheckPasswordSignInAsync(worker, loginCredential
    ↪ .Password, false);
13
14    if (result.Succeeded) {
15        var loggedWorker = new LoggedWorker {
```

```

16     Id = worker.Id,
17     FirstName = worker.FirstName,
18     LastName = worker.LastName,
19     ...
20     Token = jwtGenerator.CreateToken(worker)
21   };
22   return new Response<LoggedWorker>(loggedWorker);
23 }
24 return new Response<LoggedWorker>("Dane uwierzytelniające są niepoprawne");
25 }
```

Listing 4.5: Kod metody klasy serwisu realizującego procedurę logowania

4.3. Zarządzanie stanem aplikacji

Pojęcie systemu zarządzania stanem odnosi się do części klienckiej przygotowanego oprogramowania (tj. aplikacji mobilnej oraz webowej). W przypadku obu z tych elementów, wykorzystany został hybrydowy model przechowywania stanu aplikacji, wykorzystujący zarówno koncepcję stanu wewnętrznego poszczególnych komponentów, jak i stanu zewnętrznego, wspólnego dla wszystkich elementów aplikacji. Obiekty zawarte wewnętrznych stanach komponentów, wykorzystywane są do realizacji operacji unikalnych dla każdego z elementów składowych aplikacji. W ramach stanu zewnętrznego natomiast, przechowywane informacje są przetwarzane przez wiele niezależnych od siebie komponentów.

Zarówno dla aplikacji webowej, jak i mobilnej, jako system zarządzania stanem (tj. stanem zewnętrznym) wybrana została biblioteka MobX.

4.3.1. Struktura mechanizmu zarządzania stanem zewnętrznym

W zakresie stanu zewnętrznego, zdefiniowana została klasa głównego magazynu (*ang. RootStore*), dla której atrybutami są instancje wszystkich pozostałych magazynów podległych. Klasy podległe natomiast, mają za zadanie przechowywać elementy stanu.

Utworzono magazyny podległe, zawierają w sobie fragmenty stanu, odpowiadające wybranym funkcjonalnościami aplikacji. Dla przykładu, w ramach klasy *WorkersStore* przechowywana jest między innymi lista wszystkich pracowników, a także wszystkie notatki pracownicze. Z kolei w magazynie *AlgorithmsStore* gromadzone są informacje odnośnie wyniku planowania trasy dostawy.

Aby dane z klas-magazynów mogły zostać wykorzystane w dowolnym komponencie, cała struktura klas musi stać się elementem ogólnodostępnym w ramach aplikacji. Wymaganie to, jest realizowane poprzez wykorzystanie mechanizmu kontekstu. Mechanizm ten, reprezentuje strukturę danych osiągalną dla każdego z fragmentów aplikacji. Do struktury tej, dodawana jest nowa instancja klasy *RootStore*.

4.3.2. Odświeżanie widoków w zależności od zmiany stanu

Podstawowym celem modyfikacji danych w obrębie klasy magazynu jest aktualizacja widoku aplikacji. W ramach omawianych elementów systemu proces odświeżania widoków przebiega następująco:

- Wywołanie funkcji w ramach określonego komponentu.
- Odwołanie się funkcji komponentu do akcji konkretnego magazynu.
- Modyfikacja właściwości obserwowej, będącej elementem stanu, wewnątrz akcji magazynu.

- Opcjonalne ustalenie wartości dla powiązanej z elementem obserwowały, właściwości wyliczanej.
- Odświeżenie widoku wewnętrz komponentu.

4.4. Aktualizacja treści „na żywo”

W obszarze funkcjonalności śledzenia dokumentów oraz komunikatora tekstowego zaimplementowany został mechanizm aktualizacji treści „na żywo”. W ramach jego działania, użytkownik jest na bieżąco informowany o nowych wiadomościach przychodzących czy też zmianach statusu dokumentów.

Do realizacji opisywanego modułu wykorzystany został protokół WebSockets. Protokół ten, pozwala na dwukierunkową komunikację pomiędzy serwerem a klientem (tj. przeglądarką lub aplikacją mobilną). W przeciwieństwie do rozwiązań takich jak odpytywanie HTTP (*ang. HTTP Polling*), WebSockets umożliwia wysyłanie wiadomości z serwera, bez uprzedniego otrzymania żądania klienta. Zastosowanie tego protokołu, dostarcza możliwość informowania klienta o zachodzących zmianach w modelu danych, nad którymi kontrolę sprawuje serwer.

W zrealizowanym oprogramowaniu, wykorzystana została biblioteka firmy Microsoft, obsługująca komunikację dla protokołu WebSockets. Biblioteka ta, nazwana SignalR dostarcza zbiór metod, pozwalających na kontrolę transmisji danych, zarówno po stronie serwera jak i klienta.

Komunikację WebSockets, w ramach stworzonego systemu, można opisać za pomocą następującej sekwencji:

- Aplikacja kliencka inicjuje połączenie z wykorzystaniem protokołu gniazd sieciowych, wysyłając określone żądanie do serwera.
- Serwer zestawia połączenie z aplikacją przeglądarkową.
- Aplikacja, w czasie realizacji określonej funkcjonalności, wywołuje metodę serwera która ma zostać wykonana.
- Serwer wykonuje kod wywołanej metody, a następnie powiadamia o tym fakcie, określony zbiór lub wszystkich klientów aktualnie z nim połączonych.
- Każdy z klientów nasłuchuje na odpowiedź z serwera, związaną z wykonaniem określonego zadania. Jeśli ta odpowiedź nadaje się, aplikacje klienckie wykonują ustalony kod programu.

W celu zaprezentowania sposobu działania mechanizmu aktualizacji treści z wykorzystaniem protokołu gniazd sieciowych, przedstawiona została implementacja funkcjonalności wewnętrzne komunikatora tekstowego. Funkcjonalność tą, zrealizowano w sposób analogiczny, dla aplikacji webowej oraz mobilnej. W niniejszym podrozdziale, zaprezentowany zostanie kod źródłowy aplikacji webowej.

Aby skorzystać z komunikatora tekstowego, należy przejść do określonego widoku aplikacji. Uruchomienie tego widoku, powoduje wywołanie akcji `createHubConnection` z klasy magazynu `MessagesStore`. Zadaniem tej akcji, jest ustanowienie połączenia z wykorzystaniem protokołu WebSockets pomiędzy aplikacją a serwerem. Na listingu 4.6 zaprezentowano kod źródłowy omawianej funkcji.

W celu ustanowienia komunikacji tworzony jest obiekt `HubConnection`, z wykorzystaniem wzorca budowniczego oraz klasy `HubConnectionBuilder`. W ramach tego obiektu, określany zostaje adres serwera (przechowywany w zmiennej środowiskowej aplikacji), token autoryzujący, typ wyświetlanych komunikatów informacyjnych oraz opcja automatycznego ponawiania połączenia. Następnie, wykonywana jest metoda `start`, wysyłająca żądanie inicjujące do serwera. Wyjątki, związane z niepowodzeniem ustanowienia wymiany danych, obsługiwane są poprzez funkcję `catch`.

```
1 @action createHubConnection = () => {
2   this.hubConnection = new HubConnectionBuilder()
```

```

3   .withUrl(process.env.REACT_APP_HUB_URL, {
4     accessTokenFactory: () => window.localStorage.getItem('jwt'),
5   }).configureLogging(LogLevel.None)
6   .withAutomaticReconnect()
7   .build();
8
9 this.hubConnection.start()
10 .catch(() =>
11   toast.error('Utracono połączenie z powiadomieniami na żywo. Odśwież stronę w
12   celu ponownego połączenia')
13 );

```

Listing 4.6: Kod metody inicjującej połączenie WebSocket dla komunikatora tekstowego

Po zestawieniu połączenia pomiędzy elementami systemu, użytkownik wybiera uczestnika konwersacji. Wybór ten, skutkuje wykonaniem metody `prepareConversation` przedstawionej na listingu 4.7.

Parametr procedury, identyfikowany jako `worker` określa zbiór danych dotyczących wybranego uczestnika rozmowy.

Aby wymieniane wiadomości, widoczne były tylko w obszarze odbiorcy oraz nadawcy, w czasie każdej z konwersacji utworzona musi zostać dwuosobowa grupa uczestników rozmowy. Rozwiążanie to, pozwala serwerowi ustalić, do których klientów powinien on wysłać informację o wystąpieniu zdarzenia wysłania wiadomości.

Dla każdej z tworzonych grup, zdefiniowany musi zostać identyfikator, znany zarówno przez nadawcę jak i odbiorcę wiadomości. W związku z faktem, że obaj użytkownicy posiadają, wewnątrz metody `prepareConversation` dane drugiego z uczestników rozmowy, identyfikator grupy określony został jako sortowane leksykograficznie złączenie kodów identyfikacyjnych dwóch pracowników.

Z chwilą ustalenia nazwy grupy, wywoływane są metody przyporządkowujące wyznaczone wartości do właściwości obserwacyjnych. Kolejno, realizowane jest wykonanie procedury serwera z wykorzystaniem komunikacji WebSockets. Następnie, pobierana zostaje lista wszystkich wiadomości pomiędzy dwoma użytkownikami oraz ustawienie ich statusu jako „Przeczytane”. Operacje te, determinują aktualizację widoku aplikacji klienta.

```

1 const prepareConversation = (worker) => {
2   let idsOrder = user.id.localeCompare(worker.id);
3   let groupName = idsOrder === 1 ? `${user.id}${worker.id}` : `${worker.id}${user.id}`;
4   messagesStore.selectConversationUsers(user, worker);
5   messagesStore.selectGroupName(groupName);
6   messagesStore.loadMessagesBetween(worker.id, user.id);
7   setMessagesRead(worker);
8 };

```

Listing 4.7: Kod metody wyboru uczestnika konwersacji tekstowej

Po połączeniu użytkowników w grupy, dokonywane może zostać faktyczne przekazywanie wiadomości pomiędzy nimi. Na listingu 4.8 przedstawiona została akcja klasy magazynu odpowiedzialna za utworzenie nowej wiadomości.

Początkowo, konstruowany jest obiekt zasobu, przechowujący informacje o nowym elemencie konwersacji. Następnie, wywoływana zostaje zdalna metoda serwera, obsługująca funkcjonalność tworzenia wiadomości. Jeżeli w trakcie jej wykonywania wystąpi błąd, wyświetlony zostanie komunikat informujący użytkownika o tym fakcie.

```

1 @action addMessage = async (message) => {
2   let resource = {
3     fromWorkerId: this.conversationSelectedUser.id,
4     toWorkerId: this.conversationSelectedRemoteUser.id,

```

```

5   message: `'{message.message} @{message.mention}`'
6   received: false,
7 };
8
9 await this.hubConnection
10 .invoke('SaveConversation', resource, this.groupName)
11 .catch(() =>
12   toast.error('Nie udało się wysłać wiadomości. Spróbuj ponownie później'));
13 ...
14 };

```

Listing 4.8: Kod akcji utworzenia nowej wiadomości tekstowej

Zadaniem zdalnej metody `SaveConversation`, do której odwołuje się funkcja z listingu 4.8 jest wprowadzenie do bazy danych nowej wiadomości. W dalszej kolejności, po wykonaniu tego zadania, klienci muszą zostać o tym fakcie poinformowani. Dlatego też, wewnątrz wspomnianej procedury, następuje wysłanie komunikatów z serwera do klientów. Komunikaty wysyłane są zarówno do zdefiniowanej uprzednio grupy (wymusza to odświeżenie listy wszystkich wiadomości), jak i do pozostałych odbiorców. Dzięki temu, odbiorca nie korzystający aktualnie z modułu wiadomości będzie powiadomiony o pojawienniu się nowego fragmentu konwersacji. Kod procedury `SaveConversation` ukazany został na listingu 4.9.

```

1 public async Task SaveConversation(SaveConversationItemResource resource, string
  ↪ group) {
2   var item = mapper.Map<SaveConversationItemResource, ConversationItem>(resource);
3   var msgItem = await conversationItemService.SaveAsync(item);
4   if (!msgItem.Success)
5   {
6     throw new HttpRequestException(msgItem.Message);
7   }
8
9   var response = mapper.Map<ConversationItem, ConversationItemResource>(msgItem.
  ↪ Type);
10
11 await Clients.Group(group).SendAsync("ReceiveConversationItem", response);
12 await Clients.All.SendAsync("ReceiveConversationNotification", response);
13 }

```

Listing 4.9: Kod metody serwera realizującej utworzenie nowej wiadomości tekstowej

4.5. Aplikacja webowa

4.5.1. Przekazywanie dokumentów

Moduł przekazywania dokumentów stanowi kluczową funkcjonalność przygotowanego oprogramowania. W jego obszarze, wyróżnić można następujące elementy: przekazanie dokumentu pracownikowi, przekazanie dokumentu klientowi oraz zwrócenie dokumentu do działu sprzedaży.

Dla pierwszego typu przekazań, pracownik-nadawca, z wykorzystaniem obrazu kamery internetowej lub ręcznego czytnika kodów, skanuje barkod uwierzytelniający pracownika-odbiorcy. Następnie sczytywane są kody wszystkich dokumentów, które mają zostać przekazane. Aby operacja skanowania przebiegła pomyślnie, struktury QR, zarówno pracownika jak i dokumentów muszą posiadać odpowiednie formaty danych.

Pracownik przekazujący dokumenty, po uwierzytelnieniu nadawcy, posiada określony w konfiguracji systemu czas, w którym zeskanować może przekazywane faktury. Domyślną wartością

tego czasu jest 30 sekund, jednakże liczba ta może być modyfikowana przez osobę posiadającą uprawnienia do zmiany danych konfiguracyjnych.

Ponadto, zdefiniowane zostały określone formaty numerów dokumentów. Jeżeli numer skanowanego dokumentu nie pasuje do żadnego ze schematów, nie zostanie on przekazany pracownikowi.

Drugi element omawianego modułu, stanowi funkcjonalność przekazywania dokumentów klientowi. Zasada realizacji tej funkcjonalności, oraz wszystkie ograniczenia z nią związane są analogiczne do elementu przekazywania dokumentu pracownikowi. Istotną różnicą, pomiędzy tymi funkcjonalnościami, jest brak konieczności wyboru pracownika w trakcie procesu przekazywania faktur.

Struktury danych, definiujące pojedyncze przekazanie dokumentu są identyczne dla obu funkcjonalności. W przypadku gdy odbiorcą faktury jest klient, jego identyfikator wewnątrz tej struktury, przybiera wartość nieokreślona null.

Trzeci z wymienionych rodzajów transferów dokumentów, czyli zwrot dokumentów do działu sprzedaży, jest modyfikacją pierwszego z typów przekazań. Stanowisko działu sprzedaży jest traktowane w systemie w sposób odrębny, w przeciwieństwie do pozostałych rodzajów funkcji pracowników. Charakteryzuje się ono, możliwością przypisania do niego tylko jednego pracownika.

Dlatego też, element zwrotu dokumentu do działu sprzedaży jest w rzeczywistości funkcjonalnością przekazania dokumentu pracownikowi, w którym odbiorca faktury jest z góry określony.

W niniejszym rozdziale przedstawiona zostanie implementacja funkcji transferu dokumentów między pracownikami.

Na listingu 4.10 zilustrowany został kod źródłowy operacji, wykonywanej w momencie zescanowania barkodu QR pracownika.

Dane zawarte wewnątrz barkodu reprezentowane są w funkcji jako parametr data. Pierwszą operacją, jest sprawdzenie, czy dane te nie posiadają wartości nieokreślonej. Następnie, dokonywana jest konwersja informacji o pracowniku na format obiektowej notacji JavaScript. Pozwala to, na wykorzystanie tych informacji jako parametr żądania API. W ramach odpowiedzi interfejsu uzyskiwany jest obiekt parametrów pracownika. Dzięki temu, wiadomym jest że odbiorca jest faktycznym użytkownikiem systemu.

W dalszej kolejności, dokonywane jest porównanie identyfikatorów pracowników, w celu wyeliminowania możliwości przekazania faktury samemu sobie. Jeżeli numery identyfikacyjne aktualnie zalogowanego użytkownika oraz odbiorcy są różne, następuje określenie odbiorcy, a także zmiana wartości zmiennych stanu wewnętrznego determinujących aktywność ustalonych fragmentów widoku.

Wszystkie z opisanych operacji, zawarte są wewnątrz bloku zgłaszania wyjątków, dlatego też, w sytuacji wystąpienia niespodziewanego błędu, użytkownik zostanie o tym fakcie poinformowany.

