# Wstęp

Integracja systemów to dziedzina IT skupiająca się na komunikacji między systemami informatycznymi. Komunikacja jest tutaj rozumiana jako przekazywanie informacji, sterowanie działaniem lub udostępnianie danych. Systemem z kolei jest każda aplikacja czy zasób posiadający konkretne przeznaczenie i pełniący określone funkcje. Zagadnienie integracji jest szczególnie istotne przy projektowaniu, wdrażaniu i rozbudowywaniu rozległych rozwiązań informatycznych o budowie modułowej. W przemyśle lub nauce mamy do czynienia z wieloma klasami systemów: wsparcia produkcji, planowania zasobów; zarządzania dostawami. Każdy z nich pracuje niezależnie, obejmując dedykowany sobie obszar działalności. W środowiskach składających się z wielu niezależnych systemów kluczowe jest zapewnienie ich kompatybilności i interoperacyjności[[1]](#footnote-1).

Najlepszym obecnie znanym i szeroko stosowanym sposobem spełnienia powyższych wymagań jest zastosowanie wzorca architektury zorientowanej na usługi (ang. *Service Oriented Architecture,* w skrócie SOA). Zgodnie z tym wzorcem systemy informatyczne są traktowane jako byty udostępniające swojemu otoczeniu określone funkcjonalności. Funkcjonalności te, jeśli spełniają formalne kryteria, nazywane są w terminologii SOA usługami. Usługa jest centralnym pojęciem w koncepcji SOA, zgodnie z definicją pochodzącą z opracowania *Soa Source Book* [1] opublikowanego przez The Open Group: *jest to logiczna reprezentacja powtarzalnej aktywności, ma określony rezultat, jest samowystarczalna, może składać się z innych usług, jest czarną skrzynką dla swoich konsumentów.* Na potrzeby pracy autor proponuje rozszerzoną i bardziej ścisłą definicję, skupiającą się na aspektach technicznych projektowania architektury i implementowania aplikacji: usługa to element oprogramowania posiadający interfejs, za pomocą którego udostępnia zdefiniowane funkcjonalności. Interfejs usługi jest udokumentowany w formie jednoznacznego kontraktu. Interfejs oferuje minimalny zakres informacji niezbędny do używania usługi, ukrywając szczegóły implementacji. Interfejs jest możliwie generyczny w celu zapewnienia reużywalności usługi. Usługa działa autonomicznie, wykorzystując przydzielone zasoby i infrastrukturę. Usługa nie utrzymuje stanu i nie przechowuje informacji w trakcie działania. Usługa może być skomponowana z innych usług. Usługa realizuje luźne powiązanie między integrowanymi systemami[[2]](#footnote-2), zapewniając warstwę abstrakcji logicznej i minimalizując zależności techniczne. Usługa jest osadzona na wspólnym, jednorodnym medium komunikacyjnym zapewniającym przepływ informacji oraz umożliwiającym katalogowanie i wykrywanie istniejących usług.

Według opracowania *Architecting Service-Oriented Systems* [2] wydanego przez Software Engineering Institute (Uniwersytet Carnegie Mellon) infrastruktura wspierająca SOA składa się z poniższych głównych komponentów. Tworzą one warstwę oprogramowania pośredniczącego (ang. *Middleware software*):

**Magistrala serwisowa** (ang. *Service bus, Message bus*) jest kontenerem, w którym pracuje kod usług. Odpowiada za dopasowanie transportów i protokołów komunikacyjnych między interfejsami systemów. Wspiera implementację wzorca *VETRO*[[3]](#footnote-3) udostępniając gotowe narzędzia i biblioteki pomocnicze wykorzystywane przez oprogramowanie usług. Zapewnia monitorowanie, konfigurację i administracyjne zarządzanie pracą usług.

**Broker komunikacyjny** (ang. *Messaging system, Message broker*) odpowiada za asynchroniczną komunikację między usługami lub usługami i systemami zewnętrznymi, zapewniając ich luźne powiązanie. Obsługuje złożone modele wymiany wiadomości[[4]](#footnote-4) z wykorzystaniem topiców i kolejek[[5]](#footnote-5) na bazie których magistrala serwisowa realizuje wzorce dostarczania komunikatów jak na przykład publikacja-subskrypcja (ang. *publish-subscribe*).

**Rejestr usług** (ang. *Service registry, Service repository*) jest narzędziem do katalogowania usług mającym zapewnić ich wykrywalność i reużywalność. SOA nie precyzuje formy rejestru, popularnym standardem jest rejestr UDDI (ang. *Universal Description Discovery and Integration*)[[6]](#footnote-6). Najważniejsze funkcjonalności rejestru to: kontrola zależności między usługami, kontrola dostępu konsumentów do usług, wersjonowanie i wyszukiwanie usług, utrzymywanie spójności i aktualności ich kontraktów.

Opisane wyżej komponenty magistrali usług i brokera komunikacyjnego wchodzą w skład specyfikacji oryginalnie zaprezentowanej przez Gregora Hohpe i Bobby’ego Woolfa jako „Enterprise Integration Patterns”, następnie rozwijanej m.in. przez Thomasa Erla w „SOA Design Patterns” oraz Arnona Rotem-Gal-Oza w „SOA Patterns” a obecnie propagowanej przez serwis współtworzony przez Thomasa Erla „*soapatterns.org”*. Specyfikacja ta zawiera wzorce architektoniczne i projektowe, opracowane na bazie doświadczeń autorów z wczesnych implementacji SOA, służące powielaniu dobrych praktyk, standaryzacji i optymalizacji rozwiązań. W dalszej części pracy autor będzie się wielokrotnie odwoływał do wspomnianych wzorców, uzasadniając wybór sposobu rozwiązania konkretnego problemu i wskazując zalety danego podejścia.

Istotnym wzorcem wymagającym osobnego omówienia jest Kanoniczny model danych (ang. *Canonical data model, Canonical schema*). Wzorzec ten ma zastosowanie w sytuacji kiedy integrowane systemy wymieniają się danymi o takiej samej zawartości informacyjnej ale posługują się różną terminologią, formatami i strukturami danych. Kanoniczny model danych definiuje wspólne, generyczne encje i relacje między nimi[[7]](#footnote-7). Zadaniem usługi realizującej ten wzorzec jest transformacja komunikatu z modelu konsumenta do modelu kanonicznego, wykonanie zaprogramowanych czynności i finalnie transformacja z modelu kanonicznego do modelu dostawcy usługi. Gwarantuje to luźne powiązanie na poziomie wymiany danych (brak bezpośrednich zależności dzięki abstrakcyjnej warstwie pośredniej) oraz chroni przed mnożeniem bytów – dana encja jest zdefiniowana raz i następnie może być ponownie używana przez wszystkie pozostałe usługi.

Poniżej zostały opisane i poparte przykładami najważniejsze zalety stosowania SOA jako sposobu mierzenia się z kluczowymi wyzwaniami integracji. Zalety te w głównej mierze wynikają ze świadomego i starannego trzymania się opisywanych standardów i wzorców, bez czego SOA byłaby jedynie zbiorem pustych haseł i teoretycznych koncepcji bez odniesienia do realnych problemów.

**Brak zależności i modułowość** – możliwe jest modyfikowanie lub nawet wymiana poszczególnych komponentów architektury bez wpływu na całość rozwiązania. Dzięki warstwie pośredniczącej i abstrakcyjnemu modelowi danych zmiana w jednym systemie (z ewentualną zmianą w kodzie usługi, ale bez modyfikacji jej interfejsu) nie jest zauważalna dla otoczenia. Przykładowo zmiana struktury danych w systemie dawcy będzie wymagać tylko zmiany transformacji z modelu kanonicznego usługi, co nie wpływa w żaden sposób na wystawiany interfejs a zatem na jej konsumentów.

**Elastyczność doboru technologii** – brak ograniczeń dla platform, języków programowania czy środowisk uruchomieniowych przy podłączaniu nowych systemów. Dzięki luźnemu powiązaniu i braku zależności możliwe jest integrowanie systemów opartych o różne standardy techniczne. W przypadku konieczności wymiany zintegrowanej bazy danych, np. z Oracle na MySQL, konieczne jest dostosowanie adapterów bazodanowych w usługach i ewentualne modyfikacje transformacji w celu dopasowania do docelowej struktury. Zmiany takie mogą zostać wdrożone bez angażowania systemów konsumentów korzystających z usług opartych o tę bazę danych.

**Optymalizacja zasobów** – raz dostarczona usługa może być wykorzystywana przez wielu konsumentów, nowa funkcjonalność może powstać tylko dzięki wykorzystaniu istniejących usług. Reużywalność (ang. *reusability*) i komponowalność (ang. *composability*) usług to cechy zapobiegające mnożeniu bytów i duplikowaniu powiązań między systemami. Można przytoczyć scenariusz w którym została dostarczona i jest zarejestrowana usługa weryfikująca istnienie konta użytkownika o danym nazwisku. Inni klienci, zainteresowani taką funkcjonalnością, pozyskują usługę z katalogu i zaczynają jej reużywać, ponosząc minimalne nakłady na podłączenie do interfejsu usługi. Następnie taka usługa może wchodzić w skład złożonej kompozycji realizującej np. rejestrowanie i walidowanie kont użytkowników.

**Bezpieczeństwo komunikacji** – interfejsy systemów są odseparowane od otoczenia poprzez integracyjną warstwę pośredniczącą, na której działają zdefiniowane w danej organizacji polityki bezpieczeństwa. Na przykład usługa zwracająca dane finansowe komunikuje się z systemem dostawcą za pomocą szyfrowanego protokołu SSH, jest udostępniona tylko autoryzowanym konsumentom, którzy uwierzytelniają się odpowiednimi certyfikatami klienckimi.

**Stabilność i dostępność** – strategiczna rola warstwy pośredniczącej w zintegrowanym środowisku wymaga zapewnienia wydajności, stabilności, wysokiej dostępności i gwarancji dostarczenia (ang. *Reliable messaging, Guaranteed delivery*). Zachowanie bezstanowości i autonomiczności usług wspiera wysoką wydajność całej infrastruktury SOA. Użycie wzorców komunikacji asynchronicznej (np. z rozproszoną persystentną kolejką JMS) gwarantuje dostarczenie komunikatów do odbiorców. Funkcjonalności dławika (ang. *throttler*) i pamięci podręcznej (ang. *cache*) odpowiadają za obronę infrastruktury SOA oraz systemów dostawców przed przeciążeniami wynikającymi z niestabilnego ruchu generowanego przez konsumentów.

**Monitorowanie i zarządzanie** – obecność warstwy pośredniczącej umożliwia centralne nadzorowanie i administrację ruchem między zintegrowanymi systemami. Możliwe jest logowanie metadanych usług i treści komunikatów oraz nakładanie i weryfikowanie spełnienia gwarantowanych poziomów świadczenia usług (ang. *Service Level Agreement*, w skrócie SLA). Zachowanie autonomiczności usług umożliwia ich niezależne uruchamianie i dezaktywowanie.

## Cel i zakres projektu

Celem niniejszej pracy jest zaprojektowanie i wykonanie prototypu aplikacji obsługującej wymianę informacji w złożonym i niejednorodnym środowisku informatycznym. Praca ma na celu wykazanie w praktyce zasadności i skuteczności stosowania architektury SOA w tej klasie problemów. Wykonana aplikacja ma pełnić rolę magistrali serwisowej, będącej warstwą pośredniczącą w integracji za pomocą usług. Aplikacja ma zostać zbudowana w środowisku wspierającym implementację wzorców projektowych SOA oraz dostępnym na zasadach otwartego oprogramowania (ang. *Open Source*).

W zakresie realizowanego projektu jest: skonfigurowanie i uruchomienie komponentu magistrali serwisowej; zaprojektowanie, wytworzenie i osadzenie na magistrali kodu usług integracyjnych; skonfigurowanie brokera komunikacyjnego i zintegrowanie go z magistralą; skonfigurowanie zaślepek systemów klienckich umożliwiających testowanie usług; wytworzenie i uruchomienie zasobów i zaślepek reprezentujących systemy dostawców.

## Temat projektu

Przedmiotem projektu jest warstwa integracyjna dla systemu obsługującego w formie elektronicznej bilety uprawniające do parkowania pojazdu oraz kontrole pojazdów. System umożliwia zarejestrowanym użytkownikom – kierowcom uiszczenie opłaty za postój za pomocą aplikacji klienckiej, bez konieczności korzystania z parkomatu. Dostępne są różne formy naliczania opłaty za parking: przez określony z góry czas, do podanej godziny oraz za określoną kwotę; możliwe jest wcześniejsze zakończenie parkowania. System umożliwia użytkownikom – kontrolerom weryfikację pojazdu. W przypadku braku opłaconego parkowania system wystawia mandat obciążający konto użytkownika – kierowcy kontrolowanego pojazdu. System składa się z aplikacji klienckiej kierowcy, aplikacji kontrolera, bazy danych użytkowników i biletów parkingowych, aplikacji księgowej nadzorującej naliczanie opłat i mandatów oraz aplikacji parkingowej nadzorującej parkowania i kontrole.

## Struktura pracy

Praca składa się z czterech rozdziałów:

**Rozdział 1 „Wybór technologii”** stanowi przegląd aktualnie dostępnych technologii integracyjnych spełniających kryteria postawione w celach i zakresie pracy; zawiera opis wybranych rozwiązań i narzędzi z uzasadnieniem podjętych decyzji.

**Rozdział 2 „Specyfikacja rozwiązania”** specyfikuje wymagania funkcjonalne i niefunkcjonalne dla rozwiązania; zawiera projekt techniczny rozwiązania w zakresie: architektury logicznej otoczenia integracyjnego, punktów styku między systemami, definicji kanonicznego modelu danych, kontraktów usług.

**Rozdział 3 „Realizacja projektu”** jest technicznym opisem konfiguracji komponentów użytych w projekcie; zawiera opis implementacji usług z fragmentami kodów źródłowych; szczegółowo prezentuje zastosowane wzorce i rozwiązania napotkanych problemów; zawiera raporty z testów funkcjonalnych i wydajnościowych aplikacji.

**Rozdział 4 „Podsumowanie prac”** opisuje główne problemy napotkane w trakcie realizacji projektu; zawiera propozycje dalszych prac rozwojowych; prezentuje wnioski i podsumowanie wykonanej pracy.

# Wybór technologii

Wybór zestawu technologii został poprzedzony oceną dostępnych rozwiązań pod kątem spełnienia kryteriów zawartych w celach i zakresie pracy. W pierwszej kolejności, ze względu na wymaganie dostępności na zasadach otwartego oprogramowania, zostały wyeliminowane rozwiązania licencjonowane komercyjnie (takie jak Microsoft BizTalk, Oracle Service Bus, Tibco BusinessWorks, webMethods Integration) oraz wersje społecznościowe z wyłączeniem użytkowania produkcyjnego (Mule ESB, Red Hat JBoss Fuse, Talend ESB). Analizie porównawczej zostały poddane następujące rozwiązania udostępniane na zasadach wolnego oprogramowania zgodnie z licencjami ASL[[8]](#footnote-8) lub CDDL[[9]](#footnote-9):

**WSO2 ESB** należy do stosu technologii integracyjnych autorstwa firmy WSO2, bazujących na projektach rozwijanych przez Apache Software Foundation[[10]](#footnote-10). WSO2 ESB bazuje na projekcie Apache Synapse - magistrali serwisowej opartej o silnik usług sieciowych Apache Axis2. WSO2 Message Broker jest brokerem komunikacyjnym obsługującym protokół AMQP (ang. *Advanced Message Queueing Protocol*) bazującym na projekcie Apache Qpid.

**Open ESB[[11]](#footnote-11)** jest magistralą serwisową autorstwa Sun Microsystems, obecnie rozwijaną przez otwartą społeczność ze wsparciem firmy Pymma Consulting. Architektura magistrali bazuje na serwerze aplikacyjnym GlassFish i standardzie JBI (ang. *Java Business Integration*). JBI definiuje komponenty interfejsowe (*Binding Component*) odpowiadające za komunikację z wykorzystaniem różnorodnych protokołów i transportów (np. HTTP, FTP, JMS) oraz komponenty funkcjonalne (*Service Engine*) przetwarzające komunikaty (np. XSLT, BPEL, EJB). Rekomendowanym brokerem komunikacyjnym jest serwer kolejek OpenMQ osadzony standardowo na serwerze GlassFish.