```

1 const handleScannedWorker = (data) => {
2   if (data !== null) {
3     try {
4       let json = JSON.parse(data);
5       httpClient.Workers.listOneByUserName(json.userName);
6       .then((response) => {
7         if (response.id === user.id) {
8           setCameraOneMessage('Nie można przekazać sobie dokumentu');
9         } else {
10           setChosenWorker(response);
11           setCameraOneScanned(true);
12           setCameraOneEnabled(false);
13           setCameraOneMessage('Zeskanowano kod pracownika...');


```

```

14         setActiveForm(1);
15     }
16   })
17   .catch(() => {
18     toast.error('Brak pracownika o podanym loginie');
19   });
20 } catch (e) {
21   toast.error('Wystąpił błąd skanowania');
22 }
23 }
24 };

```

Listing 4.10: Kod funkcji wykonywanej po zeskanowaniu barkodu pracownika

Po pomyślnym zweryfikowaniu danych odbiorcy, wykonywane zostaje skanowanie dokumentów. Na listingu 4.11 przedstawiony został kod metody, uruchamianej po sczytaniu danych faktury.

Podobnie jak przy weryfikacji informacji o pracowniku, sprawdzana jest zawartość danych pod kontem wartości nieokreślonej, a także wykonywana jest konwersja na format JSON. Następnie, identyfikowany zostaje parametr „s”, oznaczający w formacie kodu, numer dokumentu. Stanowi to rodzaj weryfikacji danych faktury po stronie klienta. W kolejnym kroku, aktualnie przechowywany zbiór dokumentów do przekazania jest analizowany pod kontem występowania w nim aktualnie sczytanej faktury. Jeżeli faktury tej nie ma w zbiorze, może ona zostać do niego dodana. W przeciwnym razie, wyświetlony zostaje stosowny komunikat.

```

1 const handleScannedInvoice = (data) => {
2   if (data !== null) {
3     try {
4       let json = JSON.parse(data);
5       if (json.hasOwnProperty('s')) {
6         let documents = scannedDocuments;
7         let found = documents.filter((d) => d.s === json.s);
8         if (found.length === 0) {
9           setScannedDocuments([...documents, json]);
10        } else {
11          setCameraTwoMessage('Dokument został już zeskanowany');
12          setTimeout(() => {
13            setCameraTwoMessage('Zeskanuj kod przekazywanego dokumentu');
14            }, 1000);
15        }
16      } else {
17        setCameraTwoMessage('Kod QR nie odnosi się do dokumentu');
18      }
19    } catch (e) {
20      setCameraTwoMessage('Struktura kodu jest niepoprawna');
21    }
22  }
23 };

```

Listing 4.11: Kod funkcji wykonywanej po zeskanowaniu barkodu dokumentu

Następnym krokiem, po określeniu niezbędnych do realizacji przekazania informacji, jest wprowadzenie do bazy danych wpisów definiujących transfery faktur. Na listingu 4.12, pokazany został kod funkcji, odpowiedzialnej za przygotowanie danych oraz wywołanie żądania interfejsu API. Interfejs ten, w ramach jednego z punktów końcowych, wykona operację wprowadzenia wpisów przekazań dokumentów.

Pierwszym elementem funkcji z listingu 4.12, jest zdefiniowanie zasobu dokumentu na podstawie danych barkodu. Następnie, dokument ten staje się elementem obiektu przekazania, który pełni rolę zasobu nadziedzennego. Kolejno, wykonywana jest metoda `forwardToWorker`, której

zadaniem jest wygenerowanie żądania do interfejsu API. Wszystkie z tych operacji, przeprowadzane są wewnątrz asynchronicznej pętli `forEach`. Pozwala to na zachowanie kolejności wprowadzania przekazań, oraz ustalenie sekwencyjności długo trwających operacji. Po zakończeniu pętli, ładowane są wszystkie wiadomości dotyczące ewentualnych błędów związanych z przekazaniami.

```

1 const handleDocumentsPass = async () => {
2   await forEachAsync(scannedDocuments, async (scan) => {
3     let document = {
4       id: scan.s,
5       clientSymbol: scan.c !== undefined ? scan.c : null,
6       deliveryPointSymbol: scan.dp !== undefined ? scan.dp : null,
7       paymentMethodSymbol: scan.pm !== undefined ? scan.pm : null,
8       grossPrice: scan.pr !== undefined ? parseFloat(scan.pr) : null,
9       paymentTimeout: scan.ti !== undefined ? Number(scan.ti) : null,
10    };
11
12    let resource = {
13      remitterId: user.id,
14      recipientId: chosenWorker.id,
15      document: document,
16      comment: null,
17      confirmed: true,
18    };
19
20    await forwardToWorker(resource);
21  });
22  loadErrorMessage();
23  setErrorMessagesLoaded(true);
24 };

```

Listing 4.12: Kod funkcji przekazania dokumentów

Przedstawiony powyżej kod, jest ostatnim elementem modułu transferu dokumentów w obszarze aplikacji klienckiej. W odniesieniu do dziedziny interfejsu API, fragment procedury przekazania dokumentu przedstawiony został szczegółowo w podrozdziale 4.1.

4.5.2. Śledzenie dokumentów

Ideą modułu śledzenia dokumentów jest możliwość dostarczenia informacji o stanie faktur, zarówno tych aktualnie przekazywanych, jak i archiwalnych. Opisywany fragment oprogramowania zawiera dwie funkcjonalności (tj. przeglądanie dokumentów oraz przeglądanie archiwum).

W ramach przeglądu dokumentów, użytkownikowi ukazywany jest widok tabelaryczny, w którym kolejne z wierszy tabeli reprezentują poszczególne faktury. Dla każdego z poświadczonych sprzedży, określony jest jego aktualny stan. Dzięki temu rozwiązaniu, pracownik jest w stanie ocenić, czy dokument znajduje się u któregoś ze współpracowników, czy też został przekazany klientowi.

Ponadto, w widoku zawarte są informacje charakteryzujące fakturę (tj. symbol, dane kontrahenta oraz sposób płatności). Co więcej, opisywany element interfejsu dostarcza funkcjonalność wprowadzenia oraz przeglądania uwag i wyjaśnień związanych z dokumentem.

Dane, zawarte wewnątrz modułu przeglądania dokumentów, aktualizowane są bez odświeżania widoku, dzięki implementacji komunikacji z wykorzystaniem protokołu gniazd sieciowych.

Funkcjonalność przeglądania dokumentów archiwalnych, ma za zadanie zapewnić możliwość rozstrzygania kwestii spornych, dotyczących faktu dostarczenia poświadczania sprzedaży do klienta.

Wszystkie faktury, których stan określony jest jako „Przekazano klientowi”, w dniu następnym wprowadzone zostają do archiwum.

W ramach widoku archiwum, dostępne są te same dane, co w przypadku przeglądania aktualnie przekazywanych dokumentów.

4.5.3. Nadzorowanie dostawców

Moduł nadzorowania dostawców zawiera w sobie funkcjonalności: monitorowania trasy dostawcy, przeglądania wpisów kilometrów, przeglądania historii lokalizacji pracowników oraz przeglądania notatek handlowych.

Funkcjonalność monitorowania trasy dostawcy pozwala:

- Badać aktualną lokalizację danego pracownika
- Obserwować status dokumentów do niego przypisanych
- Prowadzić z nim konwersację tekstową
- Przeglądać jego statystyki efektywności.

W momencie przejścia do widoku funkcjonalności, pobierana zostaje z serwera lista wszystkich pracowników firmy. Na jej podstawie, generowana jest wartość właściwości wyliczanej, reprezentująca wszystkich użytkowników za wyjątkiem zalogowanego pracownika. Ponadto, następuje inicjacja połączenia z serwerem z wykorzystaniem protokołu WebSockets. Następnie, wartość wyliczana wykorzystywana jest jako zbiór danych dla listy wyboru, za pomocą której użytkownik określa, który z pracowników ma być nadzorowany.

Wybór pracownika z listy, determinuje przypisanie obiektu z jego danymi do wartości obserwowej, pobranie jego najnowszej pozycji lokalizacyjnej, powiązanych z nim dokumentów, a także statystyk efektywności. Każda z tych operacji, polega na wysłaniu konkretnego żądania do interfejsu API.

Po ustaleniu lokalizacji pracownika za pomocą aplikacji mobilnej, wywoływana zostaje metoda serwera, której jedynym zadaniem jest powiadomienie wszystkich połączonych klientów o wprowadzeniu do bazy danych nowego wpisu lokalizacji. Na takie zdarzenie, reaguje komponent aplikacji webowej, pobierając za pomocą żądania API, najnowszą lokalizację pracownika i wprowadzając ją do wartości obserwowej. Skutkuje to, aktualizacją fragmentów widoku przedstawiających adres aktualnej lokalizacji oraz ilustrację mapy. Na listingu 4.13 przedstawiono omówioną powyżej metodę nasłuchującą na zdarzenie wprowadzenia danych lokalizacyjnych.

```

1 this.hub.on('ReceiveAddLocation', async () => {
2   try {
3     let location = await httpClient.LocationPoints.getCurrentLocationForWorker(this.
4       ↪ selectedWorkerToInspect.id);
4     if (location !== '') {
5       this.inspectedWorkersLocation = location;
6     }
7     this.inspectedWorkersLocationLoaded = true;
8     return true;
9   }catch (error) {
10     return false;
11   }
12 });

```

Listing 4.13: Kod akcji nasłuchującej na zdarzenie dodania lokalizacji pracownika

W sposób analogiczny, opisać można zasadę aktualizacji danych wewnątrz bloku przypisanych dokumentów. W tym przypadku, użytkownik za pomocą aplikacji mobilnej w panelu głównym, lub za pomocą aplikacji webowej, w widoku przekazania dokumentów, wysyła do serwera żądanie wykonania metody informującej o zmianie stanu dokumentów. Metoda ta, przekazuje

informacje wszystkim pozostałym klientom, a w ramach nasłuchującej akcji, wykonywane jest pobranie aktualnych danych związanych z przypisanymi do pracownika dokumentami.

Fragment widoku prezentujący statystyki efektywności pracownika, jest elementem, który nie jest aktualizowany z wykorzystaniem protokołu gniazd sieciowych. Wynika to z faktu, że dane przedstawiane w tym fragmencie generowane są jednorazowo, po zakończeniu pracy przedsiębiorstwa, przez co w momencie przejścia do opisywanego widoku, są zawsze aktualne.

Następną funkcjonalnością modułu nadzoru dostawców, jest przeglądanie listy wpisów kilometrów. Widok tej funkcjonalności ma formę tabelaryczną i przedstawia wprowadzone wartości liczników kilometrów dla określonych pojazdów, z wyszczególnieniem ich kierowcy oraz daty utworzenia wpisu.

Z chwilą wybrania widoku, wysyłane jest żądanie pobrania wszystkich adnotacji dotyczących kilometrów pojazdów. Następnie, wartości te nadpisują, uprzedni stan właściwości obserwowalnej `kilometers`. Właściwość ta, zawarta wewnątrz widoku, determinuje jego odświeżenie. Następnie, edytowany jest fragment stanu, odpowiedzialny za wyświetlanie widoku ładowania treści. Jeżeli któraś z opisanych operacji, zakończy się niepowodzeniem, użytkownik jest o tym fakcie informowany poprzez komunikat w formie powiadomienia.

Na listingu 4.14 przedstawiony został kod źródłowy, akcji pobierania wpisów kilometrów dla pojazdów.

```

1 @action loadKilometers = async () => {
2   this.rootStore.loadingsStore.setFetchKilometersLoading(true);
3   try {
4     let kilometers = await httpClient.Kilometers.list();
5     this.kilometers = kilometers;
6     this.rootStore.loadingsStore.setFetchKilometersLoading(false);
7   } catch (error) {
8     toast.error('Nie udało się pobrać wpisów kilometrów');
9   }
10 };

```

Listing 4.14: Kod akcji pobierającej listę kilometrów dla pojazdów

Kolejnymi funkcjonalnościami modułu są widoki pobierania historii lokalizacji oraz notatek handlowych. Oba z tych widoków zostały wykonane w sposób analogiczny, ponieważ przedstawiają one znacząco pokrewne treści. W przypadku historii lokalizacji, pokazywane są punkty lokalizacyjne dotyczące pracowników, natomiast dla notatek handlowych, punkty te, rozszerzone są o treść notatki oraz jej typ.

Po wejściu do widoku przeglądu historii lokalizacji, sprawdzana zostaje wartość właściwości obserwowalnej `workers`, reprezentującej listę wszystkich pracowników. Jeżeli wartość ta, jest zero-elementową tablicą, oznacza to, że zbiór pracowników nigdy wcześniej nie został pobrany, więc należy to uczynić w tym momencie. W przeciwnym razie, procedura uzyskiwania listy użytkowników z interfejsu API jest pomijana.

Następnie, użytkownik wybiera jednego spośród pracowników, dla którego chce analizować historię wpisów lokalizacji. Skutkuje to, wywołaniem akcji magazynu, pobierającej, grupującej oraz sortującej dane adnotacji lokalizacyjnych.

Na początku, w akcji tej, sprawdzana jest zawartość zmiennej identyfikującej wybranego pracownika. Jeżeli zmienna, nie jest nieokreślona, następuje wysłanie żądania API dotyczącego wszystkich punktów lokalizacji pracownika. Po otrzymaniu tych punktów, są one grupowane po właściwości `date` (tj. dacie wprowadzenia), a następnie sortowane względem parametru `created`. Parametr ten, poza datą przechowuje także czas utworzenia wpisu lokalizacyjnego.

Przetworzone informacje są następnie wprowadzane do właściwości obserwowalnej, co determinuje wyświetlenie zbioru dat dotyczących wpisów pod elementem listy wskazującym na wybranego pracownika w widoku komponentu. Modyfikowana jest także właściwość wskazująca na wyświetlanie animacji ładowania wpisów.

Niepowodzenie procedury wykonania żądania jest obsługiwane wewnątrz bloku `catch` i wiąże się z wprowadzeniem do właściwości obserwowej punktów lokalizacyjnych wartości nieokreślonej.

Na listingu 4.15 ukazano procedurę pobrania punktów lokalizacyjnych.

```

1 @action loadLocationPoints = () => {
2   if (this.selectedWorker === null) {
3     this.locationPoints = null;
4   } else {
5     this.rootStore.loadingsStore.setFetchPointsInLocationHistoryLoading(true);
6     httpClient.LocationPoints.getForWorker(this.selectedWorker.id)
7       .then(async (response) => {
8         let grouped = await this.groupLocationPoints(response, 'date');
9         let sorted = await this.sortLocationPoints(grouped, 'created');
10        this.locationPoints = sorted;
11        this.chosenLocationPoints = null;
12        this.rootStore.loadingsStore.setFetchPointsInLocationHistoryLoading(false);
13      }).catch((error) => {
14        this.locationPoints = null;
15      });
16    }
17 };

```

Listing 4.15: Kod akcji pobierającej listę punktów lokalizacyjnych dla pracownika

W przypadku przedstawionej powyżej funkcji, należy wspomnieć o metodach grupowania oraz sortowania uzyskanych danych. W przypadku pierwszej z metod, wykorzystana została funkcja `reduce`, która na podstawie tablicy generuje pojedynczy obiekt danych. Jej pierwszym parametrem jest wywołanie zwrotne wykonywane przy każdej iteracji po tablicy, drugi parametr natomiast, to wartość inicjująca docelowy obiekt. Wynik funkcji, czyli wyliczana, zgodnie z określonym schematem wartość, jest przypisywany do zmiennej która następnie zostaje zwrocona.

Na listingu 4.16 pokazana została omówiona metoda grupowania danych.

```

1 groupLocationPoints = async (points, by) => {
2   let result = points.reduce((point, attribute) => {
3     point[attribute[by]] = point[attribute[by]] || [];
4     point[attribute[by]].push(attribute);
5     return point;
6   }, {});
7
8   return result;
9 };

```

Listing 4.16: Kod metody grupującej uzyskane punkty lokalizacyjne

Metoda sortowania danych lokalizacyjnych, opiera się na iteracji po grupowanym uprzednio obiekcie. Dla każdego z przebiegów pętli, w pomocniczej tablicy, zainicjowanej początkowo tablicą nieposortowaną, nadpisywane są wartości, po wywołaniu funkcji `sort`. Na koniec, pomocnicza tablica jest zwracana.

Funkcję sortowania punktów lokalizacyjnych przedstawiono na listingu 4.17.

```

1 sortLocationPoints = async (points, by) => {
2   let sortedLocations = points;
3   for (const [keyIndex, location] of Object.entries(sortedPoints)) {
4     sortedLocations[keyIndex] = location.sort((first, second) => first[by].
5       ↪ localeCompare(second[by]));
6   }
7   return sortedLocations;

```

```
8 };
```

Listing 4.17: Kod metody sortującej uzyskane punkty lokalizacyjne

Po przetworzeniu danych oraz wprowadzeniu ich do zmiennej obserwowej, generowane są dwie właściwości wyliczane. Pierwsza z nich reprezentuje te punkty lokalizacyjne, które zostały uzyskane w wyniku cyklicznego pobierania lokalizacji. Druga natomiast, przedstawia punkty wygenerowane w momencie przekazania dokumentów. Wyświetlanie konkretnego typu wpisów lokalizacji, jest uzależnione od zaznaczenia odpowiedniej opcji w widoku funkcjonalności.

Określony typ wpisów, ukazywany zostaje w komponencie mapy, jako zbiór interaktywnych znaczników. Wybranie dowolnego znacznika, powoduje pokazanie się małego okienka informacyjnego, zawierającego wszystkie dane konkretnego punktu lokalizacji.

Na listingu 4.18, zilustrowano fragment widoku komponentu, odpowiedzialnego za wyświetlanie mapy oraz znaczników lokalizacji.

```
1 <Map
2   center={[configuration.companyLatitude, configuration.companyLongitude]}>
3 ...
4 {togglerValue === true ? (
5 {chosenLocationPointsByLocation.map(
6   (point, index) => (
7     <>
8       <Marker icon={<Blue number={index + 1} />} position={[point.latitude, point.
9         ↪ longitude]}>
10      <Popup>
11        <List size='big'>
12          <List.Item>
13            <List.Icon name='clock' />
14            <List.Content>
15              {moment(point.created).format(
16                'HH:mm:ss'
17              )}
18            </List.Content>
19          </List.Item>
20          <List.Item>
21            <List.Icon name='map' />
22            <List.Content>
23              {point.address !== null
24                ? point.address
25                : 'Brak informacji'}
26            </List.Content>
27          </List.Item>
28          <List.Icon name='marker' />
29          <List.Content>
30            {'({point.latitude}, {point.longitude})'}
31          </List.Content>
32        </List.Item>
33        <List.Item>
34          <List.Icon name='forward' />
35          <List.Content>
36            {point.duringForward === true
37              ? 'TAK'
38              : 'NIE'}
39          </List.Content>
40        </List.Item>
41      </List>
42    </Popup>
43  </Marker>
44  ...
```

45)}

Listing 4.18: Kod komponentu wyświetlającego mapę oraz znaczniki lokalizacji

4.5.4. Powiadamianie klientów

W ramach funkcjonalności powiadamiania klienta, użytkownik wybiera dokumenty, dla których, przypisani do nich kontrahenci dostaną informację e-mailową, dotyczącą aktualnego statusu faktury.

W obrębie omawianego modułu, zdefiniowane zostały dwa szablony wiadomości e-mail. Zadaniem pierwszego z nich, jest poinformowanie klienta o przekazaniu dokumentu jednemu z pracowników wewnętrz firm. Wiąże się to z zaprezentowaniem kontrahentowi komunikatu „Twoje zamówienie jest w trakcie realizacji”. Za pomocą wiadomości opartej na drugim z szablonów, klient zostaje zawiadomiony o niezwłocznej wizycie dostawcy towaru.

Wiadomość dotycząca kroku pierwszego procedury dostarczania towaru, wysyłana jest automatycznie do klienta w chwili pierwszego przekazania dokumentu między pracownikami. Warunkiem powiadomienia kontrahenta, jest dostępność jego danych kontaktowych w bazie klientów.

Informacja, związana z drugim krokiem dostawy, wysłana może zostać za pośrednictwem dedykowanego widoku aplikacji webowej, lub poprzez naciśnięcie konkretnego przycisku w panelu głównym aplikacji mobilnej.

Do przekazywania wiadomości e-mail z wykorzystaniem interfejsu API, użyta została biblioteka MailKit. Ponadto, w związku z relatywnie długim czasem trwania wysyłania wiadomości, w porównaniu do uzyskiwania odpowiedzi odnośnie danych, metoda dostarczania e-mail jest wykonywana zgodnie z zasadą „wywołaj i zapomnij” (*ang. Fire And Forget*). Rozwiążanie to, pozwala na redukcję czasu odpowiedzi punktu końcowego API.

Na listingu 4.19 przedstawiona została metoda generycznej klasy `FireForgetHandlerService`, odpowiedzialna za wykonanie funkcji dowolnego serwisu, zgodnie z techniką „wywołaj i zapomnij”.

Jako parametr metoda `ExecuteeffTask` przyjmuje funkcję dowolnej klasy zwracającą typ `Task` (tj. funkcję asynchroniczną). Wewnątrz ciała funkcji składowej, tworzone jest nowe asynchroniczne zadanie. W zadaniu tym, wygenerowany zostaje zakres wywoływanego serwisów (*ang. scope*). Jest to możliwe, dzięki zastosowaniu instancji klasy `ServiceScopeFactory`, której abstrakcja zostaje dostarczona z wykorzystaniem mechanizmu wstrzykiwania zależności. W związku z użyciem dyrektywy `using`, zakres jest usuwany od razu po zakończeniu bloku instrukcji. Następnie, tworzona jest zmienna, przechowująca obiekt wywoływanego serwisu. Ostatnią instrukcją metody, jest uruchomienie wywołania zwrotnego, przekazanego jako parametr, dołączając jako jego atrybut instancję serwisu.

Jeżeli w trakcie którejś z opisanych operacji wystąpi nieoczekiwany błąd, sterowanie zostanie przekazane do bloku `catch`. Skutkowało to będzie wyświetleniem komunikatu niepowodzenia.

```

1 public void ExecuteeffTask(Func<T, Task> someLongRunningServiceTask) {
2     Task.Run(async () => {
3         try {
4             using var scope = serviceScopeFactory.CreateScope();
5             var service = scope.ServiceProvider.GetRequiredService<T>();
6             await someLongRunningServiceTask(service);
7         } catch (Exception e) {
8             Console.WriteLine(e);
9         }
10    });
11 }
```

Listing 4.19: Kod metody wykonującej funkcję typu „Fire And Forget”

Posiadając tak przygotowaną metodę, możliwe jest skorzystanie z niej, w kontekście wywołania serwisu wiadomości e-mail. Poczyniono tak wewnątrz klasy kontrolera `EmailsController`. Na listingu 4.20 pokazany został kod źródłowy metody serwisu odpowiedzialnej za wysyłanie wiadomości za pośrednictwem poczty elektronicznej.

Głównym fragmentem funkcji składowej jest wykonanie omówionej w poprzednich akapitach metody `ExecuteeffTask`. Metoda ta, została uruchomiona na rzecz obiektu klasy `FireForgetHandlerService`, która powiązana jest z klasą kontrolera `EmailsController` poprzez wykorzystanie mechanizmu wstrzykiwania zależności. Zasadniczym celem przedstawianej procedury, jest odwołanie się do funkcji `SendEmailAsync` serwisu `EmailService`.

```

1 public IActionResult SendAsync ([FromBody] SendClientEmailResource resource) {
2     if (!ModelState.IsValid) {
3         return BadRequest(ModelState.GetModelStateErrorMessagesInfo());
4     }
5     fireForgetHandlerService.Execute(async service => {
6         var clientEmail = mapper.Map<SendClientEmailResource, ClientEmail>(resource);
7         var result = await service.SendEmailAsync(clientEmail);
8         if (!result.Success) Console.WriteLine(result.Message);
9     });
10    return NoContent();
11 }
```

Listing 4.20: Kod punktu końcowego odpowiedzialnego za wysyłanie wiadomości e-mail

Funkcja `SendEmailAsync`, na początku ma za zadanie sprawdzić istnienie określonego dokumentu. Następnie weryfikowane zostają dane kontaktowe przypisane do miejsca dostawy. Jeżeli dane te, są nieokreślone, wykorzystywane są informacje adresowe kontrahenta.

Posiadawszy adres odbiorcy, tworzony zostaje tymczasowy obiekt klasy `Message`, przechowujący wszystkie elementy składowe wiadomości e-mail. Kolejno, jest on przekazywany jako parametr funkcji `CreateEmailMessage`, gdzie zostaje wykorzystany przy budowie obiektu `MimeMessage`. Obiekt ten, poza atrybutami struktury pomocniczej, przechowuje zawartość wysłanej wiadomości. Zawartość, wczytana zostaje z pliku `html`, a następnie w miejscu znaczników szablonu wprowadzane są określone wartości dotyczące kontrahenta.

Tak przygotowany obiekt wiadomości, staje się atrybutem metody `SendAsync`, odpowiadającej za realizację faktycznego wysłania wiadomości. Na początku tej metody, tworzona jest zmienna, przechowująca instancję klasy `SmtpClient`. Odpowiada ona za obsługę żądań w ramach klienta protokołu poczty wychodzącej. Następnie, inicjowane jest połączenie z serwerem pocztowym. Jest ono możliwe dzięki podaniu informacji uwierzytelniających, zapisanych w konfiguracji interfejsu API. W kolejnym kroku, wysyłana jest wiadomość (tj. funkcji wysyłania wiadomości przekazywany zostaje obiekt klasy `MimeMessage`). Po wysłaniu wiadomości, następuje zakończenie komunikacji z serwerem oraz zwrocenie nowej instancji wiadomości, jako sygnalizacja poprawności wykonania metody.

Na listingu 4.21, zilustrowano omówioną powyżej metodę wysyłania wiadomości e-mail.