**Apache ServiceMix[[12]](#footnote-12)** jest platformą złożoną z popularnych technologii integracyjnych: Apache CXF jako silnik usług sieciowych (ang. *Web Service*) z wsparciem API Java (JAX-WS dla protokołu SOAP oraz JAX-RS dla RESTful API); Apache Camel jako silnik routingu, framework dla integracyjnych wzorców wymiany danych i biblioteka konektorów; broker komunikacyjny Apache ActiveMQ obsługujący protokół AMQP, wykorzystujący API JMS (ang. *Java Message Service*); Apache Karaf jako kontener aplikacyjny Java zgodny ze specyfikacją OSGI (ang. *Open Services Gateway Initiative*).

Wszystkie powyższe rozwiązania pełnią podobne funkcje i są odpowiednimi narzędziami do wykonania zakładanego projektu. Różnic między nimi należy upatrywać w bazowych technologiach i standardach, o które są oparte, a co za tym idzie – w potencjalnych ograniczeniach technicznych oraz odmiennych filozofiach realizacji zadań integracyjnych. Z perspektywy autora projektu istotne są różnice w takich aspektach jak dostępność dedykowanych konektorów, czytelność kodu, wsparcie społeczności i jakość dokumentacji. Te i inne kryteria zostały szerzej omówione w poniższych punktach.

**Wsparcie wzorców integracyjnych** - zarówno WSO2 jak i Apache Camel dokumentują pełne wsparcie wzorców integracyjnych[[13]](#footnote-13). Autor nie udało się dotrzeć do aktualnych źródeł opisujących to zagadnienie w przypadku Open ESB.

**Otwartość architektury** - wszystkie omawiane rozwiązania bazują na technologii Java. Poszczególne komponenty składowe platform WSO2 i ServiceMix mogą być instalowane na dowolnych serwerach aplikacyjnych Java takich jak Tomcat czy JBoss. WSO2 promuje użycie dedykowanego serwera WSO2 Application Server opartego o Tomcat i Axis2. Społeczność OpenESB rekomenduje stosowanie jedynie serwera GlassFish. ServiceMix domyślnie używa kontenera OSGI jako środowiska uruchomieniowego Java.

**Dostępność konektorów** – WSO2 oraz Apache Camel posiadają bardzo bogate[[14]](#footnote-14) zestawy adapterów umożliwiających komunikację w różnych standardach (JDBC, HTTP, SMTP, FTP, LDAP i inne) oraz konektorów do zewnętrznych platform i aplikacji (np. AWS, SalesForce, Google, SAP). Open ESB ogranicza się do podstawowych komponentów interfejsowych zgodnych ze specyfikacją JBI 1.0[[15]](#footnote-15).

**Środowisko programistyczne** – WSO2 udostępnia kompletne środowisko programistyczne WSO2 Developer Studio bazujące na Eclipse[[16]](#footnote-16). Open ESB dostarcza Open ESB Studio oparte o NetBeans. Apache ServiceMix nie oferuje dedykowanego środowiska deweloperskiego, wymagane jest użycie rozwiązań innych dostawców: Talend Open Studio for ESB, JBoss Developer Studio lub standardowej dystrybucji Eclipse.

**Możliwości administrowania i monitorowania** – WSO2 standardowo daje aplikacje administracyjne: Application Server Management Console do administracji serwerem aplikacyjnym oraz ESB Management Console do zarządzania aplikacją ESB. Apache ServiceMix współpracuje z rozbudowaną konsolą hawt.io[[17]](#footnote-17) dedykowaną dla Apache Camel i ActiveMQ. Open ESB jest dostarczany z konsolą Open ESB Administrative Web Console.

**Popularność wśród użytkowników** – zgodnie ze statystykami stackoverflow.com najpopularniejszym z porównywanych rozwiązań jest Apache Camel z ponad 5 tys. wpisów, kolejne jest WSO2 z ponad 2 tys. wątków. Open ESB dotyczą tylko 22 tematy. Analiza Google Trends[[18]](#footnote-18) również wskazuje na największe zainteresowanie zagadnieniami związanymi z Apache Camel.

**Częstotliwość aktualizacji i aktywność twórców** – na podstawie zestawienia OpenHUB Projects Compare[[19]](#footnote-19) można stwierdzić, że Apache Camel cieszy się największą aktywnością deweloperów współtworzących rozwiązanie oraz posiada najwyższe oceny użytkowników. Dla WSO2 i Open ESB brak statystyk aktywności z ostatnich 12 miesięcy. Apache Camel poświęcone jest forum użytkowników[[20]](#footnote-20) na którym obecni są twórcy rozwiązania.

**Jakość dokumentacji i dostępność źródeł wiedzy** – wszystkie rozwiązania posiadają dokumentację dostępną na swoich stronach domowych. Dokumentacja WSO2 i Apache Camel jest zdecydowanie bogatsza od Open ESB, zawiera przykłady i propozycje gotowych rozwiązań. Dystrybucja Apache Camel zawiera 64 przykłady gotowych projektów demonstracyjnych. Zarówno Apache Camel jak i WSO2 poświęcony jest szeroki wybór specjalistycznej literatury wydawanej przez Packt Publishing i Manning Publications.

Poniżej znajduje się tabelaryczne zestawienie porównywanych rozwiązań. Autor zastosował skalę punktową od 1 do 3 w zależności od spełnienia kryteriów. Zestawienie wyraźnie prezentuje najniższe wyniki Open ESB w większości kategorii. Wyniki WSO2 ESB i Apache ServiceMix różnią się praktycznie jedynie w zakresie popularności wśród użytkowników, aktywności twórców i dostępności źródeł wiedzy – na korzyść rozwiązania Apache ServiceMix.

|  |  |  |  |
| --- | --- | --- | --- |
| **Kryterium oceny** | **Open**  **ESB** | **WSO2**  **ESB** | **Apache ServiceMix** |
| Otwartość architektury | 1 | 2 | 2 |
| Wzorce integracyjne | 1 | 3 | 3 |
| Dostępność konektorów | 1 | 3 | 3 |
| Środowisko programistyczne | 3 | 3 | 2 |
| Administracja i monitorowanie | 2 | 3 | 3 |
| Popularność wśród użytkowników | 1 | 2 | 3 |
| Częstotliwość aktualizacji i aktywność autorów | 1 | 2 | 3 |
| Jakość dokumentacji i dostępność źródeł wiedzy | 1 | 2 | 3 |
| **SUMA** | **11** | **20** | **22** |

Tabela 1 Ocena porównawcza platform integracyjnych

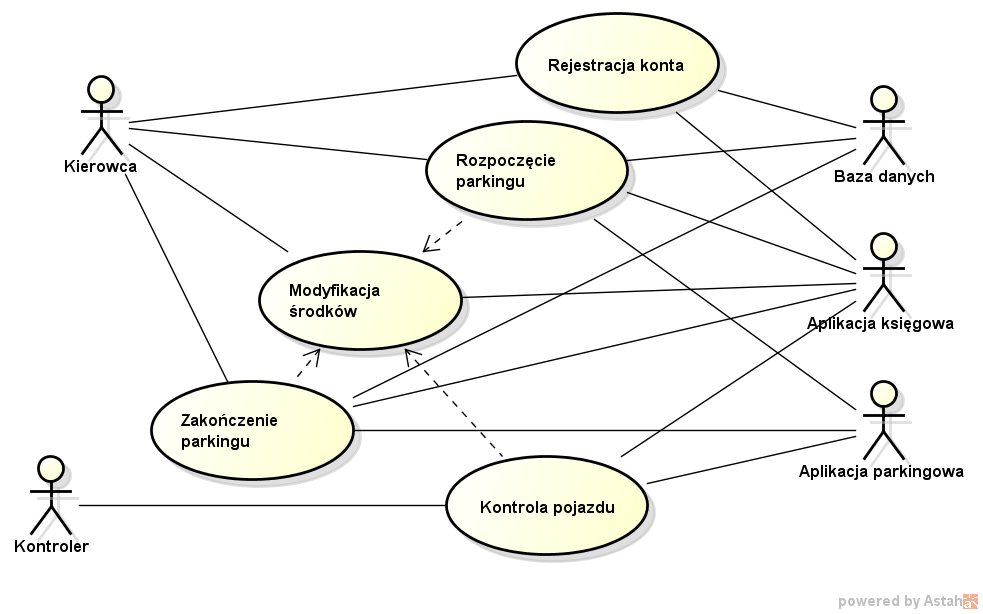
WSO2 jest rozwiązaniem stosunkowo zamkniętym, technologie na których bazuje zostały w dużym stopniu ukryte przed użytkownikiem i są dostępne jedynie za pomocą nakładek graficznych w dedykowanym środowisku programistycznym. Jest to korzystne z perspektywy programistycznej gdyż pewna część pracy została uprzednio wykonana przez autorów platformy i twórcy pozostaje jedynie konfiguracja i dostosowanie gotowych komponentów do własnych potrzeb. Z uwagi na fakt, że celem niniejszej pracy jest samodzielne zbudowanie aplikacji integracyjnej WSO2 zostało odrzucone przez autora jako rozwiązanie dające zbyt mało swobody w programowaniu, narzucające gotowe wzorce oraz niepozwalające na głębsze poznanie technologii o którą jest oparte.

Apache ServiceMix uzyskało najwyższą ocenę w przedstawionym porównaniu i jest to rozwiązanie wybrane przez autora do wykonania projektu. Ze względu na ściśle deweloperski charakter pracy autor zdecydował o pominięciu z wybranego rozwiązania komponentu Apache Karaf i zbudowaniu aplikacji w formie standardowego procesu Java. Decyzja podyktowana jest potrzebą skupienia się w niniejszej pracy na aspektach czysto programistycznych - stosowania wzorców integracyjnych i standardów tworzenia usług. Dodatkowe czynności związane z uruchomieniem kontenera OSGI lub osadzeniem aplikacji na serwerze aplikacyjnym nie wniosłyby wartości merytorycznej do projektu a jedynie utrudniły jego czytelność.

Podstawowy stos technologiczny Apache ServiceMix zostanie uzupełniony o dodatkowe komponenty: lekki serwer HTTP Jetty uruchamiany wewnątrz procesów Java; silnik usług sieciowych Restlet wspierający architekturę REST oraz bibliotekę Jackson obsługującą format wymiany danych JSON. Dostęp do bazy danych z poziomu obiektów Java zapewni Hibernate - popularny framework ORM (ang. *Object-relational mapping*) zgodny z JPA (ang. *Java Persistence API*). Jako środowisko programistyczne zostanie użyta standardowa dystrybucja Eclipse Mars wzbogacona o narzędzie automatyzujące budowę kodu Apache Maven.

# Specyfikacja rozwiązania

Poniższy diagram przypadków użycia prezentuje wysokopoziomowy zakres funkcjonalny projektowanej aplikacji. Zgodnie z tematem projektu zostali wyodrębnieni aktorzy: Kierowca jest przypisany do przypadków użycia rejestracji konta, rozpoczęcia i zakończenia parkowania oraz modyfikacji posiadanych środków; Kontroler realizuje przypadek kontroli pojazdu. Systemy uczestniczące w realizacji przypadków użycia to: Baza Danych gromadząca informacje o kontach i parkowaniu, Aplikacja księgowa nadzorująca rozliczanie opłat za parkowanie i mandaty, Aplikacja parkingowa nadzorująca czynności parkowania i kontroli.



Rysunek 1 Diagram przypadków użycia

Przypadek użycia *Rejestracja konta* jest inicjowany przez otrzymanie zlecenia rejestracji konta od konsumenta *Aplikacja kierowcy*. Zadaniem warstwy integracyjnej jest przyjęcie i walidacja zlecenia; sprawdzenie unikalności zlecenia z kontami aktualnie zarejestrowanymi w bazie danych; zapisanie danych osobowych i adresowych użytkownika oraz danych pojazdu w odpowiednich strukturach *Bazy danych*; zarejestrowanie rachunku użytkownika w formie zasobu plikowego na potrzeby *Aplikacji księgowej*; synchroniczne zwrócenie odpowiedzi do konsumenta. W przypadku powodzenia wykonania całego scenariusza odpowiedź zawiera dane identyfikacyjne utworzonego konta, w przypadku wyjątku w trakcie wykonywania scenariusza – odpowiedni kod błędu uzgodniony z konsumentem.

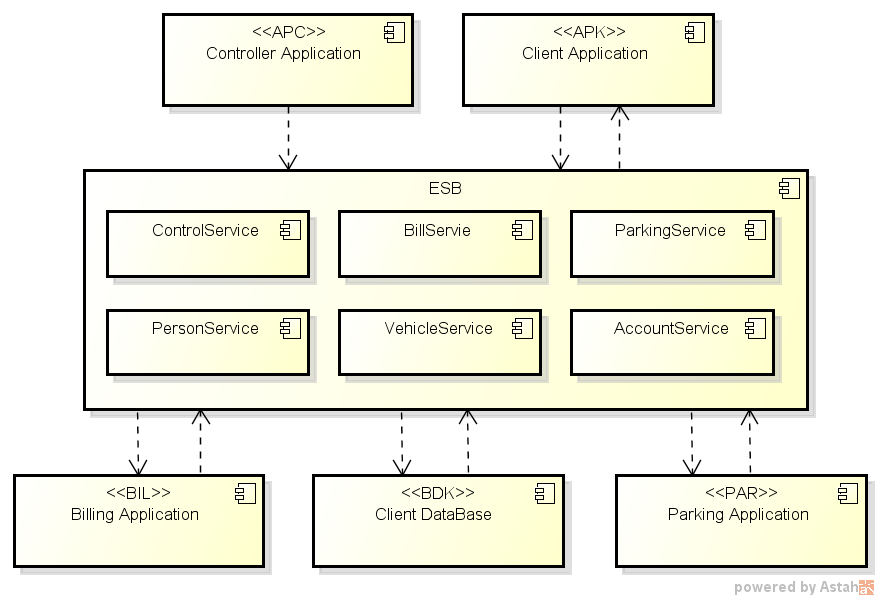
Przypadek użycia *Modyfikacja środków* jest inicjowany przez otrzymanie zlecenia modyfikacji środków od konsumenta *Aplikacja kierowcy*. Zadaniem warstwy integracyjnej jest przyjęcie i walidacja poprawności zlecenia; pobranie konta użytkownika z *Bazy danych* na podstawie przekazanych w zleceniu danych identyfikacyjnych rachunku; pobranie rachunku użytkownika z zasobu plikowego; kalkulacja salda rachunku na podstawie aktualnego stanu środków z rachunku i przekazanej w zleceniu kwoty zasilenia; aktualizacja rachunku użytkownika na zasobie plikowym zgodnie z aktualnym saldem; synchroniczne zwrócenie odpowiedzi do konsumenta. Przekazana w zleceniu kwota zasilenia jest dodatnia, w związku z czym scenariusz przewiduje tylko możliwość powiększenia salda.

Przypadek użycia *Rozpoczęcie parkingu* jest inicjowany przez otrzymanie zlecenia zakupu biletu parkingowego od konsumenta *Aplikacja kierowcy*. Zadaniem warstwy integracyjnej jest przyjęcie i walidacja poprawności zlecenia; pobranie konta użytkownika z *Bazy danych*; sprawdzenie unikalności zlecenia z aktualnie zarejestrowanymi biletami; kalkulacja opłaty za parking; modyfikacja salda rachunku zgodnie z przypadkiem *Modyfikacja środków* poprzez przekazanie zlecenia modyfikacji z ujemną kwotą zasilenia; zarejestrowanie biletu w *Bazie danych*; przekazanie informacji o rozpoczętym parkingu do dawcy *Aplikacja parkingowa*; wysłanie notyfikacji do dawcy *Aplikacja kierowcy* informującej o statusie realizacji zlecenia.