```

1 private async Task<Response<Message>> SendAsync(MimeMessage emailMessage) {
2     using var client = new SmtpClient();
3     try {
4         await client.ConnectAsync(emailConfiguration.SmtpServer, emailConfiguration.Port
5             ↪ , MailKit.Security.SecureSocketOptions.StartTls);
6         await client.AuthenticateAsync(emailConfiguration.UserName, emailConfiguration.
7             ↪ Password);
8         await client.SendAsync(emailMessage);
9         await client.DisconnectAsync(true);
10        return new Response<Message>(new Message());
11    }
12    catch (Exception e) {
13        Console.WriteLine("Mailkit {e.Message}");
14    }
15 }
```

```

12     await client.DisconnectAsync(true);
13     return new Response<Message>("Wystąpił błąd serwera pocztowego");
14 }
15 }
```

Listing 4.21: Kod metody serwisu odpowiedzialnej za wysyłanie wiadomości e-mail

4.5.5. Wyznaczanie trasy dostawy

Moduł wyznaczania trasy dostawy, implementuje przedstawiony we wstępie teoretycznym, algorytm genetyczny dla problemu komiwojażera. W ramach funkcjonalności modułu, użytkownik, na podstawie danych z przypisanych do niego dokumentów, może wyznaczyć trasę jaką powinien obrać, aby dostarczyć wszystkie towary powiązane z posiadanymi poświadczaniami sprzedażą.

Z racji specyfiki problemu, która przedstawiona została w sposób szczegółowy w rozdziale ??, przygotowany algorytm wyznacza trasę optymalną, lub w znacznym stopniu zbliżoną do optymalnej. Przy założeniu dotyczącym możliwości wprowadzania nieograniczonej liczby punktów trasy, uzyskanie wyniku optymalnego dla każdej instancji problemu jest niemożliwe.

Po przejściu do widoku opisywanej funkcjonalności, użytkownikowi ukazywana jest lista znalezionych punktów dostawy oraz fragment mapy. Dla każdego z przypisanych do pracownika dokumentów, jeżeli w ramach faktury określone zostały dane adresowe, znajdowane są współrzędne miejsca dostawy, a następnie dodawane są one na mapie. W przypadku, gdy z poświadczaniem sprzedażą nie zostały powiązane jakiekolwiek dane adresu, dokument ukazywany zostaje w sekcji znalezionych punktów z adnotacją o konieczności wybrania dla niego lokalizacji.

Po opcjonalnym przypisaniu danych lokalizacyjnych do faktur, użytkownik wybiera przycisk wywołujący wykonanie algorytmu.

Aby uzyskać wynik działania algorytmu, należy wysłać żądanie do interfejsu API, który w ramach określonego punktu końcowego, wykona kod kontrolera oraz serwisu dla procedury wyliczania trasy.

W momencie uzyskania żądania dotyczącego wyznaczenia trasy, walidowany zostaje obiekt, w którym określone są punkty pokonywanej drogi. Następnie, punkty te zostają odwzorowane na instancje klas modelu interfejsu API. W kolejnym kroku, wywoływany zostaje serwis realizujący operację wyliczania drogi.

W ramach serwisu, na początku definiowane są parametry konfiguracyjne algorytmu. Wśród tych parametrów możemy wyróżnić: prawdopodobieństwo mutacji, liczba punktów trasy, rozmiar populacji, oraz liczba osobników dominujących dla populacji potomnej. W przypadku dwóch pierwszych charakterystyk, wartości te są stałe. Dwie pozostałe natomiast, zależne są od liczby punktów trasy i wynoszą kolejno: 0.75 oraz 0.25 z liczby wszystkich miejsc dostawy. Obie wartości zaokrąglane są w dół do najbliższych liczb naturalnych.

W kolejnym kroku, po wprowadzeniu wartości konfiguracyjnych, uruchamiany jest nowy wątek, w ramach którego przeprowadzane są obliczenia. Czas trwania tych obliczeń, zdefiniowano na 60 sekund. Okres ten, stanowi warunek zakończenia algorytmu.

Początkowym krokiem procedury, jest zdefiniowanie rozwiązania początkowego, będącego losowym ułożeniem wszystkich wprowadzonych punktów dostawy. Następnie, czynności wykonywane są w pętli, dla której warunkiem zakończenia jest flaga, modyfikowana w momencie upływu czasu obliczeń.

W pierwszym kroku pętli, aktualne rozwiązanie określone zostaje jako najlepsze. Kolejno, wyliczana jest wartość funkcji przystosowania populacji dla aktualnie najbardziej optymalnej trasy. W następnym kroku, rozpoczyna się etap ewolucji rozwiązań, w ramach którego tworzona jest nowa populacja. Populacja ta, powstaje na bazie osobników dominujących z poprzedniej

grupy oraz operacji selekcji, krzyżowania, a także mutacji. Jeżeli wartość funkcji przystosowania populacji potomnej jest większa od wartości dla poprzedniej instancji, modyfikowana zostaje flaga, pozwalająca na określenie rozwiązań, jako nowe najbardziej optymalne.

Z chwilą zakończenia pętli, wywoływana jest procedura zapisująca najlepszy wynik oraz przypisującą kolejnym punktom trasy numery porządkowe.

Na listingu 4.22 przedstawiona została omówiona powyżej metoda wyznaczania trasy dostawy dokumentów.

```

1 public void Run() {
2     DeliveryRoad startDeliveryRoad = new DeliveryRoad(Coordinates);
3     Population population = Population.DrawPopulation(startDeliveryRoad, Config.
    ↪ PopulationSize);
4     var improvement = true;
5     _timer.Start();
6
7     while (!Timeout)
8     {
9         if (improvement == true) {
10             SetBestRoad(population);
11         }
12
13         improvement = false;
14         double currentFitnessFunctionValue = population.MaximalFitnessRatio;
15
16         population = population.PerformEvolution();
17         if (currentFitnessFunctionValue < population.MaximalFitnessRatio){
18             improvement = true;
19         }
20     }
21     prepareBestRoadCoords();
22 }
```

Listing 4.22: Kod metody odpowiedzialnej za wyznaczenie trasy dostawy dla zbioru dokumentów

Kluczowym elementem omawianej metody jest funkcja składowa o nazwie **PerformEvolution**. Wewnątrz tej funkcji, wybierane zostają osobniki dominujące z poprzedniej populacji oraz nowe osobniki zbioru potomnego. Populacja potomna stanowi złączenie obu zbiorów osobników.

Kod źródłowy funkcji składowej **PerformEvolution** ukazany został na listingu 4.23.

```

1 public Population PerformEvolution()
2 {
3     var dominants = GenerateDominants(Config.NumberOfDominantsInDescendantPopulation)
    ↪ ;
4     var otherDescendants = GenerateDescendantIndividuals(Config.PopulationSize -
    ↪ Config.NumberOfDominantsInDescendantPopulation);
5     var membersOfDescendantPopulation = otherDescendants.Individuals.Concat(dominants
    ↪ .Individuals).ToList();
6     var population = new Population(membersOfDescendantPopulation);
7     return population;
8 }
```

Listing 4.23: Kod funkcji składowej wyboru populacji potomnej

Aby zrozumieć zasadę działania funkcji z listingu 4.23, należy przeanalizować metody **GenerateDominants** oraz **GenerateDescendantIndividuals**. Pierwsza z nich, prowadzi do wyboru osobników o największej wartości funkcji dopasowania, spośród tych, aktualnie znajdujących się w populacji.

Na początku, tworzona jest instancja populacji na bazie aktualnie wygenerowanych osobników. Ponadto, zdefiniowana zostaje także lista rozwiązań, która będzie stanowiła zbiór osob-

ników zwracanej populacji. W pętli, wykonywanej tyle razy, ile wynosi wartość określona w parametrach konfiguracyjnych, wybierane są najlepiej dopasowane osobniki, po czym dodawane zostają do zwracanej listy.

Na listingu 4.24, zilustrowano kod metody `GenerateDominants`.

```

1 public Population GenerateDominants(int count) {
2     var population = new Population(Individuals);
3     var dominants = new List<DeliveryRoad>();
4     var i = 0;
5     while (i < count)
6     {
7         dominants.Add(population.GetBestIndividual());
8         var modifiedIndividuals = population.Individuals.Except(dominants).ToList();
9         population = new Population(modifiedIndividuals);
10        i++;
11    }
12    var dominativePopulation = new Population(dominants);
13    return dominativePopulation;
14 }
```

Listing 4.24: Kod metody generującej zbiór osobników dominujących dla nowej populacji

Metoda `GenerateDescendantIndividuals` z kolei, utworzona została w celu wygenerowania pozostałych osobników populacji potomnej, za pomocą operacji selekcji, krzyżowania oraz mutacji. Operacje te, wykonywane są w pętli, tyle razy, ilu osobników należy wygenerować do nowej populacji. Po przeprowadzeniu procesów algorytmu genetycznego, jednostki dodawane są do zwracanego zbioru.

W ramach listingu 4.27, przedstawiono kod źródłowy metody generowania pozostałych osobników populacji.

```

1 public Population GenerateDescendantIndividuals(int count) {
2     var individuals = new List<DeliveryRoad>();
3     var i = 0;
4     while (i < count)
5     {
6         var delivery = PerformSelection();
7         delivery = delivery.PerformCrossing(PerformSelection());
8
9         for (var j = 0; j < delivery.Vertices.Count; i++) {
10             delivery = delivery.PerformMutation();
11         }
12
13         individuals.Add(delivery);
14         i++;
15     }
16
17     var population = new Population(individuals);
18     return population;
19 }
```

Listing 4.25: Kod metody generującej zbiór pozostałych osobników dla nowej populacji

Podstawowymi elementami składowymi każdego algorytmu genetycznego są operacje: selekcji, krzyżowania oraz mutacji. Poniżej, każda z nich zostanie szczegółowo omówiona.

W odniesieniu do procedury selekcji osobników, pierwszym krokiem tej operacji wygenerowanie losowej wartości stanowiącej indeks osobnika z populacji. Następnie, wyliczany jest ułamek, określający szansę osobnika, na dostanie się do populacji potomnej. Szansa ta, to iloraz wartości funkcji przystosowania osobnika przez maksymalną wartość przystosowania populacji.

Jeżeli szansa osobnika, jest większa od losowo wygenerowanej liczby z przedziału $<0,1>$, wówczas osobnik trafia do populacji potomnej.

Opisana metoda selekcji, nazywana metodą ruletkową, ukazana została na listingu 4.27.

```

1 public DeliveryRoad PerformSelection() {
2   for (;;) {
3     var index = Generator.Next(0, Config.PopulationSize);
4     double percentOfChances = Individuals[index].FitnessRatio / MaximalFitnessRatio;
5     if (percentOfChances > Generator.NextDouble()) {
6       var vertices = Individuals[index].Vertices;
7       return new DeliveryRoad(vertices);
8     }
9   }
10 }
```

Listing 4.26: Kod metody generującej zbiór pozostałych osobników dla nowej populacji

Drugim z elementów algorytmu genetycznego jest operacja krzyżowania. W procedurze tej, losowane są dwie zmienne z zakresu liczności punktów dostaw. Pierwsza z nich, posiada zawsze mniejszą wartość od drugiej, ponieważ losowanie kolejnej z nich odbywa się w zakresie rozpoczynającym się od wartości liczby już wylosowanej.

Następnie wykonywana jest zamiana fragmentów list wierzchołków (punktów dostaw), w obrębie dwóch różnych osobników. Z jednego osobnika, pobierany jest fragment ograniczony przez zdefiniowane uprzednio liczby, natomiast drugi osobnik dostarcza pozostałe z fragmentów trasy.

Przedstawiona procedura krzyżowania, opisywana jest jako krzyżowanie dwupunktowe. Treść omówionego kodu źródłowego dla operacji krzyżowania zaprezentowana została na listingu 4.27.

```

1 public DeliveryRoad PerformCrossing(DeliveryRoad road) {
2   var firstPoint = Generator.Next(0, road.Vertices.Count);
3   var secondPoint = Generator.Next(firstPoint, road.Vertices.Count);
4   DeliveryRoad delivery = null;
5
6   var firstsIndividualPiece = Vertices.GetRange(firstPoint, secondPoint-firstPoint
    ↵ +1);
7   var secondsIndividualPiece = road.Vertices.Except(firstsIndividualPiece).ToList()
    ↵ ;
8   var crossed = secondsIndividualPiece.Take(firstPoint).Concat(
    ↵ firstsIndividualPiece).Concat(secondsIndividualPiece.Skip(firstPoint)).ToL
    ↵ ist();
9
10  delivery = new DeliveryRoad(crossed);
11  return delivery;
12 }
```

Listing 4.27: Kod metody generującej zbiór pozostałych osobników dla nowej populacji

Ostatnią operacją kluczową dla algorytmu genetycznego jest mutacja osobnika. W ramach tej operacji, początkowo wyliczana jest losowa wartość z przedziału [0,1], determinująca prawdopodobieństwo zajścia zdarzenia mutacji. Kolejno, porównana zostaje wylosowana wartość z daną konfiguracyjną prawdopodobieństwa mutacji. Jeżeli wartość danej konfiguracyjnej jest większa od wygenerowanej losowo liczby, omawiana operacja zachodzi.

Następnym krokiem jest wyznaczenie dwóch punktów dostaw, stanowiących wierzchołki trasy, w celu zamiany ich kolejności. Zmodyfikowana trasa, reprezentująca pojedynczego osobnika, zostaje na końcu zwrócona do funkcji wywołującej.

Na listingu 4.28, zilustrowano kod źródłowy metody wykonującej operację mutacji.

```

1 public DeliveryRoad PerformMutation() {
2   var coordinates = new List<Vertex>(Vertices);
3   var probability = Generator.NextDouble();
```