Przypadek użycia *Zakończenie parkingu* jest inicjowany przez otrzymanie zlecenia zakończenia parkingu od konsumenta *Aplikacja kierowcy*. Zadaniem warstwy integracyjnej jest przyjęcie i walidacja poprawności zlecenia; pobranie konta użytkownika i biletu parkingowego z *Bazy danych*; sprawdzenie aktualności biletu; kalkulacja opłaty za parking w związku ze skróceniem czasu parkowania; modyfikacja salda rachunku zgodnie z przypadkiem *Modyfikacja środków* poprzez przekazanie zlecenia modyfikacji z dodatnią kwotą zasilenia; przekazanie informacji o zakończonym parkingu do dawcy *Aplikacja parkingowa*; aktualizacja biletu w *Bazie danych*; wysłanie notyfikacji do dawcy *Aplikacja kierowcy* informującej o statusie realizacji zlecenia.

Przypadek użycia *Kontrola pojazdu* jest inicjowany przez otrzymanie zlecenia kontroli od konsumenta *Aplikacja kontrolera*. Zadaniem warstwy integracyjnej jest przyjęcie i walidacja zlecenia; przekazanie informacji o kontroli do dawcy *Aplikacja parkingowa*; wyszukanie konta kontrolowanego użytkownika w *Bazie danych* na podstawie przekazanych w zleceniu informacji o kontrolowanym pojeździe; pobranie rachunku z zasobu plikowego; weryfikacja posiadania przez użytkownika aktualnego biletu parkingowego dla kontrolowanego pojazdu; w przypadku pozytywnej weryfikacji biletu parkingowego - zwrócenie odpowiedzi do konsumenta o poprawnie przeprowadzonej kontroli; w przypadku negatywnej weryfikacji biletu - kalkulacja opłaty za mandat; modyfikacja salda rachunku zgodnie z przypadkiem *Modyfikacja środków* z ujemną kwotą zasilenia; przekazanie informacji o wystawionym mandacie do dawcy *Aplikacja parkingowa*; synchroniczne zwrócenie odpowiedzi do konsumenta.

Podczas typowania kandydatów na usługi sieciowe, zgodnie z dobrymi praktykami SOA, istotny jest dobór odpowiedniej „granularności”, czyli zakresu funkcjonalnego obejmowanego przez poszczególne usługi. W przypadku zbyt wysokiej granularności efektem jest duża liczba usług realizujących wąskie fragmenty funkcjonalności, co utrudnia ich utrzymywanie i wykrywanie; w przypadku zbyt niskiej granularności otrzymuje się silosy funkcjonalne utrudniające reużywalność. Realizacja powyższych przypadków użycia w warstwie integracyjnej została powierzona następującym usługom sieciowym: usługa AccountService odpowiadająca za funkcjonalności związane z rejestrowaniem, wyszukiwaniem i pobieraniem kont użytkowników; usługa BillService obsługująca tworzenie rachunków i modyfikacje salda; usługa ControlService obejmująca przypadek użycia związany z kontrolą pojazdu; usługa ParkingService realizująca funkcjonalności związane z parkowaniem – rozpoczynanie i kończenie parkowania oraz naliczanie opłat; usługi PersonService i VehicleService zapewniające obsługę kierowców i pojazdów. Zdaniem autora przedstawiony zestaw usług spełnia wymóg odpowiedniej granularności – usługi operujące na konkretnych istotnych obiektach takich jak konto czy pojazd są wysoce reużywalne ze względu na swój uniwersalny charakter wzorca CRUD (ang. *Create Read Delete Update*); usługi dostarczające wysokopoziomowe funkcjonalności takie jak rejestracja konta czy kontrola pojazdu są optymalne z perspektywy konsumentów z uwagi na ich dobrą wykrywalność i jednoznaczne kontrakty. Poniższy diagram komponentów obrazuje na wysokim poziomie architekturę integracyjną systemu z uwzględnieniem wytypowanych usług, szczegóły relacji między aplikacjami i usługami zostaną przedstawione w dalszej części pracy.



Rysunek 2 Wysokopoziomowa architektura logiczna integracji

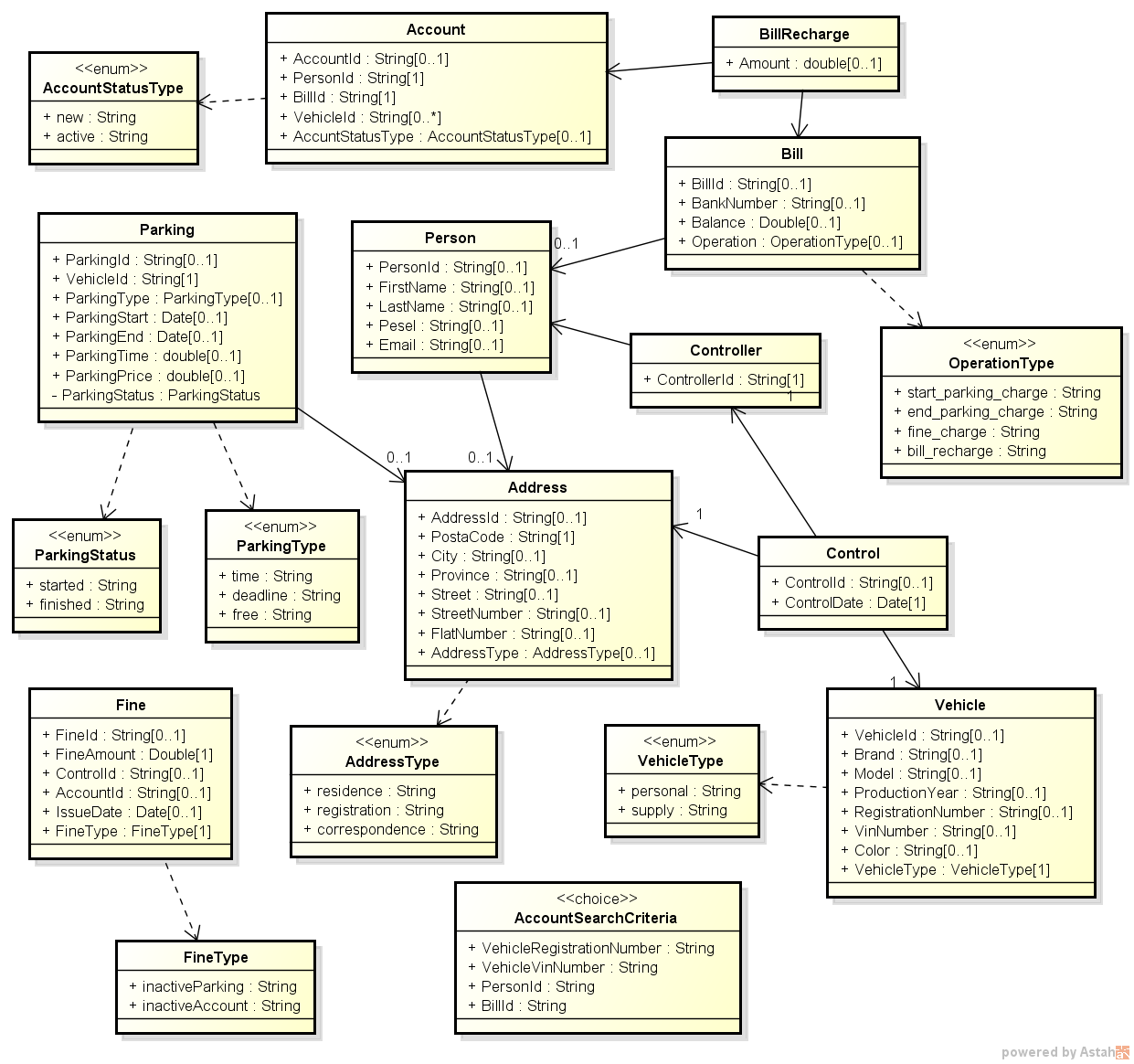
Dodatkowo system powinien spełniać wymagania niefunkcjonalne i techniczne:

1. Komunikacja z konsumentami za pomocą standardu REST przez protokół transportowy HTTP z użyciem formatu danych JSON; konieczność dopasowania interfejsów wejściowych usług ControlService, BillService, ParkingService, AccountService udostępnianych dla Aplikacji kierowcy i Aplikacji kontrolera do wymogów technicznych tych konsumentów.
2. Komunikacja z Aplikacją księgową poprzez pliki CSV na współdzielonym zasobie plikowym; wymagane dostosowanie adapterów plikowych usług do istniejącego formatu danych i zasad operowania na zasobie współdzielonym.
3. Komunikacja z Bazą danych MySQL za pomocą sterownika JDBC w określonej wersji; wymagana konfiguracja konektora uwzględniająca adres dostępowy do bazy danych oraz uwierzytelnianie użytkownika w odpowiednim schemacie bazodanowym.
4. Komunikacja z Aplikacją parkingową przez protokół SOAP zgodnie z udostępnionymi standardowymi kontraktami w formie plików WSDL (ang. *Web Services Description Language*).
5. Wsparcie dla komunikacji asynchronicznej z użyciem kolejek persystentnych.
6. Wysyłka wiadomości email do użytkowników systemu.

## Projekt techniczny integracji

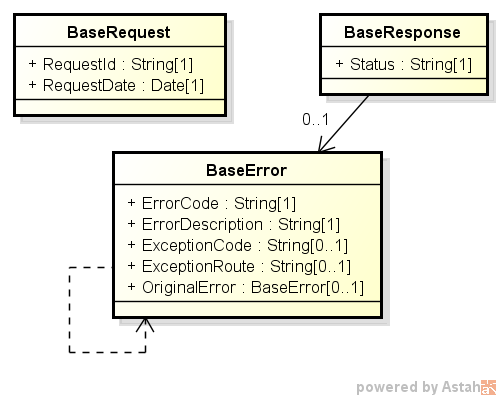
Projekt techniczny zawiera specyfikację Kanonicznego modelu danych zaprojektowanego na potrzeby realizowanych scenariuszy z udziałem integrowanych systemów i wytypowanych usług. Zgodnie z wzorcami SOA model opisuje wspólne, generyczne encje, którymi posługują się usługi podczas przetwarzania zleceń. Model został wytworzony na podstawie analizy interfejsów systemów konsumentów i dawców, jednak nie odzwierciedla ich dokładnie a raczej uogólnia w celu zapewnienia warstwy abstrakcji danych.

### Kanoniczny model danych



Rysunek 3 Kanoniczny model danych

Model kanoniczny został uzupełniony o prosty model techniczny, definiujący: standardowe wejście (*BaseRequest*) zawierające metadane takie jak identyfikator i data wywołania usługi oraz standardowe wyjście (*BaseResponse*) zawierające status wykonania usługi i opcjonalną kolekcję błędów.



Rysunek 4 Techniczny model danych

### Opis usług

Usługi zostały zaprojektowane w architekturze trójwarstwowej. Oznacza to, że każda usługa jest faktycznie realizowana przez trzy usługi w oddzielnych warstwach, odpowiedzialne za wykonywanie określonych zadań.

Warstwa *adapter* jest dedykowana dla konkretnego systemu konsumenta i odpowiada za techniczne aspekty dopasowania transportu i protokołu komunikacyjnego do interfejsu tego systemu. W zależności od wymagań usługa w tej warstwie jest pasywna (np. wystawia webservice przyjmujący zewnętrzne wywołania) lub aktywna czyli realizuje *polling* (aktywnie bada określony zasób pod kątem zmian np. aktualizacja rekordu bazodanowego, komunikat na kolejce, plik na zasobie). W tej warstwie odbywa się translacja z modelu danych konsumenta do modelu kanonicznego. Usługi warstwy *adapter* komunikują się tylko z usługami warstwy *base*.

Warstwa *base* operuje w modelu kanonicznym i realizuje logikę integracyjną. Tutaj odbywają się złożone walidacje i transformacje między encjami modelu kanonicznego, wzbogacanie danych poprzez orkiestracje[[21]](#footnote-21) wywołań innych usług, kompensowanie akcji w przypadku wystąpienia wyjątków itd. Usługi warstwy *base* nie są bezpośrednio wystawiane konsumentom a jedynie udostępniane za pomocą usług warstwy *adapter*. Ma to na celu odseparowanie implementacji usługi od jej interfejsu. Usługi warstwy *base* komunikują się, w zależności od potrzeb, z innymi usługami *base* lub *konektor*.

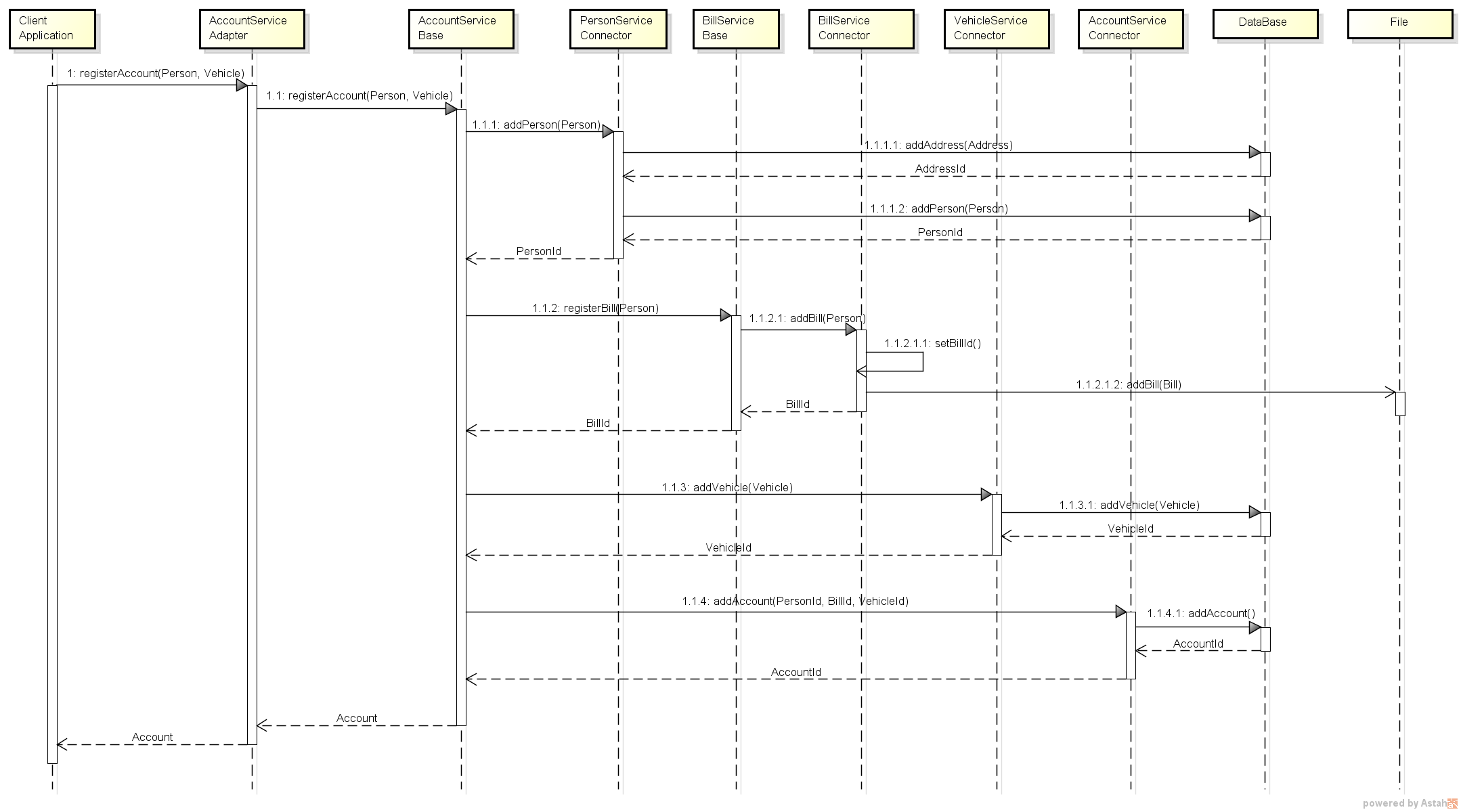
Warstwa *konektor* jest odpowiednikiem warstwy *adapter* ale dla systemów dawców. Usługi tej warstwy odpowiadają za translację danych z modelu kanonicznego do modelu systemu dawcy oraz techniczne dopasowanie do interfejsu dawcy (np. konektor plikowy lub bazodanowy odpowiadający za wirtualizację warstwy danych). W uzasadnionych przypadkach usługi warstwy *konektor* mogą reużywać innych usług z tej warstwy.