```

4 DeliveryRoad delivery = null;
5
6 if (Config.MutationProbability > probability) {
7     var swappedIndexOne = Generator.Next(0, Vertices.Count);
8     var swappedIndexTwo = Generator.Next(0, Vertices.Count);
9
10    var temp = coordinates[swappedIndexOne];
11    coordinates[swappedIndexOne] = coordinates[swappedIndexTwo];
12    coordinates[swappedIndexTwo] = temp;
13 }
14
15 delivery = new DeliveryRoad(coordinates);
16 return delivery;
17 }
```

Listing 4.28: Kod funkcji składowej wykonującej operację mutacji osobnika

4.5.6. Zarządzanie systemem

Moduł zarządzania systemem zawiera w sobie funkcjonalności, pozwalające na modyfikację zbioru danych, dotyczącego kluczowych aspektów przedstawianego oprogramowania. Wyróżnić możemy w nim: edycję danych konfiguracyjnych, usuwanie błędnie wprowadzonych dokumentów, zarządzanie wyjaśnieniami, administrację danych pracowników, aktualizację informacji o pojazdach, przyznawanie oraz odbieranie uprawnień, a także przegląd statystyk.

W ramach pierwszej funkcjonalności, użytkownik może określić dane adresowe i lokalizacyjne firmy. Informacje te, potrzebne są do prawidłowego działania modułu wyznaczania trasy dostawy oraz nadzorowania wewnętrznych dostawców. Pełnią one rolę punktu startowego, dla opisanego w podrozdziale 4.5.5 algorytmu komiwojażera, a także pozwalają określić pozycję pracownika na mapie, w sytuacji, gdy nie został wprowadzony jakikolwiek wpis lokalizacji urządzenia mobilnego.

Dodatkowo, w obszarze formularza danych konfiguracyjnych, znajdują się informacje odnośnie czasu trwania: procedury skanowania dokumentów, przechowywania wpisów w historii lokalizacji, gromadzenia dziennych statystyk kierowców, czy też magazynowania adnotacji archiwalnych dotyczących faktur. Wartości te, wykorzystywane są przez skrypty archiwizujące, w celu określenia zakresu danych, które mają zostać usunięte, wraz z zakończeniem dnia pracy przedsiębiorstwa.

Ponadto, formularz zawiera przełącznik, determinujący domyślny tryb skanowania dokumentów. Jeżeli znajduje się on w pozycji włączonej, w funkcjonalnościach skanowania uruchomione zostają komponenty odpowiedzialne za użycie ręcznego skanera kodów QR. W przeciwnym razie, wyświetlonym komponentem, jest fragment interfejsu pozwalający na wykorzystanie kamery internetowej.

Wszystkie z wymienionych opcji konfiguracyjnych, przechowywane są w bazie danych w formie rekordu jednowierszowej tabeli.

Następną funkcjonalnością modułu jest usuwanie dokumentów. Narzędzie to, zostało stworzone, aby zapobiegać sytuacji omyłkowego wprowadzenia do systemu niepożądanej faktury. W ramach aplikacji webowej, przygotowany został widok tabelaryczny, w którym, przy każdym z dokumentów znajduje się przycisk jego usunięcia. Naciśnięcie przycisku, powoduje wywołanie żądania usunięcia dowodu sprzedaży poprzez interfejs API. W związku z ustawnionymi wymogami, dotyczącymi modyfikacji danych w bazie (*ang. constraints*), w momencie usunięcia dokumentu, w sposób kaskadowy skasowane zostają informacje o jego przekazaniach, a także wszystkie przypisane do niego uwagi i wyjaśnienia. Rekomendowanym jest, przypi-

sanie uprawnienia do tej funkcjonalności, tylko i wyłącznie użytkownikowi sprawującemu rolę administratora systemu, który mógłby z niej skorzystać tylko w szczególnych przypadkach.

Trzecią funkcjonalnością segmentu zarządzania systemem jest kontrola wyjaśnień. W ramach tej części aplikacji, użytkownikowi przedstawiany jest widok, w którym w kolejnych wierszach tabeli ukazane zostają informacje, dotyczące utworzonych wyjaśnień dokumentów. Pracownik może analizować przedstawione wyjaśnienia, a także, po naciśnięciu określonego przycisku, unieważniać je.

Unieważnienie wyjaśnienia dokumentu wiąże się z jego usunięciem z bazy danych, co skutkuje pozostaniem poświadczenia sprzedaży „w obiegu”, na kolejne dni pracownicze.

Kolejnymi funkcjonalnościami są widoki służące do modyfikacji danych pracowników oraz pojazdów. W ich kontekście, przedstawić możemy podstawowe operacje na zbiorach danych, takie jak dodawanie, usuwanie czy modyfikacja informacji o pracownikach, a także samochodach, dostępnych w obrębie oddziału firmy. Oba widoki oparte są o tabele przedstawiające zbiory danych, a także formularze w oknach modalnych, wykorzystywane do wprowadzania oraz edycji elementów zbiorów.

Poza przedstawionymi funkcjonalnościami wyróżnić możemy także, segment przyznawania uprawnień pracownikom. W segmencie tym, użytkownik wybiera pracownika, spośród listy wszystkich dostępnych, a następnie, z wykorzystaniem listy przełączników, definiuje do których fragmentów aplikacji pracownik powinien mieć dostęp.

Operacje przyznawania oraz odbierania uprawnień pracowniczych, sprowadzają się do wysyłania żądań dodania lub usunięcia wpisu za pomocą interfejsu API.

Ostatnią funkcjonalnością modułu zarządzania systemem jest przegląd statystyk. W ramach tego fragmentu aplikacji, użytkownikowi pokazywane są cztery rodzaje wykresów, opisujących efektywność pracowników oraz zużycie pojazdów. W związku z większym stopniem zaawansowania operacji, dotyczących dostarczania danych dla wykresów, w przeciwieństwie do działań wykonywanych w ramach poprzednich funkcjonalności, przedstawiony zostanie opis przytaczanej operacji oraz kody źródłowe dla konkretnych jej elementów.

Po uruchomieniu widoku przeglądania statystyk, aplikacja webowa wysyła do interfejsu API cztery niezależne żądania, dotyczące danych dla wykresów. Żądania te, obsługiwane są przez metody punktów końcowych, zgodnie ze schematem wprowadzonym w podrozdziale 4.1.

Momentem faktycznego uzyskiwania danych dla wykresów, jest wywołanie metody klasy repozytorium. Jedna z takich metod, tj. funkcja składowa uzyskiwania kilometrów wykonanych przez pracowników, przedstawiona została na listingu 4.29.

Na początku funkcji, pobierana zostaje lista wszystkich pracowników oddziału przedsiębiorstwa. Następnie, tworzony jest pomocniczy obiekt, który na końcu metody zostanie zwrocony jako rozwiązanie.

Dla każdego z pracowników, uzyskiwane są wpisy kilometrów ich autorstwa, wykonane w przedziale czasowym określonym przez obiekt `DateRange`, będący parametrem funkcji składowej. Dla opisywanego żądania, tj. wykonywanego w momencie uruchomienia widoku, obiekt zakresu dat posiada takie wartości, które nie ograniczają wyboru wpisów.

Jeżeli uzyskany zbiór adnotacji kilometrowych nie jest pusty, tworzony zostaje obiekt statystyki dla pojedynczego użytkownika. Następnie, zdefiniowany jest licznik, przechowujący liczbę wykonanych przez pracownika kilometrów. Iterując po każdej z adnotacji, znajdowana zostaje liczba kilometrów, przypisana do pojazdu uwzględnionego w adnotacji, z datą z przed wprowadzeniem wpisu. Jeżeli taka liczba istnieje, należy ją dodać do sumy kilometrów pracownika, w przeciwnym razie, suma kilometrów uzupełniana jest wartością z wpisu użytkownika.

Po wykonaniu obliczeń w ramach pętli adnotacji kilometrowych, do obiektu statystyki pracownika dopisywana jest wyznaczona liczba kilometrów, a do zwracanej listy, obiekt statystyki.

Z chwilą zakończenia pętli iterującej po zbiorze pracowników, zwracana jest skonstruowana lista.

```

1 public async Task<IList<WorkerStat>> GetKilometersForWorkersStatsAsync(DateRange
    ↪ dateRange) {
2     var workers = await context.Workers.ToListAsync();
3     var stats = new List<WorkerStat>();
4
5     foreach (var worker in workers) {
6         var workersOdometers = await context.VehicleOdometers
7             .Include(w => w.Worker)
8             .Include(w => w.Vehicle)
9             .Where(w => w.WorkerId == worker.Id && w.Date.Date.CompareTo(dateRange.
    ↪ FromDate.Date) >= 0 && w.Date.Date.CompareTo(dateRange.ToDate.Date)
    ↪ <= 0)
10            .ToListAsync();
11
12        if (workersOdometers != null) {
13            var workerStat = new WorkerStat {
14                WorkerId = worker.Id,
15                WorkerFirstName = worker.FirstName,
16                WorkerLastName = worker.LastName,
17            };
18
19            kilometers = 0.0;
20
21            foreach (var odometers in workersOdometers) {
22                var previousKilometers = await context.VehicleOdometers
23                    .Where(x => x.VehicleId == odometers.VehicleId && x.Date.CompareTo(
    ↪ odometers.Date) < 0)
24                    .OrderByDescending(x => x.Date).ToListAsync();
25
26                if (previousKilometers.Count > 0) {
27                    var previous = previousKilometers.First();
28                    kilometers += (odometers.Value - previous.Value);
29                }
30                else {
31                    kilometers += odometers.Value;
32                }
33            }
34            workerStat.Kilometers = kilometers;
35            stats.Add(workerStat);
36        }
37    }
38    return stats;
39 }
```

Listing 4.29: Kod metody repozytorium przygotowującej dane dla wykresu kilometrów

Po zilustrowaniu w widoku wszystkich wykresów, użytkownik może wprowadzać do znajdujących się nad nimi formularzy, określone zakresy dat. Czynność ta, skutkuje ponowieniem żądania uzyskania danych od interfejsu API. W tym przypadku jednak, widoczne na listingu 4.29 kalkulacje, uwzględniąć będą dokładny przedział czasowy.

4.6. Aplikacja mobilna

4.6.1. Panel główny użytkownika

Panel główny użytkownika jest podstawowym widokiem utworzonym w ramach aplikacji mobilnej. Wyróżnić możemy w nim trzy zakładki, prezentujące odpowiednio: dokumenty dostarczone przez pracownika, dokumenty dla których pracownik zgłosił uwagę oraz pozostałe fakty. W obszarze każdej zakładki, wyświetlana jest lista dokumentów, wraz ze zdefiniowanymi dla nich akcjami.

Dla każdego z dokumentów wykonać można następujące operacje:

- Przekazanie klientowi
- Uruchomienie nawigacji do klienta
- Przekazanie innemu pracownikowi
- Poinformowanie klienta o zbliżającej się dostawie
- Zgłoszenie uwagi.

Operacje przekazania faktur do klienta lub pracownika, zaimplementowane zostały w sposób analogiczny do aplikacji webowej. Po naciśnięciu określonego przycisku, aplikacja wysyła żądanego do interfejsu API, dostarczając jako zasób obiekt przekazania, zawierający między innymi nadawcę, odbiorcę oraz transferowany dokument. Po otrzymaniu poprawnej odpowiedzi na wysłane żądanie, aplikacja wywołuje metodę odświeżenia listy dokumentów.

Funkcja uruchomienia nawigacji do klienta, w początkowej fazie weryfikuje obecność danych adresowych, dotyczących miejsca dostawy oraz kontrahenta. Jeżeli informacje o miejscu dostawy są przypisane do klienta, wywoływana jest metoda `navigate` klasy `LaunchNavigator`. Parametrem tej metody, jest wynik zwrócony z funkcji, przygotowującej ciąg tekstowy na podstawie danych adresowych.

W przypadku braku informacji dotyczących adresu miejsca dostawy, weryfikowane są dane adresowe kontrahenta. W sytuacji pomyślnej identyfikacji tych danych, aplikacja wyświetla komunikat, w ramach którego, użytkownik musi zdecydować, czy chce być kierowany pod adres klienta, czy też procedura ma się zakończyć.

Jeżeli użytkownik zaakceptuje wyświetlony komunikat, wykonywana jest analogiczna czynność do opisanej powyżej dla miejsca dostawy.

Moduł informowania klienta o zbliżającej się dostawie, został zaimplementowany analogicznie do funkcjonalności powiadamiania kontrahentów w aplikacji webowej. Po wybraniu przycisku informowania, wykonywany jest kod punktu końcowego API, odpowiedzialnego za wysyłanie wiadomości e-mail. Na podstawie dostarczonego do API obiektu zasobu żądania, podczas wysłania wiadomości wybierany zostaje szablon, w treści którego widnieje informacja o niezwłocznej wizycie pracownika. Nadanie wiadomości, determinowane jest obecnością w bazie danych, informacji kontaktowych dla miejsca dostawy. W przypadku braku tych informacji sprawdzane są dane kontaktowe kontrahenta.

Funkcjonalność zgłoszenia uwagi, pozwala pracownikowi poradzić sobie w sytuacji, w której dla danego dokumentu, lista zabranych przez niego z magazynu produktów jest niekompletna. Jeżeli użytkownik zauważa taką ewentualność podczas dostawy towaru, musi utworzyć notatkę uwagi. Po naciśnięciu przycisku uwagi dla konkretnego dokumentu, wyświetlane jest pole tekstowe, wewnątrz którego należy wprowadzić treść informacji. Po zatwierdzeniu wpisanego tekstu, notatka uwagi zapisywana jest do bazy danych poprzez interfejs API.

Zapis notatki, determinuje aktualizację widoków śledzenia dokumentów dla wszystkich klientów aplikacji webowych. Dzięki temu, pracownik stacjonarny oddziału firmy, na bieżąco dowiaduje się o zgłoszonych uwagach.

4.6.2. Wprowadzanie wpisów kilometrów

Widok wprowadzania wpisów kilometrów, pozwala użytkownikowi określić stan licznika kilometrów, dla pojazdu, z wykorzystaniem którego dostarczał towar. Przedstawiana aktywność zawiera formularz, w ramach którego wyróżnić możemy cztery pola.

Pierwsze z nich, to lista rozwijana, z której pracownik wybiera samochód. Z chwilą uruchomienia widoku, pobierana zostaje lista wszystkich pojazdów, a także, jeżeli jest taka konieczność, zbiór pracowników. Jeżeli do pracownika, przypisany jest pojazd domyślny, zostanie on wyświetlony jako pierwszy w liście rozwijanej.

Kolejne pola określają pracownika oraz datę wprowadzenia wpisu. Pole te są wypełniane automatycznie, a użytkownik nie może zmienić ich zawartości.

Ostatnim fragmentem formularza jest pole tekstowe pozwalające na wprowadzenie liczby kilometrów. W momencie, w którym pole to, zostanie wypełnione, a także wybrany jest pojazd, przycisk zatwierdzenia staje się aktywny i może zostać wysłane żądanie zapisu adnotacji do bazy danych.

Dane, wprowadzane przez użytkownika, podobnie jak w pozostałych formularzach w obrębie aplikacji mobilnej, przechowywane są z wykorzystaniem stanu wewnętrznego komponentu aplikacji. W przypadku operacji dodania wpisu, odwołuje się ona natomiast, do klasy magazynu, wpływającej na stan zewnętrzny.

4.6.3. Wprowadzanie notatek handlowych

Aktywność wprowadzania notatek, pozwala pracownikom handlowym, na definiowanie adnotacji dotyczących wykonywanych przez nich odwiedzin klientów. W ramach widoku, wyszczególnionych jest pięć pól formularza, określających: pracownika, aktualny czas, lokalizację, typ oraz treść notatki.

Po uruchomieniu aktywności, pobierana jest lista pracowników, dostępne typy notatek oraz aktualna lokalizacja telefonu (informacje dotyczące sposobu pozyskiwania lokalizacji telefonu użytkownika opisane zostały w rozdziale 4.6.4). Informacje te, poza lokalizacją, uzyskiwane są poprzez odpowiedź na żądanie interfejsu programowania aplikacji, tylko wtedy, gdy nie zostały one wcześniej pozyskane przez inny widok aplikacji.

Formularz wprowadzania notatek, podobnie jak opisywany w rozdziale 4.6.2 formularz wprowadzania kilometrów, wykonany został z wykorzystaniem techniki komponentów kontrolowanych. Oznacza to, że dla każdego z wypełnianych pól ankiety, zdefiniowany został wewnętrzny element stanu przechowujący jego zawartość. W momencie zdarzenia, zmiany wartości pola, wywoływana jest metoda aktualizacji elementu stanu. Dzięki wykorzystaniu tej metody, wartości pól formularza dostępne są dla funkcji zapisującej ankietę, w sposób nie wymagający tworzenia obiektów referencji.

4.6.4. Cykliczna lokalizacja urządzenia

Kluczową funkcjonalnością, wykorzystywaną w wielu widokach aplikacji mobilnej jest lokalizowanie telefonu użytkownika. Moduł ten, wykonany został, zarówno poprzez zdefiniowanie odpowiednich komponentów i usług za pomocą biblioteki React-Native, jak i poprzez napisanie kodu natywnego w języku Java dla systemu Android.

Lokalizacja użytkownika, odbywa się poprzez cykliczne wywołania funkcji, wewnątrz aplikacji komponentów stworzonych w React-Native. Aby funkcja ta, mogła być wywoływana, niezależnie od renderowanego widoku aplikacji (tj. działać w tle), musi zostać zdefiniowana jako osobne zadanie (tzw. *Headless Task*).

Poprawne zarejestrowanie osobnego zadania, wymaga dokonania zmian w kodzie natywnym aplikacji. Zmiany te, polegają na utworzeniu nowej klasy dziedziczącej po serwisie

HeadlessJsTaskService. Klasa ta, w metodzie `getTaskConfig`, ma za zadanie pobrać informacje dodatkowe, dotyczące przekazywanej jako argument intencji. Intencja natomiast, stanowi obiekt umożliwiający połączenie ze sobą aktywności w ramach aplikacji. Jeżeli obiekt intencji, posiada dodatkowe informacje, to są one wykorzystywane jako parametr wywołania konstruktora obiektu zwracanej klasy `HeadlessJsTaskConfig`. Ponadto, pozostałymi parametrami są: nazwa tworzonego zadania, wartość ograniczenia czasowego dla zadania, a także możliwość wykonywania działania z koniecznością poinformowania o tym fakcie użytkownika. Opisany kod metody, z klasy dziedziczącej po klasie serwisu `HeadlessJsTaskService`, przedstawiony został na listingu 4.30.

```

1 protected HeadlessJsTaskConfig getTaskConfig(Intent passedIntent) {
2   Bundle intentsExtras = passedIntent.getExtras();
3   return new HeadlessJsTaskConfig("Notification", intentsExtras != null ? Arguments
    ↪ .fromBundle(intentsExtras) : Arguments.createMap(), 0, true);
4 }
```

Listing 4.30: Kod metody pozwalającej na zarejestrowanie osobnego zadania

Aby kod funkcji, zdefiniowanej z obszarze aplikacji React, wykonywany był cyklicznie, zdefiniowany musi zostać nowy wątek, dla klasy serwisu powiadomień, wywołujący zadanie, co określony przedział czasu. Wątek ten, rozpoczyna swoje działanie z chwilą wywołania metody `startService` dla obiektu klasy serwisu.

W ramach wątku (tj. obiektu klasy `Runnable`), uruchamiana zostaje metoda `run`. W metodzie tej, na początku pobierany jest kontekst aplikacji. Dzięki niemu, możliwe jest utworzenie nowej intencji (obiektu klasy `Intent`), która wiąże stan aplikacji w obecnym momencie jej działania z obiektem klasy osobnych zadań, opisany w powyższych akapitach. Posiadając takie połączenie, wywoływana zostaje metoda uruchomienia serwisu, której parametrem jest utworzona intencja. Następnie, wykorzystywana zostaje funkcja `acquireWakeLockNow`, której celem jest sprawienie, aby urządzenie nie przeszło w stan uśpienia, przed rozpoczęciem zadania. Ostatnim elementem wątku, jest wywołanie całej metody `run` ponownie, po upływie piętnastu minut (tj. 300000ms). Omówiona funkcja ukazana została na listingu 4.31.

```

1 private Runnable runnableLocationCode = new Runnable() {
2   @Override
3   public void run() {
4     Context mainApplicationContext = getApplicationContext();
5     Intent intentToHeadlessTask = new Intent(mainApplicationContext, HeadlessTaskConfig.
    ↪ class);
6     mainApplicationContext.startService(intentToHeadlessTask);
7     HeadlessJsTaskService.acquireWakeLockNow(mainApplicationContext);
8     handler.postDelayed(this, 300000);
9   }
10 };
```

Listing 4.31: Kod wątku wykonującego zadanie lokalizacji z interwałem czasowym

Po zdefiniowaniu kodu dla warstwy natywnej aplikacji mobilnej, przygotowany został kod zadania, wykonywanego wewnątrz warstwy aplikacji React-Native. W ramach tego kodu, weryfikowane jest istnienie zalogowanego użytkownika. Jeżeli weryfikacja ta, przebiega pomyślnie, wywoływana zostaje statyczna metoda, której zadaniem jest uzyskanie lokalizacji oraz zapisanie pozycji pracownika w bazie danych.

Wewnątrz metody lokalizującej, na początku wykonywane jest żądanie uprawnień od użytkownika. Jeżeli pracownik zezwoli na udostępnienia do lokalizacji, wykonywana zostaje faktyczna procedura pozyskiwania danych geolokalizacyjnych.

W ramach tej procedury, wykorzystywana jest metoda `getCurrentPosition` klasy `Geolocation`, będąca elementem wbudowanym pakietu dostarczanego przez twórców React-Native. Wewnątrz wywołania zwrotnego, wykonywanego w momencie uzyskania danych,

punkty geograficzne zapisywane są do atrybutów obiektu własnej klasy, a następnie wysyłane poprzez żądanie do interfejsu API. W przypadku błędu podczas uzyskiwania lokalizacji, wywoływana zostaje metoda analogiczna do `getCurrentPosition`, lecz z parametrem `enableHighAccuracy` posiadającym wartość `false`. Parametr ten, determinuje dokładność lokalizacji, poprzez wykorzystanie konkretnych metod lokalizujących. Na listingu 4.32, zilustrowano opisany w niniejszym akapicie kod źródłowy procedury lokalizacji.