W poniższych opisach usług zastosowano następujące nazewnictwo:

1. *AccountService* – ogólna nazwa usługi dotycząca wszystkich warstw i operacji.
2. *AccountServiceAdapter* – nazwa usługi w danej warstwie.
3. *registerAccount* – ogólna nazwa operacji dotycząca wszystkich warstw.
4. *registerAccountAdapter* – nazwa operacji w danej warstwie.
5. *AccountServiceAdapter.registerAccountAdapter* – pełna formalna nazwa operacji.

#### Usługa AccountService

Usługa realizuje czynności związane z kontami użytkowników. W warstwie adapter usługa udostępnia tylko operację rejestracji nowego konta *registerAccountAdapter* dla konsumenta Aplikacja kierowcy (kontrakt usługi został przedstawiony w dalszej części rozdziału). Operacja ta przyjmuje wywołania od konsumenta, transformuje komunikat do modelu kanonicznego i przekazuje do operacji *registerAccountBase*. W warstwie base wykonywana jest orkiestracja usług zgodnie z poniższym diagramem sekwencji:

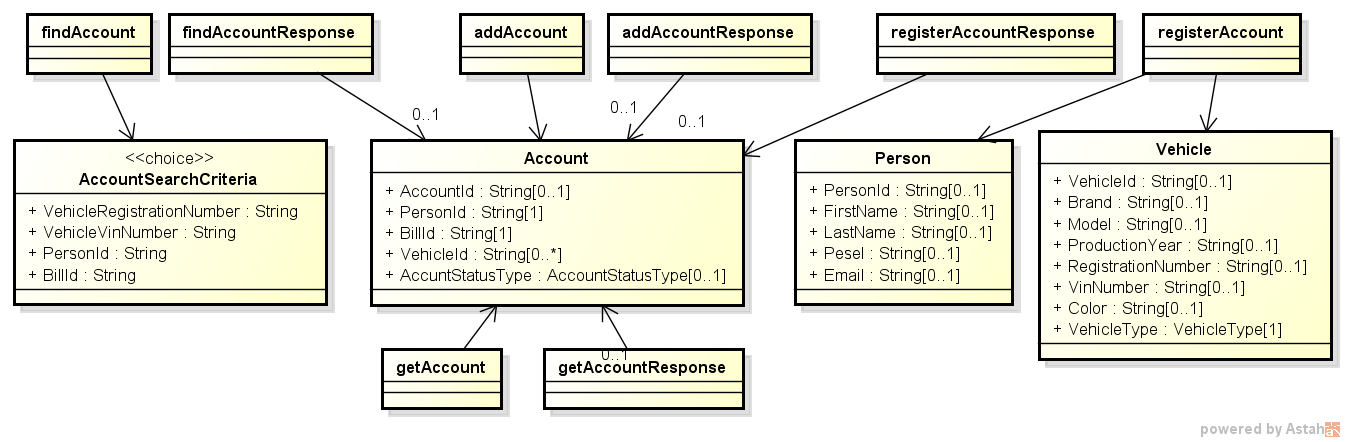


Rysunek 5 Diagram sekwencji operacji AccountService.registerAccount

W warstwie konektor usługa zawiera operacje będące konektorami bazodanowymi:

1. *addAccountConnector* - zapisuje nowy rekord konta w bazie danych.
2. *getAccountConnector* - pobiera rekord konta o podanym identyfikatorze.
3. *findAccountConnector* - wyszukuje rekord konta na podstawie podanych kryteriów.

Model danych usługi korzysta z encji kanonicznego modelu danych: Account, Person, Vehicle, AccountSearchCriteria:



Rysunek 6 Model danych usługi AccountService

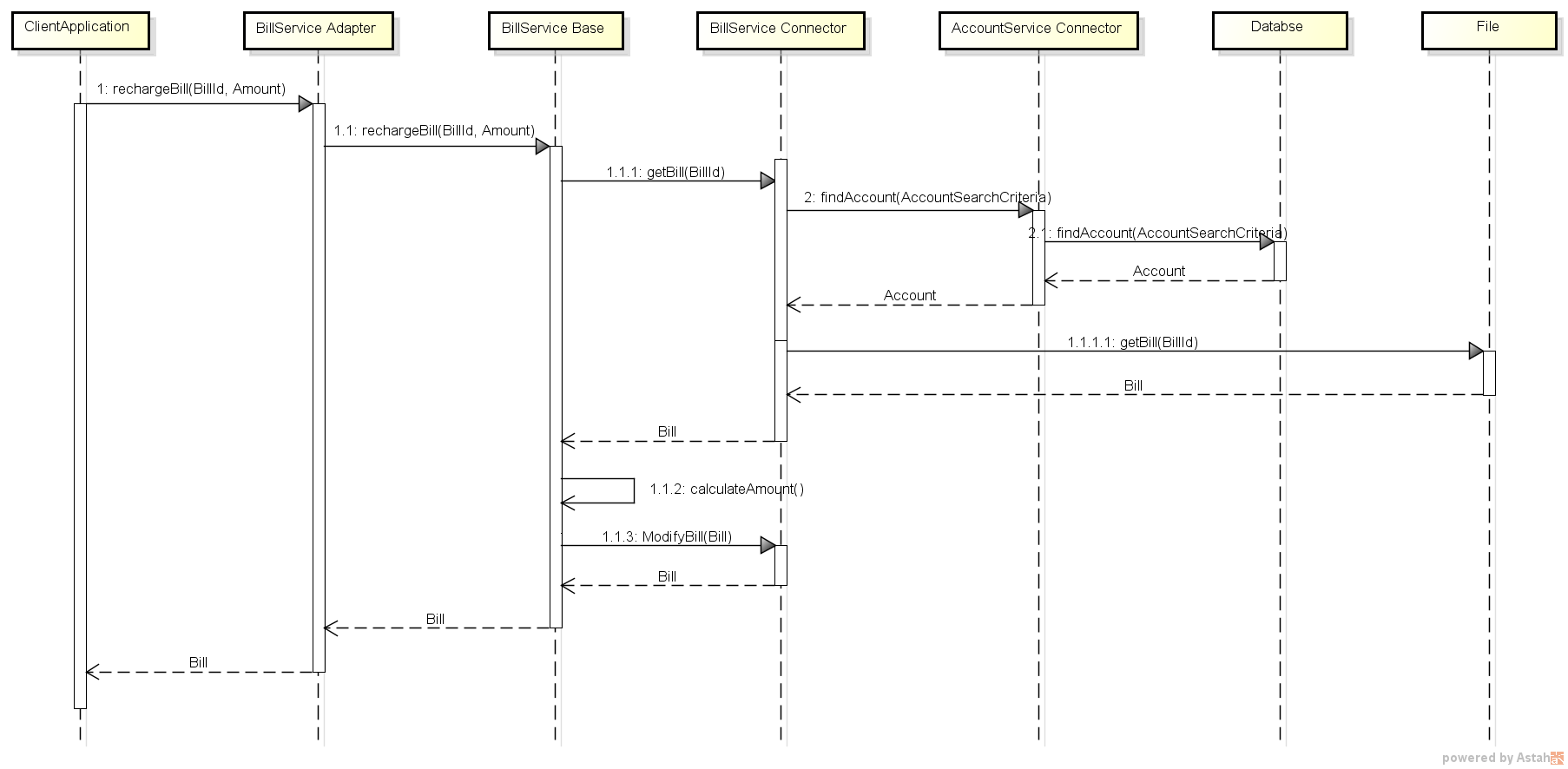
Kontrakt usługi AccountService w zakresie operacji registerAccount dla konsumenta Aplikacja kierowcy:

|  |  |
| --- | --- |
| **AccountServiceAdapter.registerAccountAdapter REST API** | |
| Metoda HTTP | POST |
| Adres HTTP | <http://eparking.client.app:8083/api/account/register> |
| Nagłówki HTTP | Content-Type: application/json |
| Request Body | {  "person":{  "first\_name":"<IMIE:tekst>",  "last\_name":"<NAAZWISKO:tekst>",  "pesel":"<PESEL:liczba:NNNNNNNNNNN>",  "email":"<EMAIL:tekst>",  "address":{  "address\_type":"<TYP\_ADRESU:tekst:residence|registration|correspondence>",  "province":"<WOJEWODZTWO:tekst>",  "postal\_code":"<KOD\_POCZTOWY:liczba:NN-NNN>",  "city":"<MIASTO:tekst>",  "street":"<ULICA:tekst>",  "street\_number":"<NUMER\_DOMU:liczba>",  "flat\_number":"<NUMER\_MIESZKANIA:liczba>"  }  },  "vehicle":{  "vehicle\_type":"<TYP\_POJAZDU:personal|supply>",  "brand":"<MARKA:tekst>",  "model":"<MODEL:tekst>",  "registration\_number":"<NUMER\_REJESTRACYJNY:tekst>",  "vin\_number":"<NUMER\_VIN:teskt>",  "production\_date":"<DATA\_PRODUKCJI:data:YYYY-MM-DD>",  "color":"<KOLOR:tekst>"  }  } |
| Response – Sukces | HTTP Response code: 200  Content-type: application/json  {  "account\_id":"<IDENTYFIKATOR\_KONTA:tekst>",  "person\_id":"<IDENTYFIKATOR\_KIEROWCY:teskt>",  "billing\_id":"<IDENTYFIKATOR\_RACHUNKU:tekst>",  "vehicle\_id":"<IDENTYFIKATOR\_POJAZDU:tekst>",  "status":"<STATUS\_KONTA:new>"  } |
| Response – Błąd walidacji wejścia | HTTP Response code: 400  Content-type: text/plain  Content-length: 0  Exception: Validation failed for predicate [treść wyjątku walidatora] |
| Response – Błąd unikalności danych | HTTP Response code: 501  Content-type: text/plain  Content-length: 0  Exception: AccountServiceAdapter registerAccount duplicate account error |
| Response – Błąd  bazy danych | HTTP Response code: 500  Content-type: text/plain  Content-length: 0  Exception: AccountServiceAdapter registerAccount general error |

Tabela 2 Kontrakt usługi AccountService

#### Usługa BillService

Usługa realizuje czynności związane z rachunkami i saldem. W warstwie adapter usługa udostępnia tylko operację zasilenia rachunku *rechargeBillAdapter* dla konsumenta Aplikacja kierowcy (kontrakt usługi został przedstawiony w dalszej części rozdziału). Operacja przyjmuje wywołania od konsumenta i przekazuje do operacji *rechargeBillBase*. W warstwie base wykonywana jest orkiestracja usług zgodnie z poniższym diagramem sekwencji:

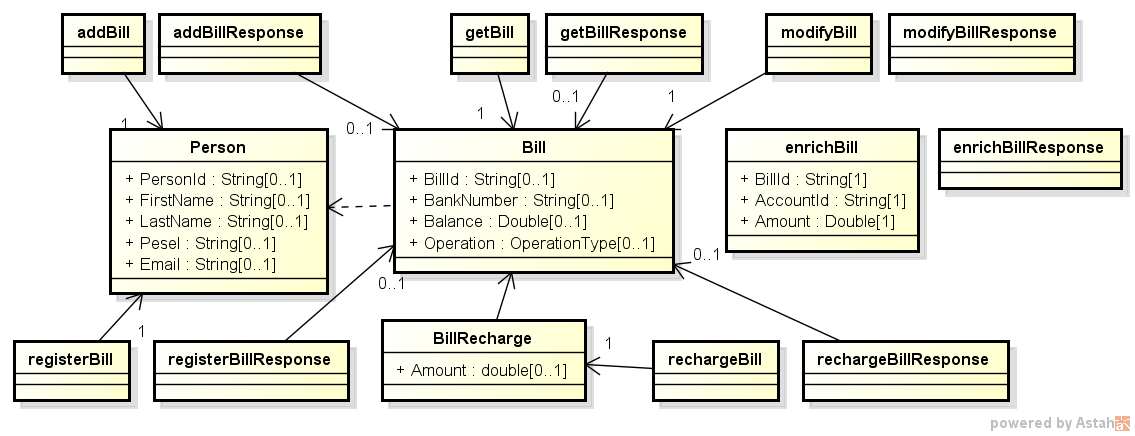


Rysunek 7 Diagram sekwencji operacji BillService.rechargeBill

W warstwie base poza operacją zasilenia rachunku jest osadzona operacja rejestracji rachunku *registerBillBase* reużywana przez usługę AccountService w ramach operacji registerAccount. Operacja *registerBillBase* nie posiada złożonej logiki i realizuje jedynie wywołanie usługi konektora plikowego *addBillConnector*. W warstwie konektor usługa zawiera łącznie trzy operacje będące konektorami plikowymi:

1. *addBillConnector* - tworzy nowy rachunek na zasobie plikowym dla Aplikacji księgowej.
2. *getBillConnector* - pobiera istniejący rachunek z zasobu plikowego.
3. *modifyBillConnector* - modyfikuje istniejący rachunek na zasobie plikowym.

Model danych usługi korzysta z encji kanonicznego modelu danych: Bill, Person:



Rysunek 8 Model danych usługi BillService

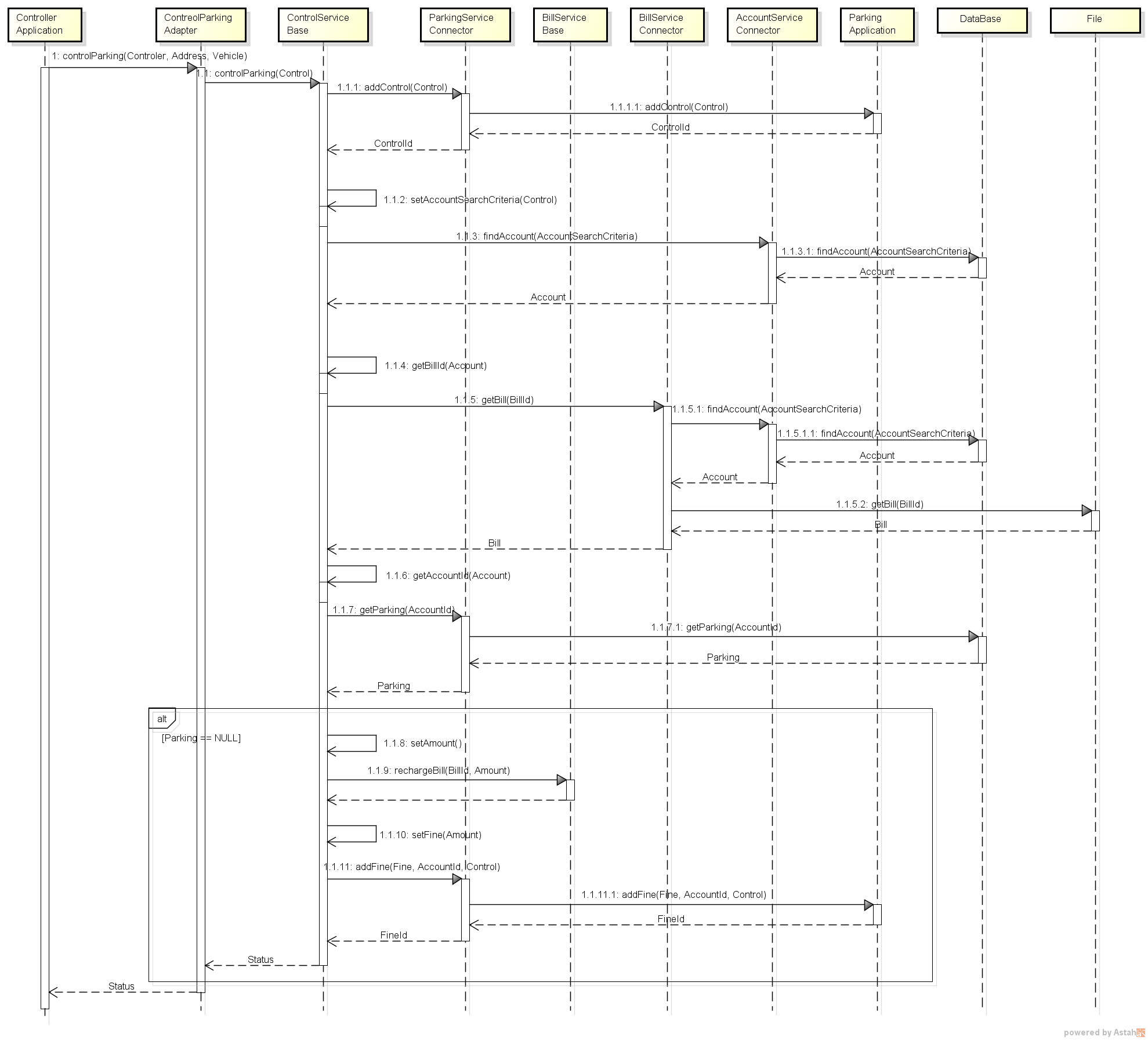
Kontrakt usługi BillService w zakresie operacji rechargeBill dla konsumenta Aplikacja kierowcy:

|  |  |
| --- | --- |
| **BillServiceAdapter.rechargeBillAdapter REST API** | |
| Metoda HTTP | POST |
| Adres HTTP | [http://eparking.client.app:8082/api/bill/<BILL\_ID>/recharge/<KWOTA](http://eparking.client.app:8082/api/bill/%3cBILL_ID%3e/recharge/%3cKWOTA)>  <BILL\_ID> - 32 znakowy identyfikator rachunku  <KWOTA> - wartość kwoty zasilenia rachunku w formacie dziesiętnym |
| Nagłówki HTTP | Content-Type: application/json  Content-Length: 0 |
| Request Body | - |
| Response – Sukces | - |
| Response – Błąd walidacji wejścia | HTTP Response code: 400  Content-type: text/plain  Content-length: 0  Exception: Validation failed |
| Response – Brak rachunku | HTTP Response code: 500  Content-type: text/plain  Content-length: 0  Exception: Error processing getBill |