```

1 getLocation = async (user) => {
2     Geolocation.getCurrentPosition(
3         async (position) => {
4             this.currentLocation.latitude = Number(
5                 position.coords.latitude,
6                 ).toFixed(7);
7             this.currentLocation.longitude = Number(
8                 position.coords.longitude,
9                 ).toFixed(7);
10            await this.upload(user);
11        },
12        (error) => {
13            this.getLocationWithoutHighAccuracy(user);
14        },
15        {enableHighAccuracy: true, timeout: 5000},
16    );
17};

```

Listing 4.32: Kod metody uzyskującej lokalizację urządzenia użytkownika

4.7. Skrypty archiwizujące

W celu usuwania zbędnych informacji, oraz archiwizacji dokumentów nie będących już w obiegu pracowniczym, utworzone zostały skrypty porządkujące dane systemu. Skrypty te, mogą być wywoływane cyklicznie, o określonych porach dnia, poprzez dodanie ich jako zadania, w narzędziach takich jak np. aplikacja `cron`.

Głównym zadaniem skryptu jest kontrolowanie czasu od poprzedniego wykonania, a w określonym momencie, zdefiniowanie żądania do interfejsu API, oraz odebranie jego odpowiedzi.

Utworzono w ramach systemu skrypty dotyczą: przechowywania wpisów historii lokalizacji, gromadzenia dziennych statystyk, zachowywania dokumentów w archiwum, a także przenoszenia faktur kończących obieg pracowniczy, do zbioru archiwalnego.

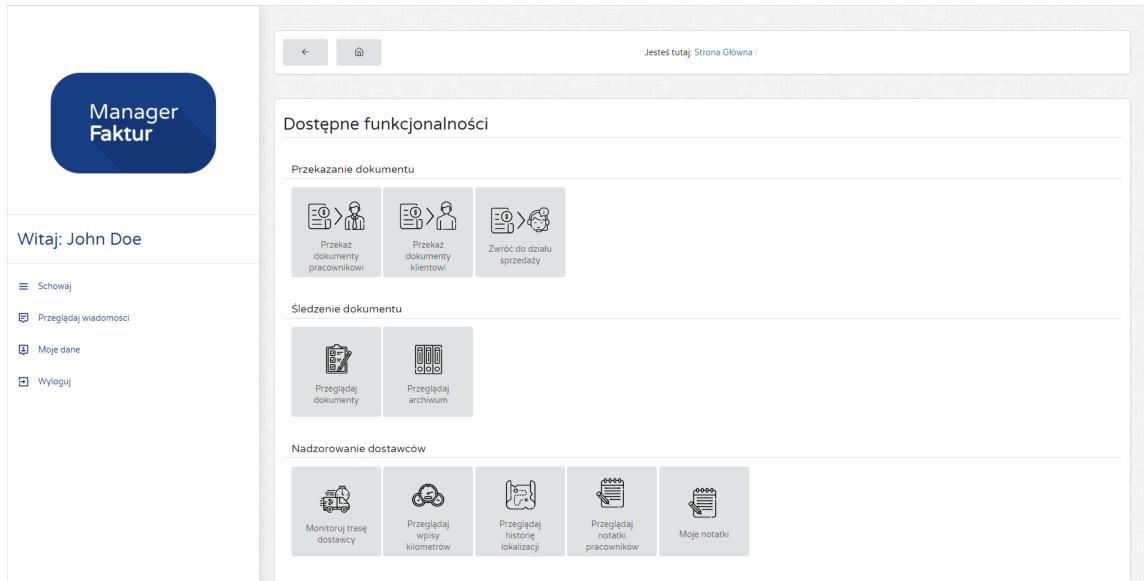
W obecnym rozdziale, przedstawiona zostanie implementacja programu, kontrolującego proces przechowywania wpisów historii lokalizacji.

Na początku, wykonywane jest żądanie logowania, w celu uzyskania tokenu autoryzującego. Danymi tego żądania, są login oraz hasło dedykowanego do wykonywania skryptów użytkownika systemu. Po poprawnym uwierzytelnieniu, pobierany z serwera jest rekord, określający konfigurację oprogramowania. Na podstawie jego danych, będzie można zweryfikować konieczność wykonywania operacji archiwizacji. Po uzyskaniu danych konfiguracji, wyliczana jest różnica dni pomiędzy aktualną datą a momentem ostatniej modyfikacji parametru przechowywania wpisów lokalizacyjnych. Jeżeli różnica ta, jest większa od zera, a ponadto reszta z jej dzielenia przez liczbę przechowywaną w konfiguracji jest równa zero, należy wywołać punkt końcowy, archiwizujący wpisy lokalizacyjne z konkretnej liczby ostatnich dni. W momencie niepowodzenia, jakiekolwiek z opisywanych operacji, lub zwrócenia nieoczekiwanej kodu odpowiedzi na żądanie, informacja ta jest odnotowywana w dzienniku logów systemu operacyjnego.

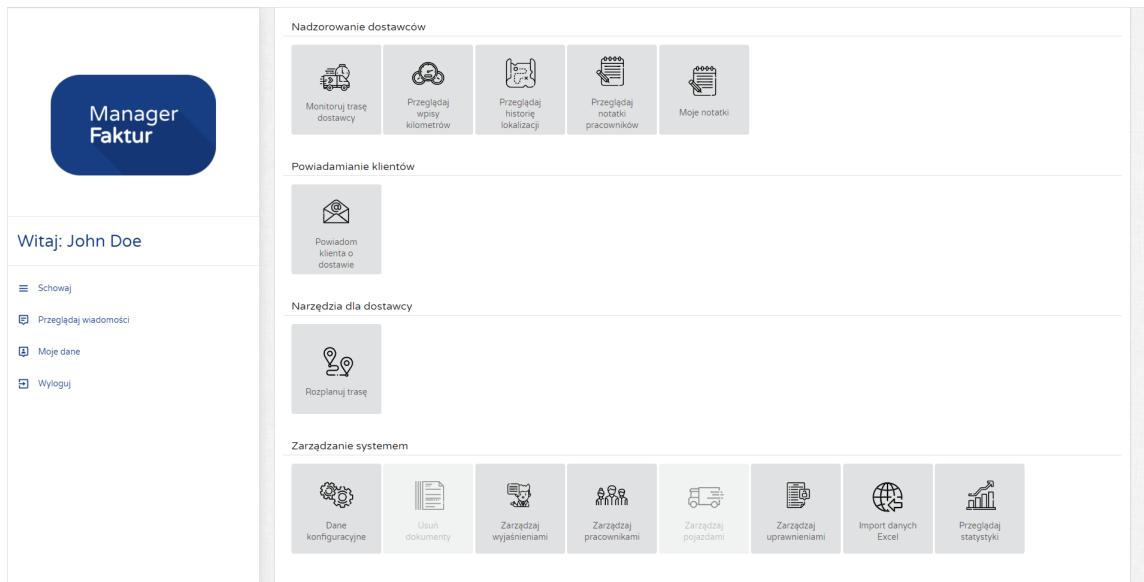
Omówiony fragment skryptu archiwizującego zilustrowany został na listingu ??.

4.8. Interfejs użytkownika

W niniejszym rozdziale, przedstawiono widoki gotowego interfejsu użytkownika obu aplikacji klienckich. W związku z mnogością, utworzonych elementów interfejsu, zdecydowano się na prezentację, tylko tych kluczowych.



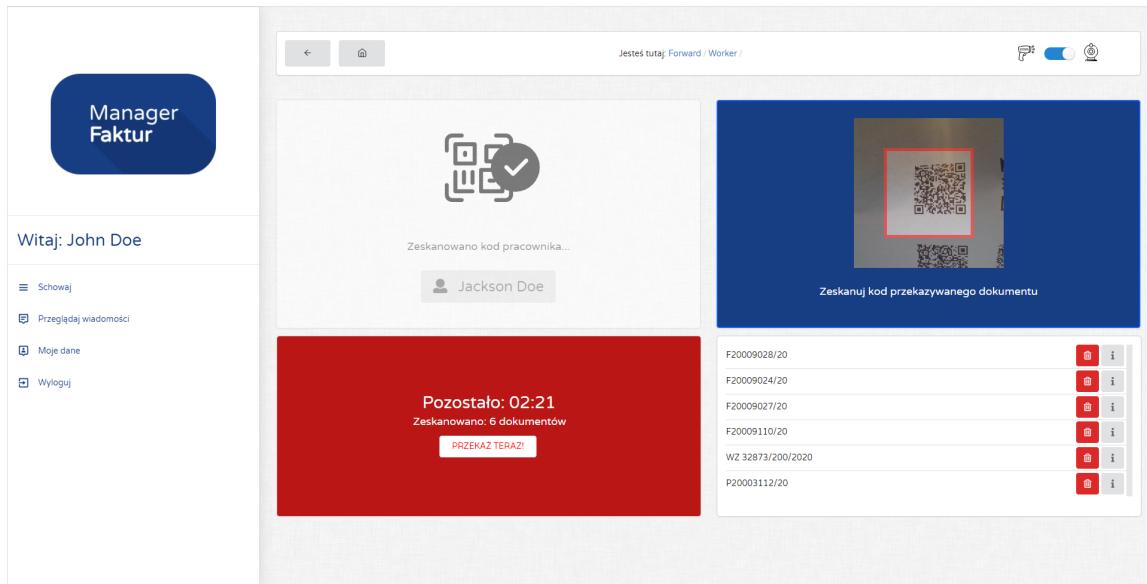
Rys. 4.1: Widok panelu głównego aplikacji webowej



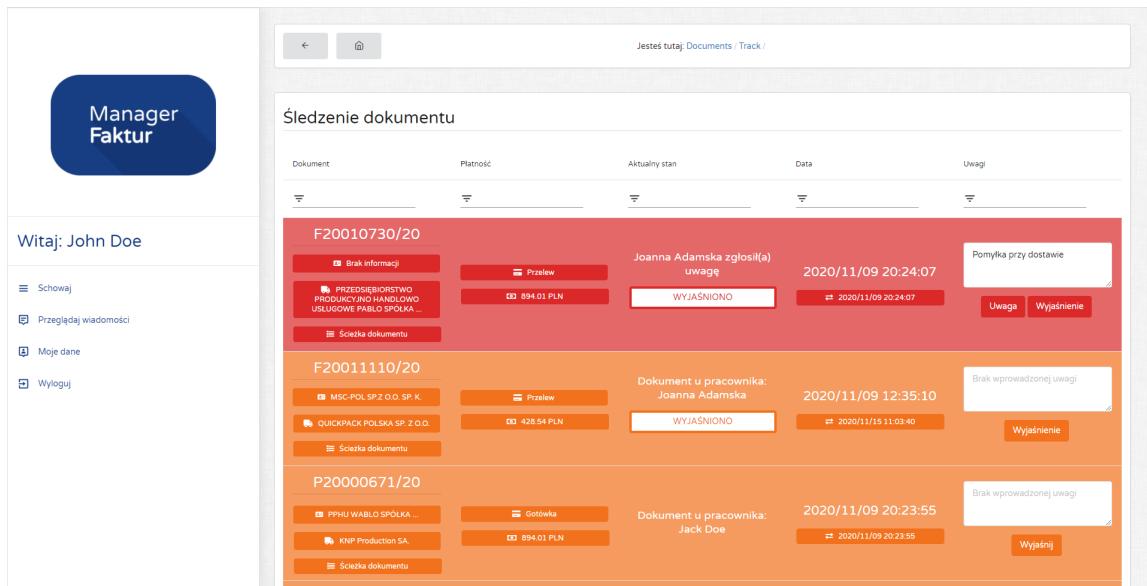
Rys. 4.2: Widok panelu głównego aplikacji webowej (cz 2.)

Na ilustracjach 4.1 i 4.2 ukazano widok panelu głównego aplikacji webowej. Użytkownik przechodzi z nich do innych widoków, wybierając określony przycisk funkcjonalności. Przyciski widoków mogą być aktywne lub niedostępne, w zależności od posiadanych przez pracownika uprawnień.

Na rysunku 4.3 widzimy moduł przekazywania dokumentów pomiędzy pracownikami. W momencie tworzenia tej ilustracji, zeskanowany został kod pracownika, a wczytywany zostaje bąkod QR dokumentu. Poniżej widoku kamery, zlokalizowane są dwa kontenery, prezentujące czas pozostały do zakończenia procedury skanowania oraz listę wczytanych dokumentów.



Rys. 4.3: Widok funkcjonalności przekazywania dokumentów pracownikowi

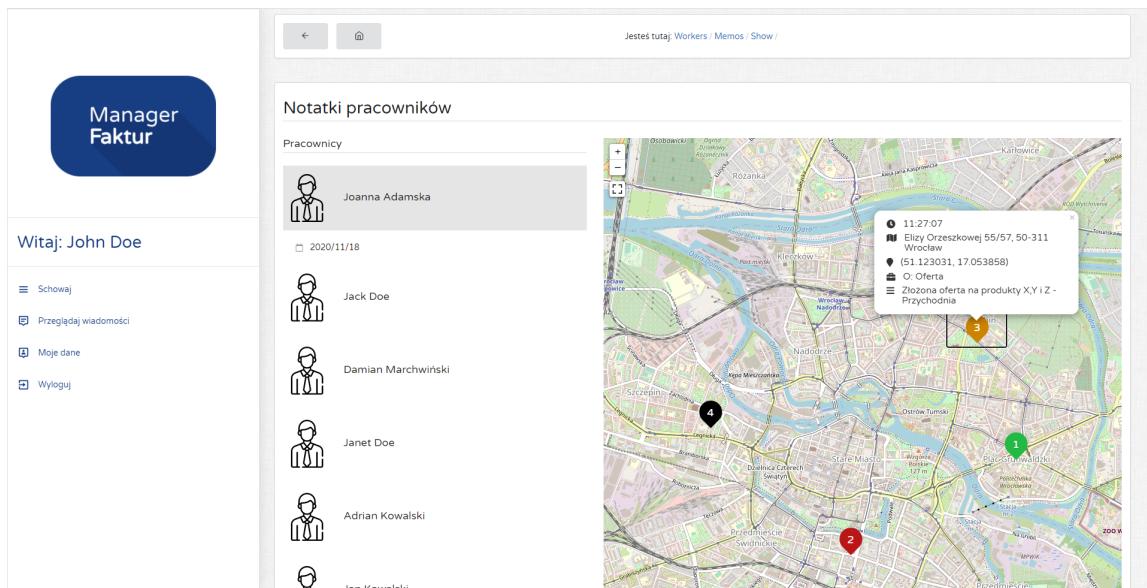


Rys. 4.4: Widok funkcjonalności śledzenia dokumentów

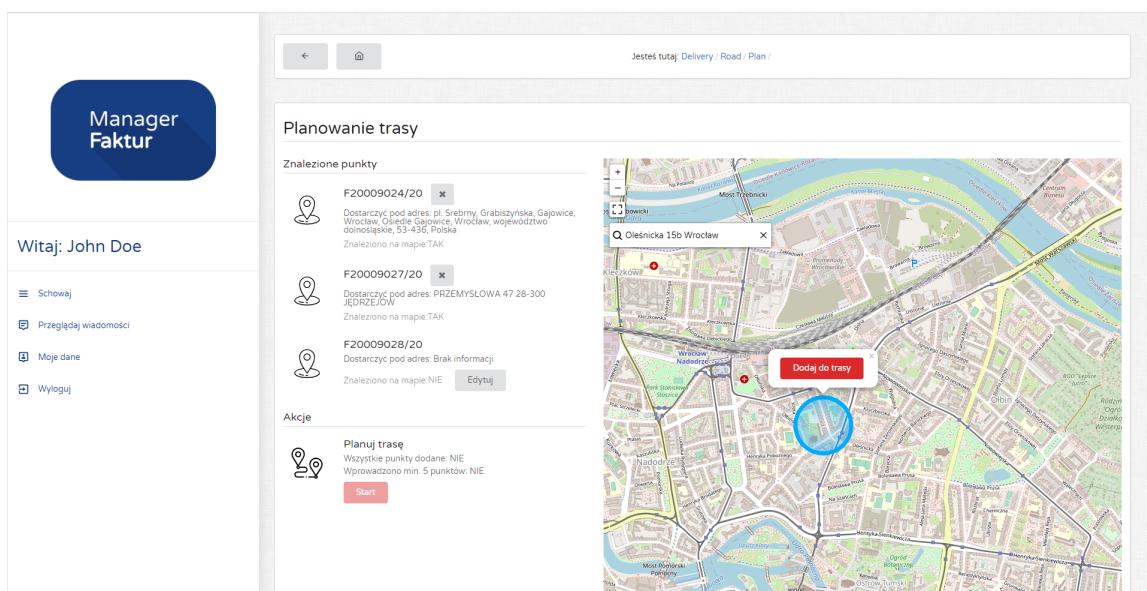
Ilustracja 4.4 obrazuje widok śledzenia dokumentów. W każdym wierszu tabeli, prezentowane są informacje o dokumencie, jego aktualnym statusie w systemie, a także wszystkich akcjach na nim wykonanych. Użytkownik, posiadający uprawnienia dostępu do tego widoku, może ponadto wprowadzić notę wyjaśniającą dla dokumentu (przycisk „Wyjaśnij”), co spowoduje archiwizację faktury po zakończeniu dnia pracy.

Przedstawiona grafika 4.5 ukazuje widok modułu przeglądania notatek handlowych. Użytkownik, w ramach tego widoku, po wybraniu pracownika oraz daty utworzenia notatek, może przeglądać wszystkie wprowadzone przez pracownika wpisy informacyjne. W zależności od typu wpisu, ukazane na mapie punkty lokalizacyjne, charakteryzują się odmienną kolorystyką. Po wybraniu określonego punktu na mapie, widoczne są dane notatki dla tej lokalizacji.

Na ilustracji 4.6, widoczny jest element segmentu planowania trasy dostawy. Po lewej stronie widoku, dostępne są wszystkie dokumenty przypisane do zalogowanego pracownika, natomiast po prawej stronie, znajduje się fragment mapy. Mapa ta, wykorzystywana jest do wyboru loka-

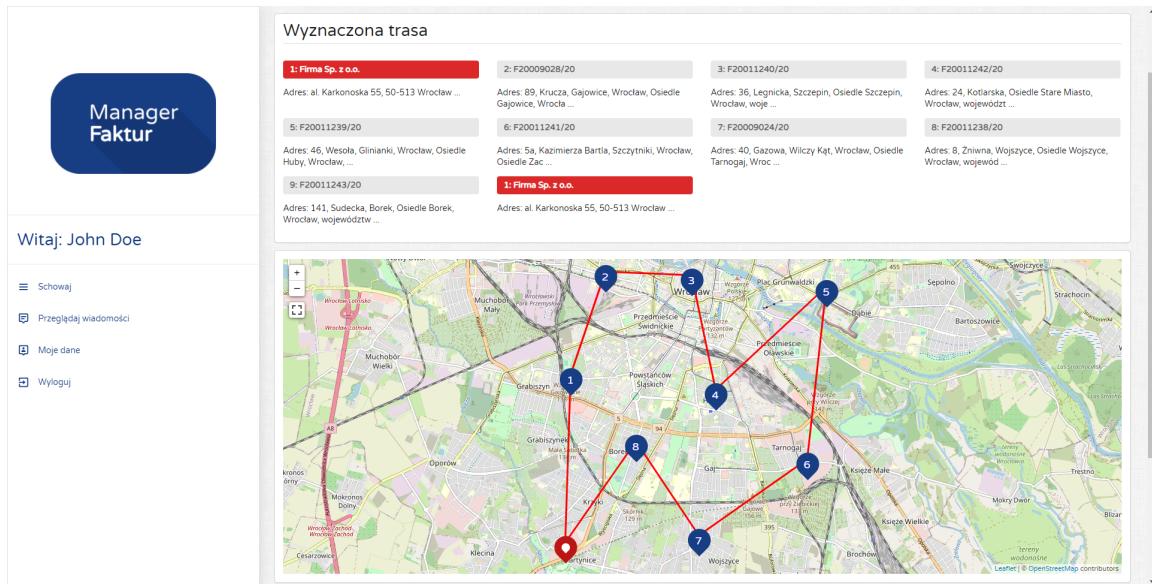


Rys. 4.5: Widok funkcjonalności przeglądania notatek pracowników handlowych



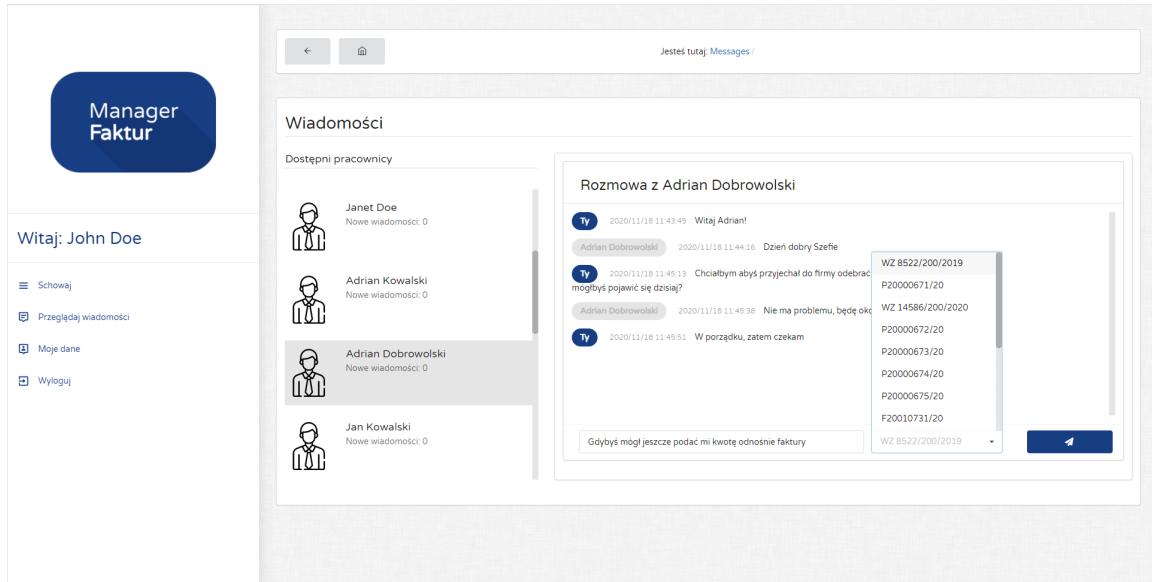
Rys. 4.6: Widok funkcjonalności planowania trasy dostawy - wybór lokalizacji

lizacji dostawy dla dokumentu, w sytuacji gdy dane faktury nie pozwolą na jej automatyczne uzyskanie.



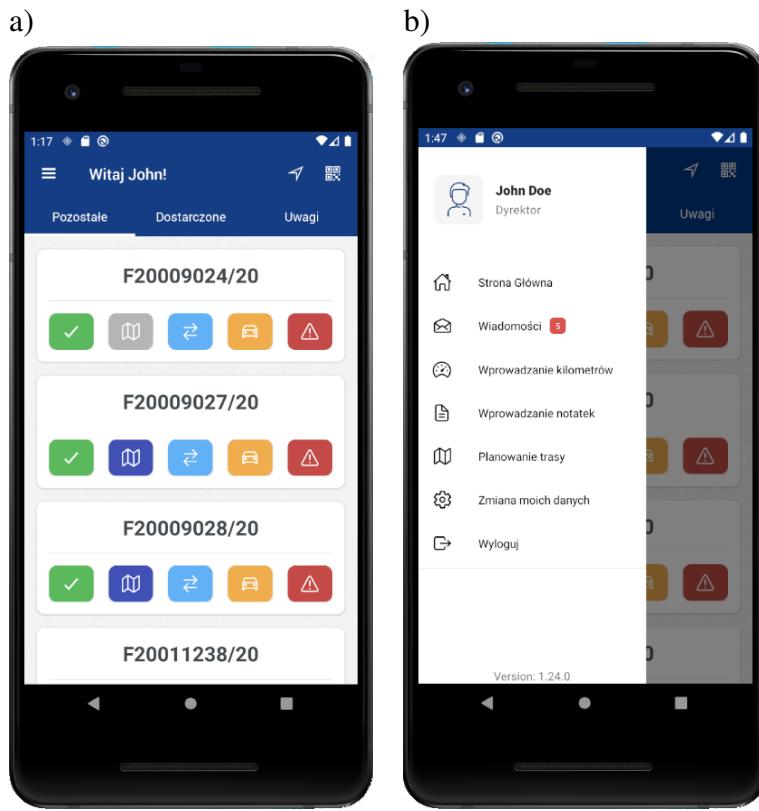
Rys. 4.7: Widok funkcjonalności planowania trasy dostawy - rezultat

Po wyborze wszystkich punktów lokalizacyjnych, uruchamiany jest algorytm wyliczania trasy dostawy. Z chwilą otrzymania wyniku działania algorytmu, ukazuje się widok przedstawiony na ilustracji 4.7. Widzimy w nim listę kolejnych lokalizacji na drodze dostawy, a także mapę unaoczniającą sekwencję odwiedzanych punktów. Po wybraniu dowolnego z punktów, wyświetlna jest informacja o powiązanym dokumencie oraz danych adresowych.



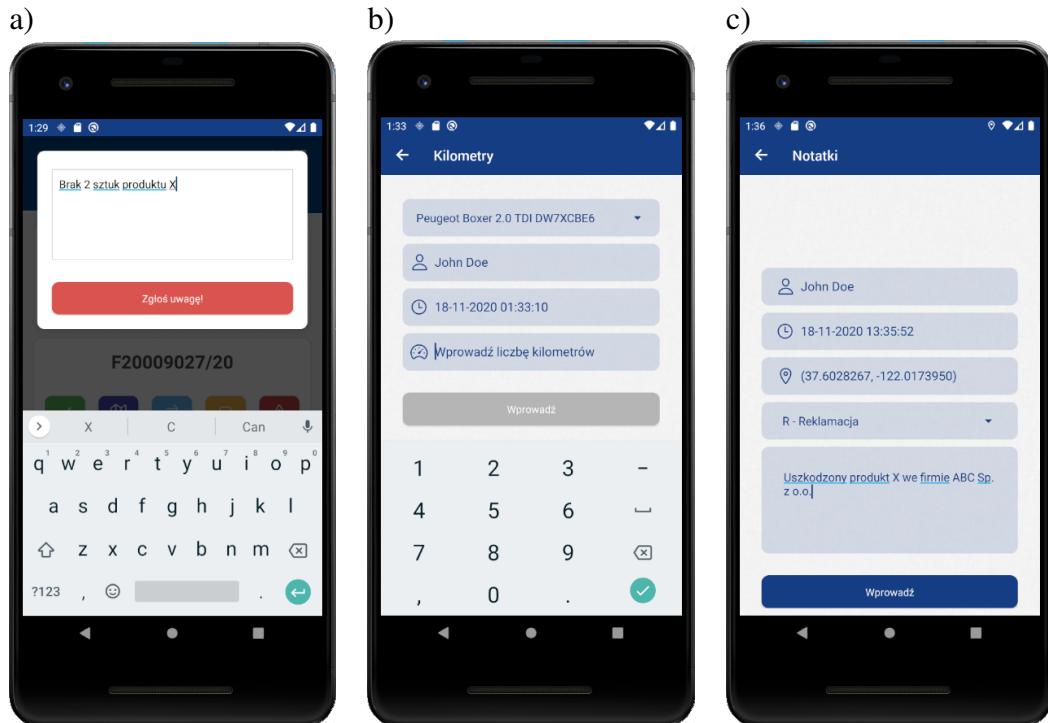
Rys. 4.8: Widok funkcjonalności komunikatora tekstowego

Rysunek 4.8, prezentuje wygląd widoku wewnętrznego komunikatora tekstowego. Wyróżnić możemy w nim listę pracowników, z którymi chcemy rozpoczęć konwersację, a także okno wysyłanych wiadomości. Poza możliwością wprowadzania tekstu wiadomości, istnieje także funkcjonalność wyboru dokumentu, o którym chcemy „wspomnieć” w wiadomości. Wprowadzenie wspomnienia dla faktury, skutkuje wyświetleniem adekwatnej informacji, w widoku śledzenia dokumentów.