Tabela 3 Kontrakt usługi BillService

#### Usługa ControlService

Usługa realizuje czynności związane z kontrolą parkingu. W warstwie adaptera usługa udostępnia tylko operację rejestracji kontroli *controlParkingAdapter* dla konsumenta Aplikacja kontrolera (kontrakt usługi został przedstawiony w dalszej części rozdziału). Operacja przekazuje wywołania do operacji *controlParkingBase*. W warstwie base wykonywana jest orkiestracja usług zgodnie z poniższym diagramem sekwencji:



Rysunek 9 Diagram sekwencji operacji ControlService.controlParking

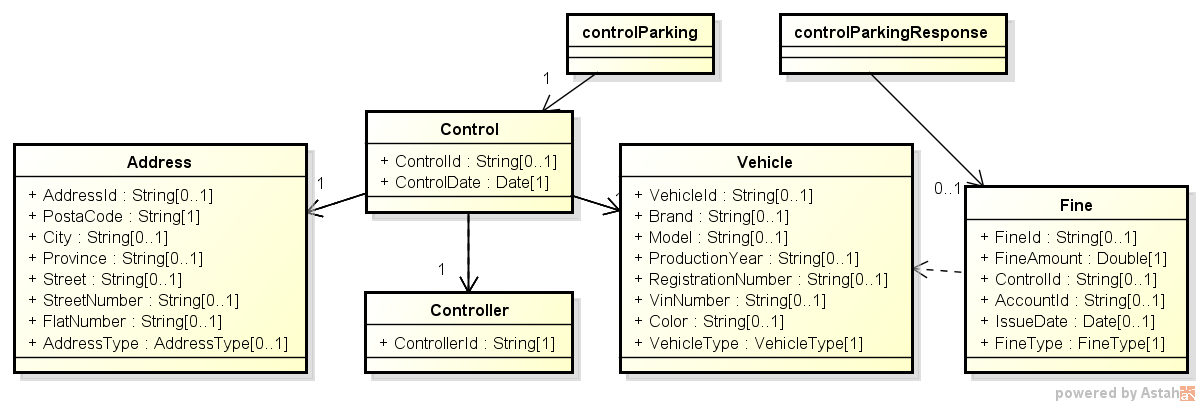
Usługa nie posiada operacji w warstwie konektora. Cała logika zawarta w warstwie base, opisana na powyższym diagramie, jest realizowana poprzez reużycie operacji z innych usług:

1. *ParkingServiceConnector.addControl* – powiadamia Aplikację parkingową o nowej kontroli.
2. *AccountServiceConnector.findAccount* – wyszukuje konta użytkownika w bazie danych.
3. *BillServiceConnector.getBill* – pobiera rachunek użytkownika z zasobu plikowego.
4. *ParkingServiceConnector.getParking* – pobiera bilet parkingowy użytkownika z bazy danych.

W przypadku braku biletu parkingowego usługa decyduje o wystawieniu mandatu, nalicza kwotę mandatu i wywołuje kolejne operacje:

1. *BillServiceBase.rechargeBill* – aktualizacja salda rachunku o kwotę mandatu.
2. *ParkingServiceConnector.addFine* - powiadomienie Aplikacji parkingowej o mandacie.
3. *ControlServiceBase.eparkingEmailNotifier* – operacja pomocnicza będąca konektorem SMTP do wysyłki wiadomości email na adres użytkownika obciążonego mandatem.

Model danych usługi korzysta z encji kanonicznego modelu danych: Address, Vehicle, Fine, Control, Controller:



Rysunek 10 Model danych usługi ControlService

Kontrakt usługi ControlService w zakresie operacji controlParking dla konsumenta Aplikacja kontrolera:

|  |  |
| --- | --- |
| **ControlServiceAdapter.controlParkingAdapter REST API** | |
| Metoda HTTP | POST |
| Adres HTTP | <http://eparking.client.app:8085/api/control> |
| Nagłówki HTTP | Controller-id: <IDENTYFIKATOR\_KONTROLERA:tekst>  Controller-name: <IMIE\_NAZWISKO\_KONTROLERA:tekst>  Controller-mail: <ADRES\_EMAIL\_KONTROLERA:text>  Control-code: <MIEJSCE\_KONTROLI\_KOD\_POCZTOWY:liczba:NN-NN>  Control-city: <MIEJSCE\_KONTROLI\_MIASTO:tekst>  Control-province: <MIEJSCE\_KONTROLI\_WOJEWODZTWO:tekst>  Control-street: <MIEJSCE\_KONTROLI\_ULICA:tekst>  Control-streetnumber: <MIEJSCE\_KONTROLI\_ULICA\_NUMER:tekst>  Control-vehicleregistration: <POJAZD\_KONTROLOWANY\_NUMER\_REJESTRCYJNY:tekst>  Control-vehicletype: <POJAZD\_KONTROLOWANY\_TYP:tekst:personal|supply>  Content-Length: 0 |
| Request Body | - |
| Response – Sukces | HTTP Response code: 200  Content-length: 0 |
| Response – Błąd walidacji wejścia | HTTP Response code: 400  Exception: Invalid headers: [treść wyjątku walidatora]  Content-length: 0 |
| Response – Błąd Aplikacji parkingowej | HTTP Response code: 500  Exception: Error processing addControl  Content-length: 0 |
| Response – Błąd Aplikacji parkingowej | HTTP Response code: 500  Exception: Error processing addFine  Content-length: 0 |

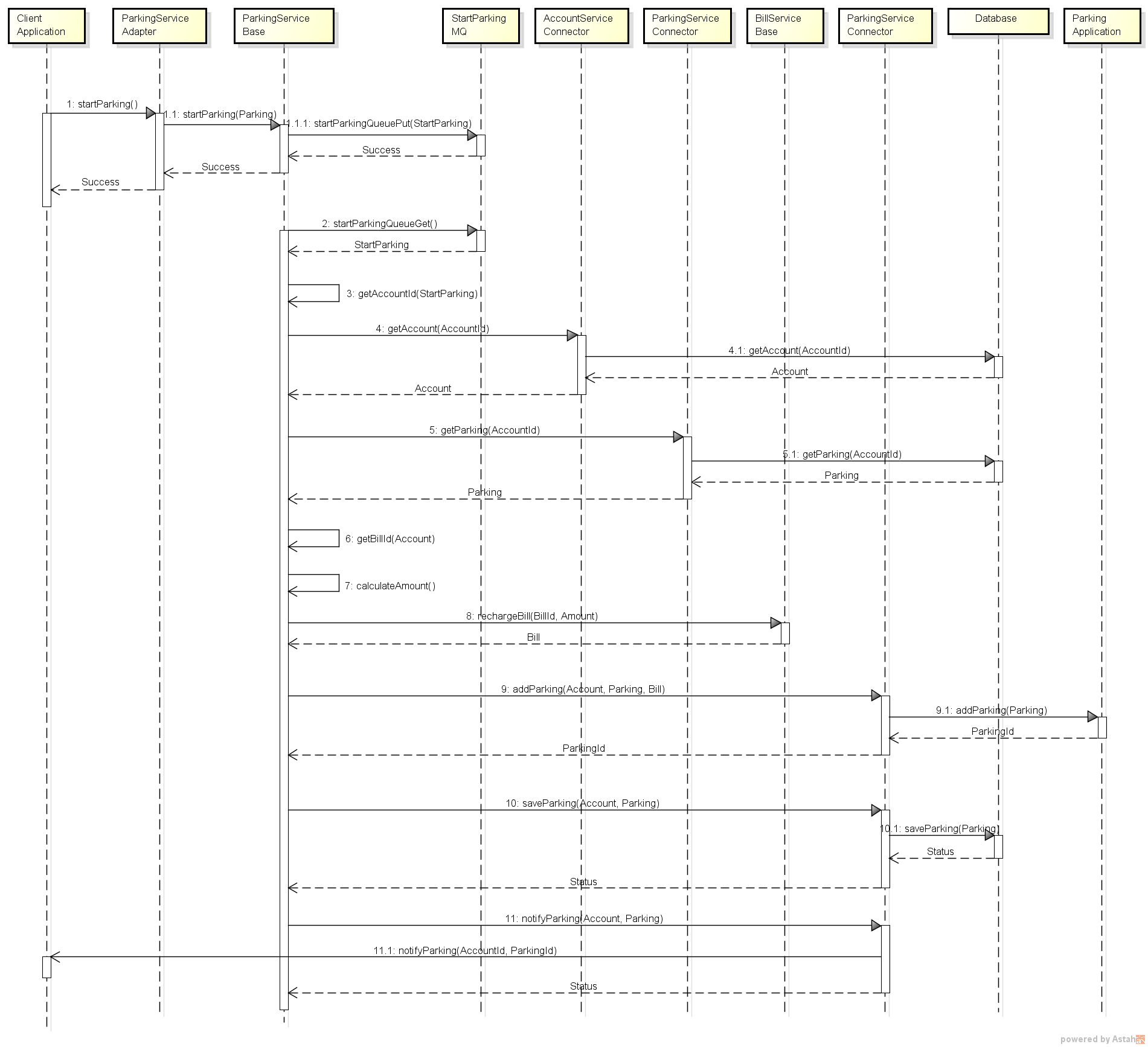
Tabela 4 Kontrakt usługi BillService

#### Usługa ParkingService

Usługa realizuje czynności związane z parkowaniem – umożliwia rozpoczęcie i zakończenie parkowania oraz wspiera usługę obsługującą kontrole w zakresie komunikacji z Aplikacją parkingową. W warstwie adapter usługa udostępnia dwie operacje (kontrakt usługi został przedstawiony w dalszej części rozdziału) dla konsumenta Aplikacja kierowcy:

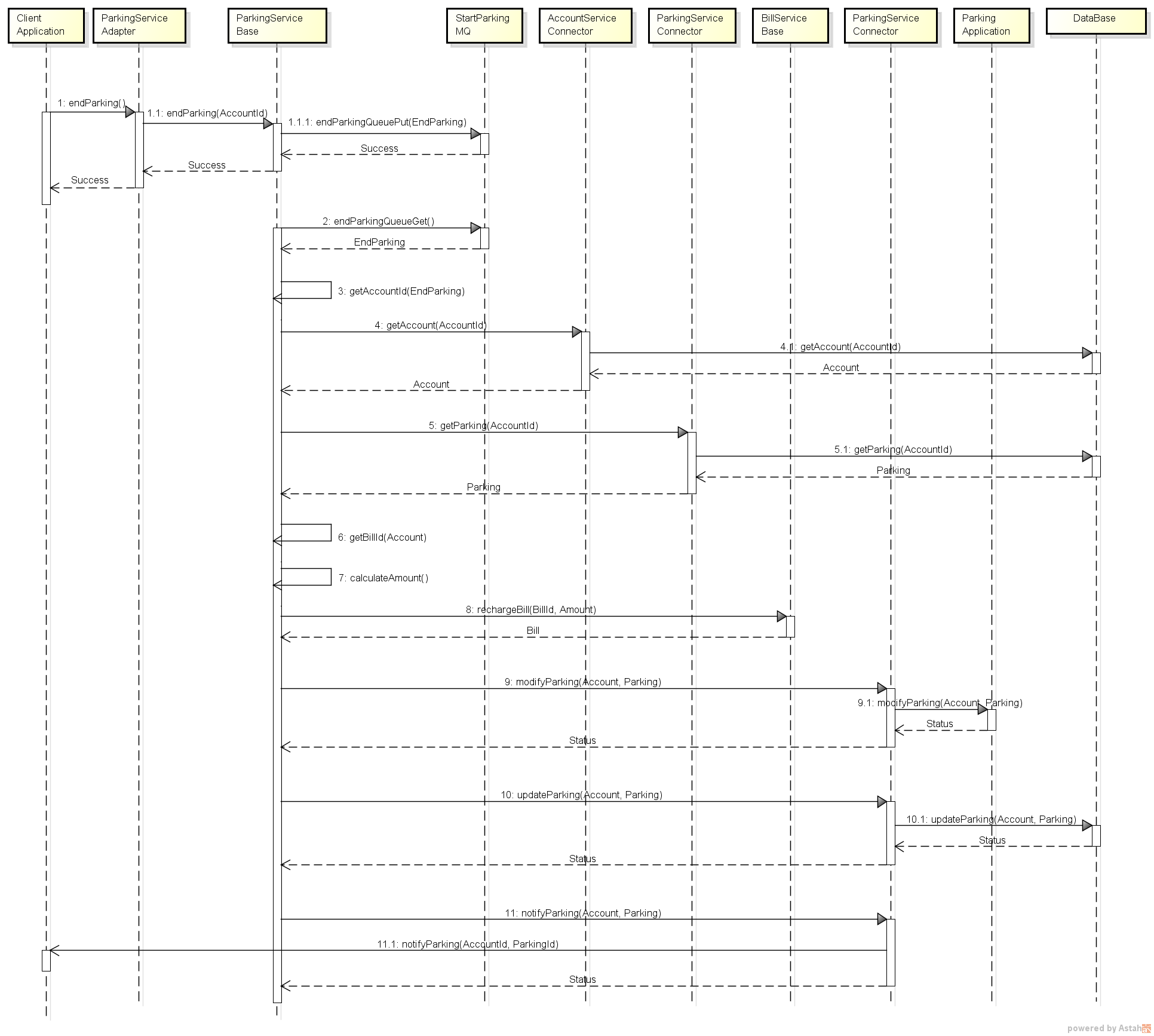
1. *startParking* – zakup biletu parkingowego i uruchomienie parkingu w jednym z 3 trybów.
2. *endParking* – zakończenie parkowania i zwrot różnicy w opłacie za parking.

Operacja *startParkingAdapter* przyjmuje wywołania od konsumenta i przekazuje je do warstwy base. W warstwie base komunikat zlecenia rozpoczęcia parkingu jest zapisywany na kolejkę w celu obsłużenia go w trybie asynchronicznym. Operacja *startParkingBaseQueuePut* zapisuje komunikat na kolejce i zwraca odpowiedź do adaptera, który przekazuje informację zwrotną do konsumenta (pierwsza część usługi realizowana synchronicznie). Operacja *startParkingBaseQueueGet* ściąga komunikat z kolejkii wykonuje dalszą orkiestrację usług zgodnie z poniższym diagramem sekwencji. Po zakończeniu procesowania do konsumenta wysyłana jest notyfikacja w celu poinformowania o statusie realizacji zlecenia (druga część również realizowana synchronicznie).



Rysunek 11 Diagram sekwencji operacji ParkingService.startParking

Operacja *endParkingAdapter* przyjmuje wywołania i przekazuje je do warstwy base. W warstwie base komunikat jest obsługiwany w trybie asynchronicznym analogicznie jak dla operacji powyżej (z użyciem osobnych operacji i osobnej dedykowanej kolejki). Operacja wykonuje orkiestrację usług zgodnie z poniższym diagramem sekwencji:



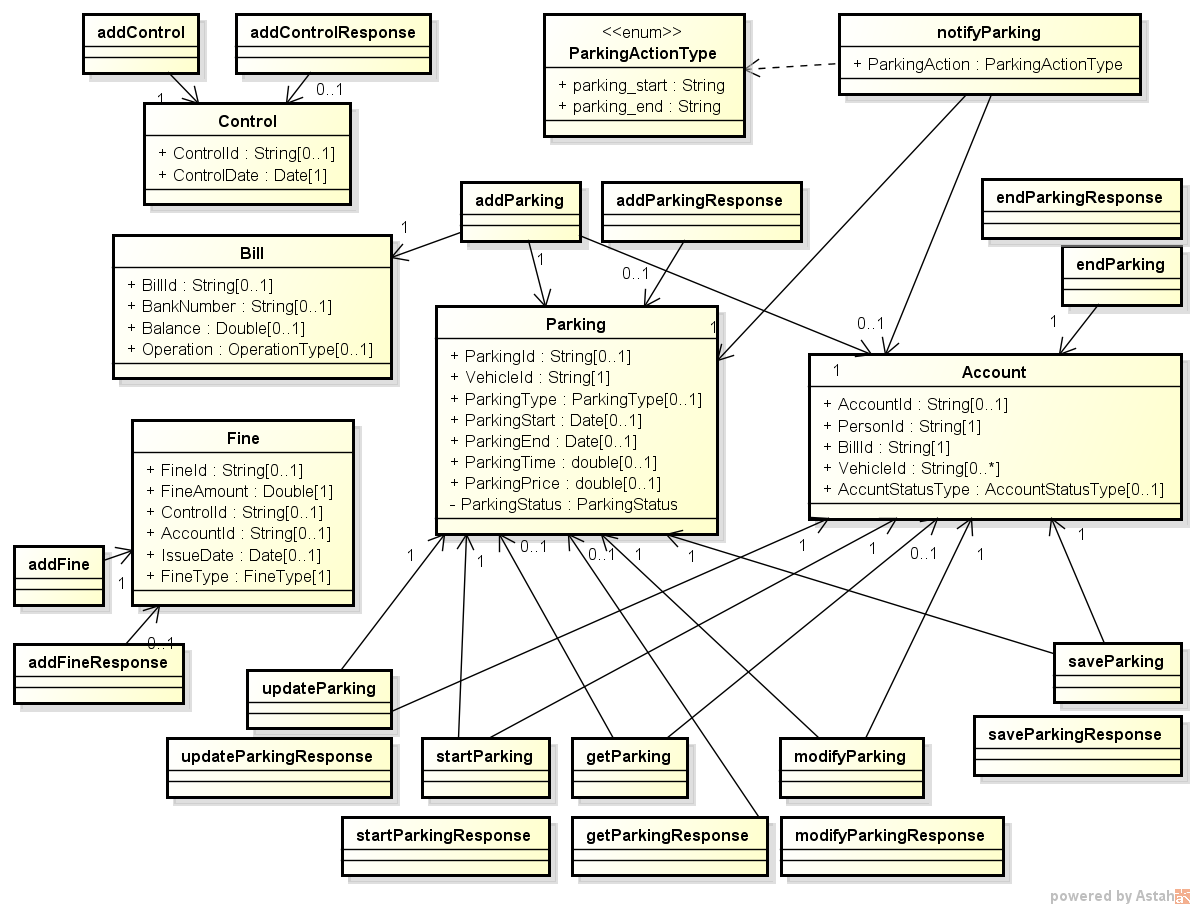
Rysunek 12 Diagram sekwencji operacji ParkingService.endParking

Usługi *startParking* i *endParking* są określane jako asynchroniczne ponieważ realizują dwa niezależne wątki – pierwszy przyjmuje zlecenie, zapisuje je na kolejce i synchronicznie odpowiada konsumentowi; drugi pobiera zlecenie z kolejki, obsługuje je i wysyła notyfikację do konsumenta. Taki sposób przetwarzania komunikatów jest zalecany przez standardy SOA dla zapewnienia gwarancji dostarczenia (w przypadku błędów aplikacji lub chwilowej niedostępności systemu dawcy komunikat jest zapisany na kolejce i jego obsługa może być ponawiana) oraz w celu lepszego odseparowania konsumentów od dawców, np. jako zabezpieczenie przed przeciążeniem systemu dawcy zbyt dużym wolumenem komunikatów.

W warstwie konektor usługa zawiera szereg operacji na potrzeby obsługi biletów parkingowych w bazie danych, wysyłki informacji do Aplikacji parkingowej o parkowaniu, kontrolach i mandatach oraz notyfikacji użytkownika:

1. *notifyParkingConnector* – wysyła powiadomienia REST do Aplikacji kierowcy o rozpoczętym lub zakończonym parkowaniu zgodnie z kontraktem zamieszczonym niżej.
2. *addParkingConnector* – komunikuje się z Aplikacją parkingową w celu przekazania informacji o rozpoczętym parkowaniu.
3. *modifyParkingConnector* – komunikuje się z Aplikacją parkingową w celu przekazania informacji o zakończonym parkowaniu.
4. *saveParkingConnector* – zapisuje bilet parkingowy w bazie danych.
5. *getParkingConnector* – pobiera bilet parkingowy z bazy danych.
6. *updateParkingConnector* – aktualizuje bilet parkingowy w bazie danych.
7. *finishParkingConnector* – operacja pomocnicza pracująca niezależnie od pozostałych usług zgodnie ze skonfigurowanym harmonogramem (uruchamiana automatycznie co 60 sekund), której zadaniem jest kończenie parkowania dla biletów którym wyczerpał się czas obowiązywania. Operacja dezaktywuje w bazie danych bilety dla których minęła data obowiązywania.
8. *addControlConnector* - komunikuje się z Aplikacją parkingową w celu przekazania informacji o nowej kontroli, operacja używana przez usługę ControlService.
9. *addFineConnector* - komunikuje się z Aplikacją parkingową w celu przekazania informacji o wystawionym mandacie, operacja używana przez usługę ControlService.

Model danych usługi ParkingService korzysta z encji Account, Bill, Parking, Control, Fine:



Rysunek 13 Model danych usługi ParkingService

Kontrakt usługi ParkingService w zakresie operacji startParking dla konsumenta Aplikacja kierowcy:

|  |  |
| --- | --- |
| **ParkingServiceAdapter.startParkingAdapter REST API** | |
| Metoda HTTP | POST |
| Adres HTTP | http://eparking.client.app:8084/api/parking/start/account/<ID\_KONTA> |
| Nagłówki HTTP | Content-Type: application/json |
| Request Body – Parking na czas | {  "parking\_type":"<PARKING\_TYP\_BILETU:tekst:time>",  "parking\_time":"<PARKING\_CZAS:liczba>",  "province":"<PARKING\_ADRES\_WOJEWODZTWO:tekst>",  "postal\_code":"<PARKING\_ADRES\_KOD\_POCZTOWY:licznba:NN-NN>",  "city":"<PARKING\_ADRES\_MIASTO:tekst>",  "street":"<PARKING\_ADRES\_ULICA:tekst>",  "street\_number":"<PARKING\_ADRES\_ULICA\_NUMER:liczba>"  } |
| Request Body – Parking do godziny | {  "parking\_type":"<PARKING\_TYP\_BILETU:tekst:deadline>",  "parking\_end":"<PARKING\_DATAGODZINA\_KONCA:dataczas>",  "province":"<PARKING\_ADRES\_WOJEWODZTWO:tekst>",  "postal\_code":"<PARKING\_ADRES\_KOD\_POCZTOWY:licznba:NN-NN>",  "city":"<PARKING\_ADRES\_MIASTO:tekst>",  "street":"<PARKING\_ADRES\_ULICA:tekst>",  "street\_number":"<PARKING\_ADRES\_ULICA\_NUMER:liczba>"  } |
| Request Body – Parking za kwotę | {  "parking\_type":"<PARKING\_TYP\_BILETU:tekst:free>",  "parking\_time":"<PARKING\_CENA:liczba>",  "province":"<PARKING\_ADRES\_WOJEWODZTWO:tekst>",  "postal\_code":"<PARKING\_ADRES\_KOD\_POCZTOWY:licznba:NN-NN>",  "city":"<PARKING\_ADRES\_MIASTO:tekst>",  "street":"<PARKING\_ADRES\_ULICA:tekst>",  "street\_number":"<PARKING\_ADRES\_ULICA\_NUMER:liczba>"  } |
| Response – Sukces | HTTP Response code: 200  Content-length: 0 |
| Notyfikacja - Sukces | Zgodnie z kontraktem ParkingServiceConnector.notifyParkingConnector REST API. |
| Notyfikacja – Błąd | Zgodnie z kontraktem ParkingServiceConnector.notifyParkingConnector REST API. |

Kontrakt usługi ParkingService w zakresie operacji endParking dla konsumenta Aplikacja kierowcy:

|  |  |
| --- | --- |
| **ParkingServiceAdapter.endParkingAdapter REST API** | |
| Metoda HTTP | POST |
| Adres HTTP | http://eparking.client.app:8084/api/parking/end/account/<ID\_KONTA> |
| Nagłówki HTTP | - |
| Request Body | - |
| Response – Sukces | HTTP Response code: 200  Content-length: 0 |
| Notyfikacja - Sukces | Zgodnie z kontraktem ParkingServiceConnector.notifyParkingConnector REST API. |
| Notyfikacja – Błąd | Zgodnie z kontraktem ParkingServiceConnector.notifyParkingConnector REST API. |

Kontrakt usługi ParkingService w zakresie operacji notifyParking dla dawcy Aplikacja kierowcy:

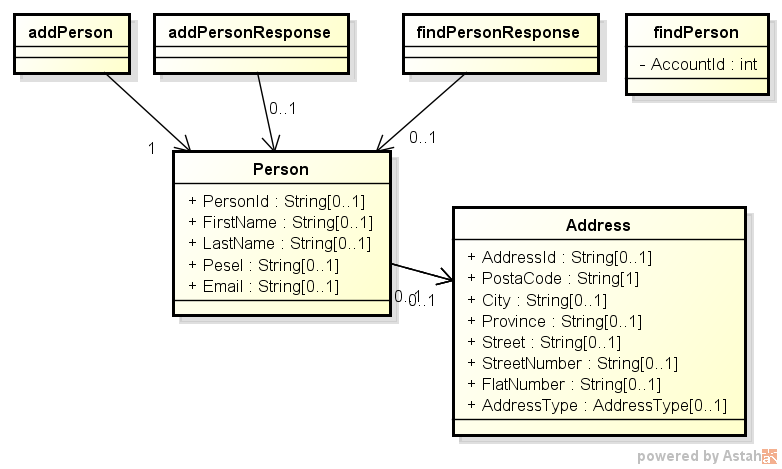
|  |  |
| --- | --- |
| **ParkingServiceConnector.notifyParkingConnector REST API** | |
| Metoda HTTP | POST |
| Adres HTTP | **[Błąd! Nieprawidłowy odsyłacz typu hiperłącze.](http://<HOST>:<PORT>/api/account/<KONTO_ID>/notify/<PARKING_ID>/<PARKING_AKCJA)**>  <HOST> - adres hosta interfejsu Aplikacji kierowcy  <PORT> - numer portu interfejsu Aplikacji kierowcy  <KONTO\_ID> - identyfikator konta użytkownika  <PARKING\_ID> - identyfikator biletu parkingowego w przypadku pozytywnego rozpoczęcia lub zakończenia parkingu, kod „parking\_error” w przypadku wystąpienia wyjątku  <PARKING\_AKCJA> - „parking\_start” dla rozpoczynania, „parking\_end” dla kończenia parkowania |
| Nagłówki HTTP | - |
| Request Body | - |
| Response – Sukces | HTTP Response code: 200  Content-length: 0 |
| Response – Błąd | Nie dotyczy – operacja jednokierunkowa. |

#### Usługa PersonService

Usługa operuje na obiekcie Kierowca (encje Person i Adress Kanonicznego modelu danych) umożliwiając dodanie i wyszukanie kierowcy (użytkownika aplikacji Aplikacja kierowcy) w bazie danych. Usługa posiada tylko operacje w warstwie konektora (używane przez inne usługi opisane wyżej):

1. *addPersonConnector* – zapisuje rekord kierowcy i adresu w bazie danych.
2. *findPersonConnector* – wyszukuje w bazie danych imię, nazwisko i adres email kierowcy dla zadanego identyfikatora konta; funkcjonalność wykorzystywana przez usługę ParkingService przy notyfikacji email o wystawieniu mandatu.

Model danych usługi PersonService korzysta z encji Person i powiązanej z nią encji Address.



Rysunek 14 Model danych usługi PersonService

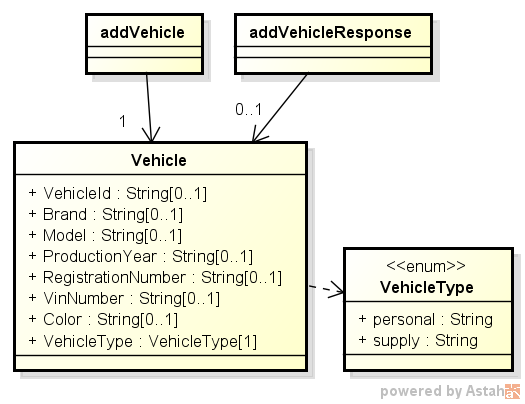
Ze względu na brak operacji w warstwie adaptera dla usługi nie są zdefiniowane żadne kontrakty.

#### Usługa VehicleService

Usługa obsługuje obiekt Pojazd (encja Vehicle w KMD) umożliwiając dodanie pojazdu w bazie danych podczas rejestracji konta użytkownika. Usługa posiada tylko operacje w warstwie konektora:

1. *addVehicleConnector* – zapisuje rekord pojazdu w bazie danych.

Model danych usługi PersonService korzysta z encji Person i powiązanej z nią encji Address.

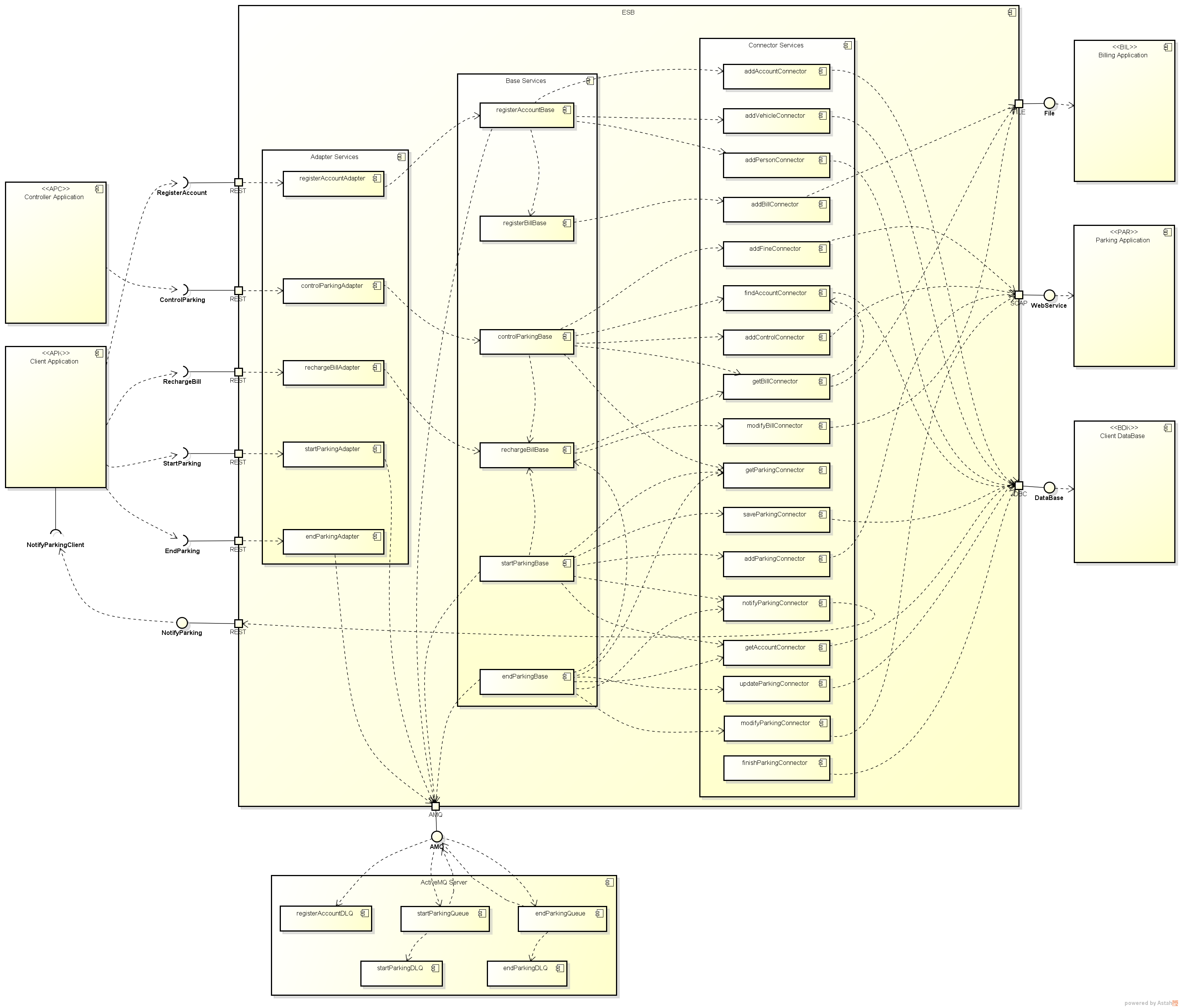


Rysunek 15 Model danych usługi VehicleService

Ze względu na brak operacji w warstwie adaptera dla usługi nie są zdefiniowane żadne kontrakty.