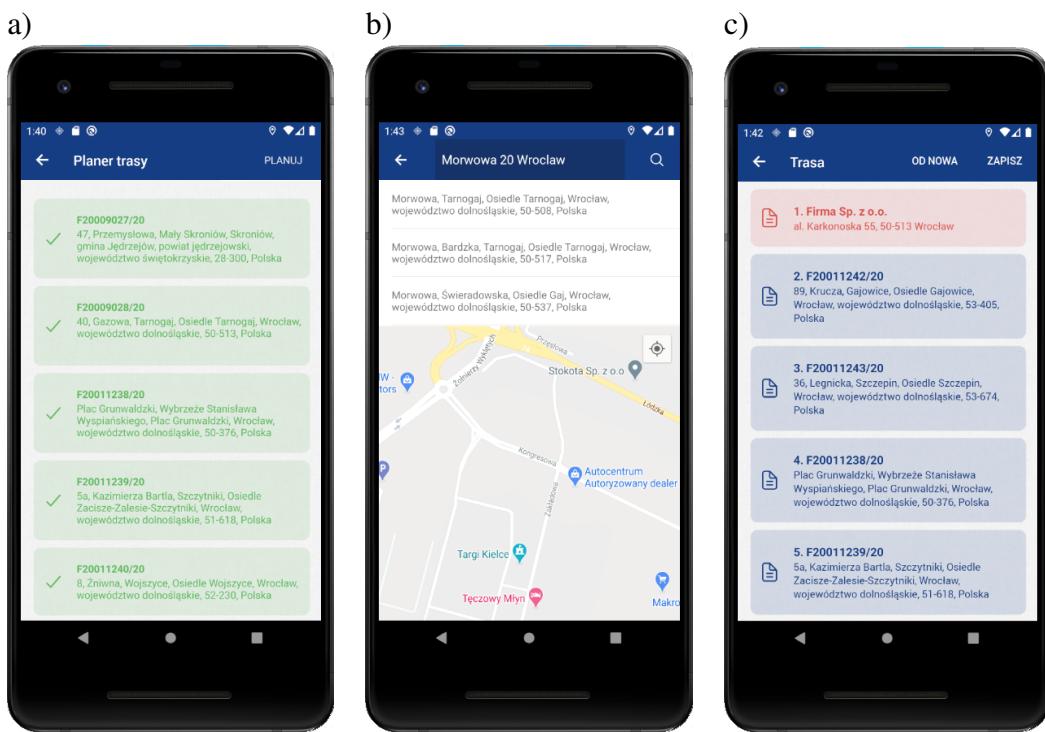
Rys. 4.9: Aktywność główna aplikacji mobilnej a) lista dokumentów b) panel boczny

Na ilustracjach 4.9a oraz 4.9b, przedstawiona została aktywność główna aplikacji mobilnej. Pierwsza z grafik, ukazuje widok listy dokumentów, ładowany domyślnie, po zalogowaniu się do systemu. Na drugim rysunku natomiast, widzimy rozsuwany panel boczny, zawierający większość z dostępnych funkcjonalności aplikacji.



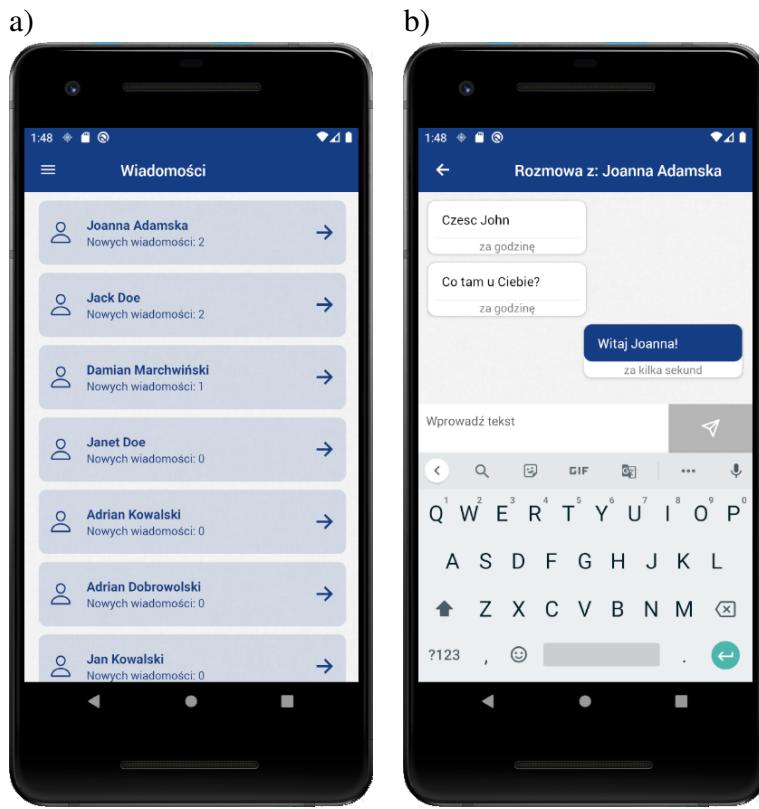
Rys. 4.10: Aktywności wprowadzania danych a) zgłoszanie uwagi b) wpis kilometrów c) wpis notatki

W ramach rysunków 4.10a, 4.10b oraz 4.10c, ukazane zostały aktywności wprowadzania danych w aplikacji mobilnej. Pierwsza z nich, tyczy się zgłoszenia uwagi dotyczącej dokumentu. Użytkownik, po naciśnięciu czerwonego przycisku, znajdującego się przy fakturze, wprowadza zawartość notatki uwagi, do przedstawionego pola tekstopowego. Druga aktywność, zdefiniowana została w celu generowania adnotacji dotyczących liczników kilometrów, w pojazdach firmy. W widoku tym, jeżeli do użytkownika przypisany jest pojazd domyślny, zostaje on wybrany z listy rozwijanej automatycznie. Trzecia z aktywności, wykorzystywana jest do wprowadzania wpisów dotyczących notatek handlowych.



Rys. 4.11: Aktywności planowania trasy dostawy a) lista dokumentów b) wybór lokalizacji c) rezultat obliczeń

Na grafikach 4.11a, 4.11b oraz 4.11c, widzimy widoki aplikacji mobilnej, dotyczące wyznaczania trasy dostawy. W pierwszym widoku, zawarta jest lista przypisanych do pracownika dokumentów. Po wybraniu jednej z faktur, jeżeli nie posiada ona przypisanej automatycznie lokalizacji, przechodzimy do aktywności drugiej. W ramach tej aktywności, użytkownik może wyszukać dowolną lokalizację, a następnie przypisać ją do dokumentu. Widok trzeci natomiast, reprezentuje rezultat, w postaci ustalonej trasy dostawy. Po kliknięciu przycisku „zapisz”, użytkownik kierowany jest do panelu głównego, a dokumenty w nim zawarte, sortują się w kolejności zgodnej z trasą dostawy.



Rys. 4.12: Aktywność komunikatora aplikacji mobilnej a) lista pracowników b) wymiana wiadomości

Ostatnie z przedstawionych grafik (tj. grafika 4.12a oraz 4.12b), ukazują funkcjonalność wewnętrznego komunikatora tekstowego aplikacji mobilnej. Wyróżnić możemy tutaj widok listy pracowników, w którymi można rozpoczęć konwersację, a także komponent przedstawiający rozmowę z konkretnym użytkownikiem.

Rozdział 5

Badania eksperymentalne

5.1. Uzyskane efekty pracy

Z chwilą przystąpienia do realizacji oprogramowania, planowano stworzenie systemu zarządzania dokumentami w obrębie przedsiębiorstwa. System ten, miał składać się z aplikacji klienckich (tj. aplikacji webowej oraz mobilnej), a także interfejsu API.

W ramach strony webowej, użytkownik, po poprawnym uwierzytelnieniu oraz autoryzacji, powinien posiadać możliwości: przekazywania dokumentów, śledzenia ich aktualnego statusu, nadzorowania dostawców, przeglądania faktur archiwalnych, powiadamiania klientów o przetwarzaniu zamówienia, wyznaczania drogi dostawy towarów, prowadzenia wymiany wiadomości tekstowych z innymi użytkownikami, a także zarządzania zbiorami danych dotyczących konfiguracji systemu.

Ponadto, w obszarze aplikacji mobilnej, osoba korzystająca z programu, powinna móc: przekazywać dokumenty, wprowadzać uwagi dotyczące faktur, powiadamiać klienta o dostawie, wyznaczać trasę dostawy, wprowadzać wpisy dotyczące kilometrów pojazdu, definiować notatki handlowe, a także prowadzić konwersacje z wykorzystaniem komunikatora.

W kontekście interfejsu API, zaimplementowany miał zostać wydajny, a także bezpieczny system pozyskiwania oraz modyfikacji wszystkich danych, zawartych w ramach oprogramowania, posiadający mechanizmy uwierzytelniania oraz autoryzacji.

Wszystkie z przedstawionych powyżej założeń, zostały pomyślnie zaimplementowane w ramach stworzonego projektu. Dzięki doborowi odpowiednich narzędzi przed rozpoczęciem tworzenia oprogramowania, praca nad nim przebiegała w sposób systematyczny, a także efektywny. Podjęta decyzja, związana z wyborem bibliotek stylów Semantic-UI, a także Native-Base, odpowiednio dla aplikacji webowej oraz mobilnej, skutkowała powstaniem klarownych, łatwych w obsłudze i modułowych interfejsów użytkownika obu aplikacji.

Ponadto, wybór technologii .Net Core, jako środowiska dla interfejsu programowania aplikacji, przyczynił się do budowy skalowalnego, wydajnego i bezpiecznego interfejsu API, który może być z łatwością rozszerzany o nowe funkcjonalności.

Wszystkie z tworzonych elementów systemu, zostały sprawdzone w kontekście ewentualnych błędów w kodzie programu.

5.2. Możliwości dalszego rozwoju

Struktury i interfejsy graficzne aplikacji webowej oraz mobilnej, stworzone zostały w taki sposób, aby umożliwić łatwe wprowadzanie nowych funkcjonalności, w zależności od potrzeb użytkownika docelowego.

Jedną z takich potrzeb, może być moduł efektywnego pakowania pojazdu. W ramach tego rozwiązania, system określa które z dokumentów, w zależności od ich zbiorów produktów, mogą być zapakowane do pojazdu, aby maksymalnie wypełnić jego przestrzeń ładunkową. Z myślą o tej perspektywie, w modelu danych, zdefiniowane zostały atrybuty wymiarów powierzchni ładunkowej dla każdego z pojazdów.

Interesującą możliwością dalszego rozwoju oprogramowania jest implementacja algorytmu wyznaczania trasy dostawy, na podstawie informacji dotyczących ruchu drogowego. W przypadku takiego algorytmu, aplikacja identyfikuje ulice po których będzie poruszał się pojazd, a także pobiera dla nich informacje związane z aktualnym ruchem ulicznym, po czym uwzględnia je w kalkulacjach optymalnej trasy. Dzięki realizacji części serwerowej systemu, jako interfejs programowania aplikacji bazujący na punktach końcowych, proces tworzenia algorytmu może być realizowany w całkowicie odseparowanym środowisku. Po wykonaniu testów poprawności działania algorytmu, może on zostać w prosty sposób zintegrowany z aplikacją, poprzez wprowadzenie jego kodu wewnątrz endpointu.

Dodatkowo, w przypadku wykorzystania oprogramowania w kontekście komercyjnym, zastosować można wybór płatnego dostawcy danych lokalizacyjnych (np. Google Maps), w celu zwiększenia ich wiarygodności oraz dokładności. Chcąc zastosować takie rozwiązanie w przygotowanym systemie, wystarczy wykupić usługę interfejsu lokalizowania od dostawcy, a następnie wprowadzić odpowiedni adres do tej usługi jako wartość właściwości dostawcy danych dla komponentów mapy.

Rozdział 6

Podsumowanie

Literatura