### Architektura logiczna

Poniższy rysunek prezentuje diagram komponentów zawierający szczegółową mapę logicznych zależności między integrowanymi systemami a operacjami w usługach na magistrali serwisowej. Po lewej stronie diagramu znajdują się konsumenci usług – Aplikacja kierowcy i Aplikacja kontrolera. Po prawej stronie umieszczeni są dawcy dla usług – Baza danych, Aplikacja parkingowa i zasób plikowy współdzielony z Aplikacją księgową. Centralne miejsce diagramu zajmuje komponent magistrali serwisowej z poszczególnymi operacjami zgrupowanymi w warstwy adapter, base, konektor. Strzałki prezentują powiązania interfejsów wejściowych i wyjściowych szyny z obsługującymi je operacjami. Zgodnie z wzorcem architektury trójwarstwowej operacje adapterowe wystawiają interfejsy wejściowe, operacje konektory zapewniają dostęp do systemów dawców a operacje bazowe odpowiadają za logiczne powiązanie jednych i drugich.



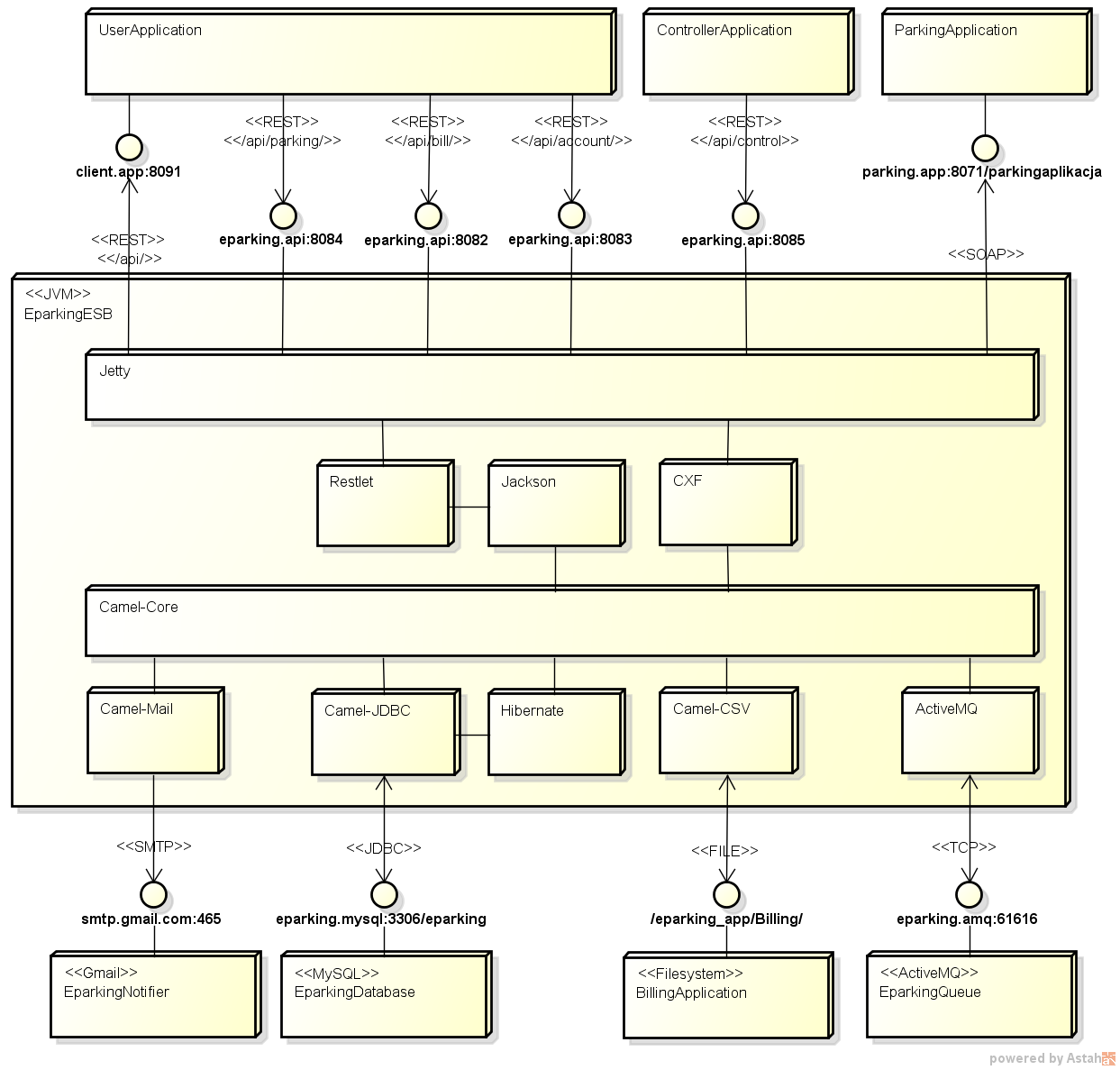
Rysunek 16 Niskopoziomowa architektura logiczna integracji

Na dole diagramu został umieszczony komponent brokera komunikacyjnego z kolejkami dedykowanymi dla poszczególnych operacji: startParkingQueue, endParkingQueue oraz kolejkami DLQ (ang. *Dead Letter Queue*), które zostaną omówione w rozdziale poświęconym obsłudze błędów.

### Architektura fizyczna

Podsumowaniem rozdziału dotyczącego projektu technicznego integracji jest diagram architektury fizycznej. Znajdują się na nim kluczowe komponenty magistrali serwisowej odpowiadające za komunikację z jej otoczeniem:

1. **Jetty** – wystawiający i konsumujący interfejsy Aplikacji kierowcy, kontrolera i parkingowej:
   1. <http://eparking.api:8082/api/bill/>
   2. <http://eparking.api:8083/api/account/>
   3. <http://eparking.api:8084/api/parking/>
   4. <http://eparking.api:8085/api/control>
   5. <http://client.app:8091/api/account/>
   6. <http://parking.app:8071/parkingaplikacja?wsdl>
2. **ActiveMQ** – interfejs do serwera ActiveMQ <tcp://eparking.amq:61616>.
3. **JDBC** –interfejs do bazy danych MySQL <jdbc://eparking.mysql:3306/eparking>.
4. **Camel-CSV** – zapewniający komunikację z zasobem plikowym [/eparking\_app/Billing/](file:///C:\eparking_app\Billing\).
5. **Camel-Mail** – komunikujący się z serwerem SMTP <smtps://smtp.gmail.com:465>.



Rysunek 17 Architektura fizyczna systemu

# Realizacja projektu

## Raport z testów funkcjonalnych

Testy funkcjonalne zostały przeprowadzone na bazie zdefiniowanych przypadków testowych, opisujących realne scenariusze działania aplikacji i interakcji z jej otoczeniem. Testy weryfikują scenariusze pozytywne, dotyczące prawidłowego działania aplikacji, oraz scenariusze negatywne, wynikające z błędów pochodzących z otoczenia aplikacji takich jak niepoprawne wywołania, braki dostępu i błędne odpowiedzi z systemów zewnętrznych. Raport z testów ma na celu potwierdzenie, że aplikacja jest poprawna funkcjonalnie i działa zgodnie ze specyfikacją.

**TEST-11: Poprawna rejestracja konta**

Test mający na celu weryfikację możliwości poprawnego zarejestrowania konta nowego użytkownika. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne i unikalne dane użytkownika i pojazdu. W odpowiedzi aplikacja zwróciła poprawne dane obiektów zarejestrowanych w systemach wewnętrznych. Aplikacja poprawnie utworzyła rekordy bazodanowe oraz zasób plikowy dla danych rachunkowych. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-11\_Poprawna rejestracja konta.docx.

**TEST-12: Błąd unikalności danych konta**

Test sprawdzający weryfikację unikalności danych konta przy próbie rejestracji nowego użytkownika. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne i nieunikalne dane użytkownika i pojazdu. W odpowiedzi aplikacja zwróciła informację o błędzie wynikającym z braku unikalności konta. Żadne rekordy bazodanowe nie zostały zapisane i nie został utworzony zasób plikowy. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-12\_Błąd unikalności danych konta.pdf.

**TEST-13: Negatywna walidacja danych wejściowych do rejestracji konta**

Test sprawdzający walidację danych wejściowych przy próbie zarejestrowania konta nowego użytkownika. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało niepoprawne dane użytkownika z błędną długością pola pesel. W odpowiedzi aplikacja zwróciła informację o błędzie wynikającym z negatywnej walidacji pola pesel względem wyrażenia regularnego. Żadne rekordy bazodanowe nie zostały zapisane i nie został utworzony zasób plikowy. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-13\_Negatywna walidacja danych wejściowych do rejestracji konta.pdf.

**TEST-14: Brak dostępu do bazy danych podczas rejestracji konta**

Test sprawdzający obsługę sytuacji wyjątkowej w postaci braku dostępu aplikacji do wewnętrznej bazy danych. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane użytkownika i pojazdu. Na potrzeby testu został wyłączony serwer bazodanowy MySQL. W odpowiedzi aplikacja zwróciła informację o błędzie wynikającym z braku dostępu do bazy danych. Żadne rekordy bazodanowe nie zostały zapisane i nie został utworzony zasób plikowy. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-14\_Brak dostępu do bazy danych podczas rejestracji konta.pdf.

**TEST-21: Poprawne zasilenie konta**

Test mający na celu weryfikację możliwości poprawnego zasilenia środkami pieniężnymi konta istniejącego użytkownika. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawny identyfikator konta rachunkowego i zasilaną kwotę. W odpowiedzi aplikacja zwróciła aktualny stan środków na koncie rachunkowym użytkownika. Aplikacja poprawnie zaktualizowała zasób plikowy dla danych rachunkowych. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-21\_Poprawne zasilenie konta.pdf.

**TEST-22: Brak konta o podanym identyfikatorze**

Test sprawdzający zachowanie aplikacji przy próbie zasilenia na nieistniejące w systemie konto. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało nieistniejący w systemie identyfikator konta rachunkowego i zasilaną kwotę. W odpowiedzi aplikacja zwróciła błąd związany z brakiem konta o podanym identyfikatorze. Aplikacja nie zaktualizowała żadnego zasobu plikowego. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-22\_Brak konta o podanym identyfikatorze.pdf.

**TEST-23: Negatywna walidacja danych wejściowych do zasilenia konta**

Test sprawdzający walidację danych wejściowych przy próbie zasilenia konta użytkownika. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało niepoprawną wartość kwoty zasilenia. W odpowiedzi aplikacja zwróciła informację o błędzie wynikającym z negatywnej walidacji. Aplikacja nie zaktualizowała żadnego zasobu plikowego. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-23\_Negatywna walidacja danych wejściowych do zasilenia konta.pdf.

**TEST-31: Poprawne rozpoczęcie parkowania na określony czas**

Test sprawdzający rejestrację w systemach wewnętrznych akcji rozpoczęcia parkowania na określony czas. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane zlecenia rozpoczęcia parkowania. W odpowiedzi aplikacja zwróciła informację o powodzeniu przyjęcia zlecenia. Aplikacja poprawnie zaktualizowała zasób plikowy dla danych rachunkowych, zarejestrowała rekord parkowania w bazie danych oraz przekazała informację do aplikacji parkingowej. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-31\_Poprawne rozpoczęcie parkowania na określony czas.pdf.

**TEST-32: Poprawne rozpoczęcie parkowania za określoną opłatę**

Test sprawdzający rejestrację w systemach wewnętrznych akcji rozpoczęcia parkowania za określoną opłatę. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane zlecenia rozpoczęcia parkowania. W odpowiedzi aplikacja zwróciła informację o powodzeniu przyjęcia zlecenia. Aplikacja poprawnie zaktualizowała zasób plikowy dla danych rachunkowych, zarejestrowała rekord parkowania w bazie danych oraz przekazała informację do aplikacji parkingowej. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-32\_Poprawne rozpoczęcie parkowania za określoną opłatę.pdf.

**TEST-33: Poprawne rozpoczęcie parkowania do określonej godziny**

Test sprawdzający rejestrację w systemach wewnętrznych akcji rozpoczęcia parkowania do podanej godziny. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane zlecenia rozpoczęcia parkowania. W odpowiedzi aplikacja zwróciła informację o powodzeniu przyjęcia zlecenia. Aplikacja poprawnie zaktualizowała zasób plikowy dla danych rachunkowych, zarejestrowała rekord parkowania w bazie danych oraz przekazała informację do aplikacji parkingowej. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-33\_Poprawne rozpoczęcie parkowania do określonej godziny.pdf.

**TEST-34: Poprawne zakończenie parkowania**

Test sprawdzający rejestrację w systemach wewnętrznych akcji zakończenia parkowania dla danego konta użytkownika. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane zlecenia rozpoczęcia parkowania. W odpowiedzi aplikacja zwróciła informację o powodzeniu przyjęcia zlecenia. Aplikacja poprawnie zaktualizowała zasób plikowy dla danych rachunkowych, zarejestrowała rekord parkowania w bazie danych oraz przekazała informację do aplikacji parkingowej. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-34\_Poprawne zakończenie parkowania.pdf.

**TEST-35: Negatywna odpowiedź z aplikacji obsługującej parkowanie**

Test sprawdzający obsługę błędu zwróconego z aplikacji parkingowej przy próbie dodania nowego parkowania. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane zlecenia rozpoczęcia parkowania. W odpowiedzi aplikacja zwróciła informację o powodzeniu przyjęcia zlecenia. Aplikacja, po otrzymaniu błędu z aplikacji parkingowej, wykonała wycofanie naliczenia opłaty parkingowej, żaden rekord parkingu nie został dodany do bazy danych, została wysłana notyfikacja do aplikacji użytkownika końcowego o braku powodzenia rozpoczęcia parkingu. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-35\_Negatywna odpowiedź z aplikacji obsługującej parkowanie.pdf.

**TEST-36: Błąd aplikacji obsługującej parkowanie**

Test sprawdzający obsługę braku komunikacji z aplikacją parkingową przy próbie dodania nowego parkowania. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów klienckich obsługiwanych przez użytkowników końcowych. Wywołanie zawierało poprawne dane zlecenia rozpoczęcia parkowania. W celu przeprowadzenia testu została wyłączona zaślepka symulująca aplikację parkingową. W odpowiedzi aplikacja zwróciła informację o powodzeniu przyjęcia zlecenia. Aplikacja, po wykryciu błędu z aplikacji parkingowej, wykonała wycofanie naliczenia opłaty parkingowej, żaden rekord parkingu nie został dodany do bazy danych, została wysłana notyfikacja do aplikacji użytkownika końcowego o braku powodzenia rozpoczęcia parkingu. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-36\_Błąd aplikacji obsługującej parkowanie.pdf.

**TEST-41: Poprawna kontrola parkowania bez wystawienia mandatu**

Test sprawdzający rejestrację w systemach wewnętrznych informacji o kontroli parkowania z aplikacji kontrolera. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów obsługujących rejestrację kontroli. Wywołanie zawierało poprawne dane o kontroli. W odpowiedzi aplikacja zwróciła informację o powodzeniu rejestracji kontroli. Aplikacja poprawnie przekazała informację o kontroli do aplikacji parkingowej. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-41\_Poprawna kontrola parkowania bez wystawienia mandatu.pdf.

**TEST-42: Poprawna kontrola parkowania z wystawieniem mandatu**

Test sprawdzający rejestrację w systemach wewnętrznych informacji o kontroli parkowania z wystawieniem mandatu. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów obsługujących rejestrację kontroli. Wywołanie zawierało poprawne dane o kontroli. Na potrzeby testu zostało zarejestrowanie rozpoczęcie i zakończenie parkowania, dane kontroli dotyczyły pojazdu dla którego zostało uprzednio zakończone parkowanie. W odpowiedzi aplikacja zwróciła informację o powodzeniu rejestracji kontroli oraz o wystawieniu mandatu. Aplikacja poprawnie naliczyła mandat jako aktualizację rachunku na zasobie plikowym, przekazała informacje o kontroli i wystawieniu mandatu do aplikacji parkingowej. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-42\_Poprawna kontrola parkowania z wystawieniem mandatu.pdf.

**TEST-43: Negatywna walidacja danych wejściowych do kontroli parkowania**

Test sprawdzający walidację danych wejściowych przy próbie rejestracji kontroli. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów obsługujących rejestrację kontroli. Wywołanie zawierało puste wartości kodu pocztowego i województwa w adresie kontroli. W odpowiedzi aplikacja zwróciła informację o błędzie wynikającym z negatywnej walidacji. Aplikacja nie zaktualizowała żadnego zasobu plikowego oraz nie przekazała żadnych informacji do systemów wewnętrznych. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-43\_Negatywna walidacja danych wejściowych do kontroli parkowania.pdf.

**TEST-44: Negatywna odpowiedź z aplikacji obsługującej kontrole**

Test sprawdzający obsługę błędu zwróconego z aplikacji parkingowej przy próbie dodania nowej kontroli. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów obsługujących rejestrację kontroli. Wywołanie zawierało poprawne dane o kontroli. W odpowiedzi aplikacja zwróciła informację o błędzie związanym z procesowaniem informacji o kontroli. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-44\_Negatywna odpowiedź z aplikacji obsługującej kontrole.pdf.

**TEST-45: Błąd aplikacji obsługującej kontrole**

Test sprawdzający obsługę błędu komunikacji zwróconego z aplikacji parkingowej przy próbie dodania nowego mandatu. W celu przeprowadzenia testu została wywołana usługa REST wystawiana przez aplikację dla zewnętrznych systemów obsługujących rejestrację kontroli. Wywołanie zawierało poprawne dane o kontroli. W odpowiedzi aplikacja zwróciła informację o błędzie związanym z procesowaniem informacji o mandacie. Aplikacja, po wykryciu błędu z aplikacji parkingowej, wykonała wycofanie naliczenia mandatu. Raport z testu i logi z wykonania usługi znajdują się w załączniku TEST-45\_Błąd aplikacji obsługującej kontrole.pdf.

## Raport z testów wydajnościowych

Testy wydajnościowe miały na celu sprawdzenie kompletnego rozwiązania pod kątem jakości i wydajności. Zostały wykonane dwa typy testów: weryfikujące stabilność rozwiązania przy średnim obciążeniu w długim okresie oraz testy wytrzymałości, sprawdzające rozwiązanie przy maksymalnym obciążeniu w krótkim czasie. Niniejszy rozdział zawiera informacje o konfiguracji sprzętowej środowiska testowego, przyjęte założenia wolumetryczne i opis realizacji testów. Finalny raport prezentuje w formie wykresów profil wydajnościowy aplikacji w ujęciu ilościowym (obsłużone wywołania w jednostce czasu) i jakościowym (czasy odpowiedzi poszczególnych operacji). Wykresy zostały opracowane na bazie raportów JMeter „Results Table”. Różnice w opracowanych wynikach w porównaniu z przedstawionymi raportami JMeter „Aggregate Report” wynikają z różnych metodyk wyznaczania obciążenia. JMeter wykonuje jednorazową operację dzielenia ilości próbek przez czas testu w sekundach. Opracowane przez autora profile agregują ilości wywołań w interwałach sekundowych w których realnie wystąpiły próbki (ilości wywołań na sekundę oraz średnie czasy odpowiedzi na sekundę) po czym uśredniają wartości za pomocą linii trendu.

Przeprowadzone testy wykazały poprawne działanie aplikacji i całej platformy. Pod kątem stabilności – obsłużenie 400 tyś. komunikatów w trakcie 8 godzin przy średnim obciążeniu 10 wywołań na sekundę. Pod kątem wytrzymałości – 48 tyś. komunikatów przez 8 minut z obciążeniem 115 wywołań na sekundę. Zdaniem autora raportowane niżej wyniki są bardzo pozytywne i świadczą o stabilności użytego rozwiązania oraz dobrej jakości napisanego kodu.

### Założenia i konfiguracja

W celu urealnienia wyników testowane były równolegle wszystkie dostępne usługi. Jedynym odstępstwem od tej zasady był brak usługi AccountService w teście stabilności. Konta zostały zarejestrowane we wcześniejszym przebiegu w ramach przygotowania danych testowych. Rejestracja przebiegła wydajnie (800 tyś. kont w 1 godz.) w związku z czym operacja AccountService.registerAccount została wyłączona z dalszego zakresu testowania stabilności.

|  |  |  |
| --- | --- | --- |
| **Założenie** | **Test stabilności** | **Test wytrzymałości** |
| Ilość równoległych sesji | 11 użytkowników | 120 użytkowników |
| Czas narastania | 60 sek. | 60 sek. |
| Długość testu | 8 godz. | 8 min. |
| Ilość komunikatów | 400 tyś. | 48 tyś. |
| Średni rozmiar komunikatu | 1.2 KB | 1.2 KB |
| Czas narastania obciążenia | 60 sek. | 60 sek. |
| Średni czas odpowiedzi aplikacji parkingowej | 0 – 10 msek. | 0 – 50 msek. |
| Średnia odległość między wywołaniami | 1 – 2 sek. | 1 – 5 sek. |

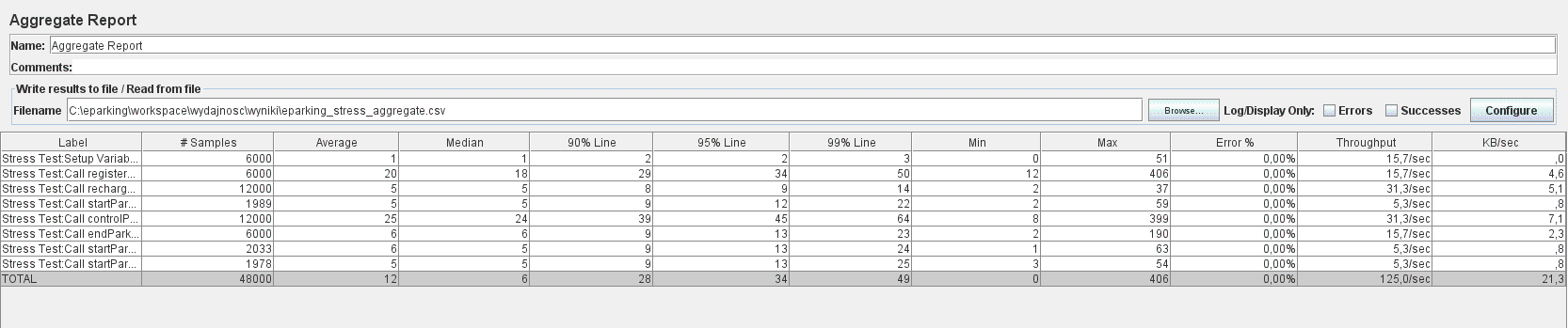
Tabela 5 Założenia do testów wydajności

|  |  |
| --- | --- |
| **Parametry środowiska testowego** | **Wartość** |
| Ilość rdzeni procesora | 6 CPU 2.4 GHz |
| Wielkość pamięci RAM | 12 GB 1 GHz |
| Typ pamięci masowej | SSD 550/510 odczyt/zapis MB/sek. |
| Uruchomione procesy | Aplikacja integracyjna Apache Camel  Serwer kolejek ActiveMQ  Baza Danych MySQL  Generator ruchu JMeter  Aplikacja testowa SoapUI |

Tabela 6 Środowisko do testów wydajności

### Raport z testów wytrzymałości

Poniższy raport prezentuje agregację wyników przebiegu testowego testu wytrzymałości w narzędziu JMeter (Aggregate Report). Test składał się z 48 tyś. próbek; obciążenie wyniosło 125 wywołań na sekundę. Mediana czasów odpowiedzi wyniosła 6 milisekund; linia 90% (90% próbek uzyskało odpowiedź w krótszym czasie, 9 decyl) wyniosła 28 milisekund. Najistotniejszą obserwacją jest wyraźnie odbiegająca od średniej charakterystyka operacji ParkingService.controlParking - mediana czasów odpowiedzi wyniosła 24 milisekundy przy linii 90% na poziomie 39 milisekund. Również operacja AccountService.registerAccount miała podwyższone charakterystyki - mediana czasów odpowiedzi wyniosła 18 milisekund przy linii 90% na poziomie 39 milisekund.



Rysunek 18 Raport JMeter dla testu wytrzymałości

Poniższy wykres obrazuje średnie obciążenie dla testu wytrzymałości. Na osi odciętych umieszczony jest czas trwania testu, na osi rzędnych ilość wywołań na sekundę. Linia trendu (funkcja liniowa) jest praktycznie płaska na poziomie 115 wywołań na sekundę.



Rysunek 19 Średnie obciążenie dla testu wytrzymałości

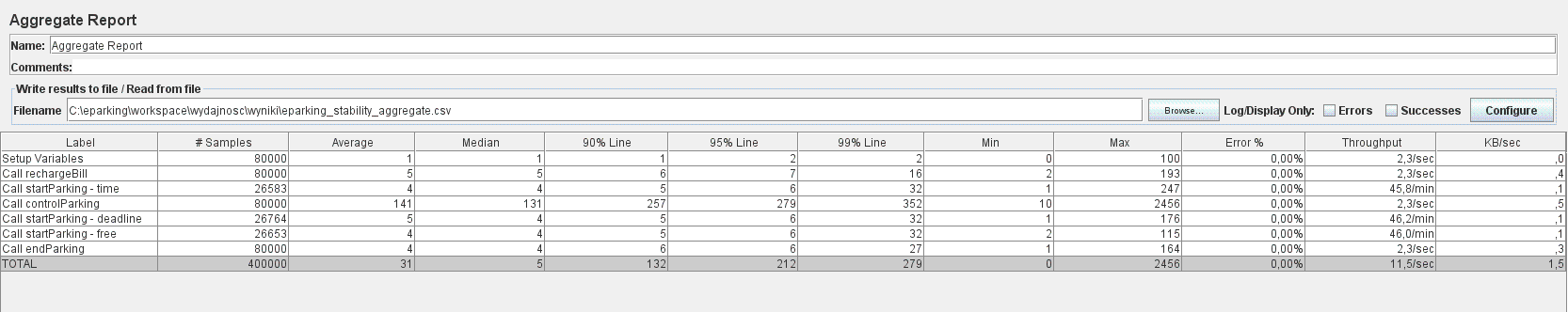
Kolejny wykres prezentuje średnie czasy odpowiedzi dla pełnego zestawu usług. Na osi odciętych umieszczony jest czas trwania testu, na osi rzędnych średni czas odpowiedzi. Charakterystyka wykresu wskazuje na stabilne zachowanie operacji asynchronicznych w usłudze ParkingService: startParking i endParking oraz operacji BillService.rechargeBill – linie trendu na poziomie ok. 5 milisekund. Większe, ale również nieznaczne, odchylenia można zaobserwować dla złożonych operacji synchronicznych AccountService.registerAccount i ControlService.controlParking. Pierwsza z nich ma płaską linię trendu na poziomie 20 milisekund co jest akceptowalną wartością. Druga wykazuje negatywny liniowy wzrost od 13 do 37 milisekund w trakcie testu. Szczegółowa analiza charakterystyk operacji ControlService.controlParking została przedstawiona poniżej w raporcie z testów stabilności.



Rysunek 20 Czasy odpowiedzi dla testu wytrzymałości

### Raport z testów stabilności

Poniższy raport prezentuje agregację wyników przebiegu testowego testu stabilności w narzędziu JMeter (Aggregate Report). Test składał się z 400 tyś. próbek; obciążenie wyniosło 11.5 wywołania na sekundę (mierzone jako ilość próbek / czas testu). Mediana czasów odpowiedzi wyniosła 5 milisekund; linia 90% wyniosła 132 milisekundy. Analogicznie jak w raporcie z testu wytrzymałości odbiegały od średniej parametry operacji ControlService.controlParking – mediana czasów odpowiedzi wyniosła 131 milisekund przy linii 90% na poziomie 257 milisekund.



Rysunek 21 Raport JMeter dla testu stabilności

Poniższy wykres przedstawia obciążenie w teście stabilności. Na osi odciętych umieszczony jest czas trwania testu, na osi rzędnych ilość wywołań na sekundę. Linia trendu (funkcja liniowa) jest praktycznie pozioma i znajduje się na poziomie 9.8 wywołania na sekundę.



Rysunek 22 Średnie obciążenie dla testu stabilności

Kolejne dwa wykresy prezentują średnie czasy odpowiedzi usług. Pierwszy wykres dotyczy operacji ParkingService.startParking, ParkingService.endParking i BillService.rechargeBill. Wykres drugi przedstawia usługę ControlService.controlParking. Na osi odciętych umieszczony jest czas trwania testu, na osi rzędnych średni czas odpowiedzi. Charakterystyka wykresu 1 wskazuje na stabilne działanie aplikacji. Zauważalny jest nieznaczny wzrost linii trendu (średnie pogorszenie charakterystyki o ok 1 milisekundę przez 8 godzin) lecz zdaniem autora jest to pomijalne przy tak niskich czasach odpowiedzi rzędu 4 – 6 milisekund. Korzystne wyniki dla usługi ParkingService wynikają z zastosowania komunikacji asynchronicznej z użyciem kolejek. W warstwie adapterowej komunikaty są umieszczane na serwerze ActiveMQ. Logika orkiestracji i komunikacja z systemami wewnętrznymi jest realizowana w warstwie bazowej i konektorach po zdjęciu komunikatów z kolejki. Ten aspekt działania operacji nie jest odzwierciedlony w wynikach testów, natomiast obserwacje logów systemów wewnętrznych (SoapUI, MySQL) nie wykazywały anomalii i pozwoliły stwierdzić poprawność i stabilność kompletnych przepływów.



Rysunek 23 Czasy odpowiedzi dla testu stabilności 1

Na poniższym wykresie widoczny jest liniowy przyrost czasów odpowiedzi z operacji ControlService.controlParking od 20 do 280 milisekund. Jest to niekorzystna charakterystyka obrazująca stałe pogarszanie jakości w trakcie testu. ControlService .controlParking wykonuje łącznie 4 odczyty bazodanowe, 3 operacje plikowe oraz 2 wywołania webservice. Wszystkie czynności są realizowane synchronicznie. Wykrycie przyczyny utraty jakości wymaga w pierwszej kolejności zbadania adaptera plikowego i bazodanowego (logi SoapUI z wywołań webservice wykazują stabilne działanie z odpowiedziami na poziomie kilku milisekund). W przypadku potwierdzenia braku opóźnień z systemów zewnętrznych autor rekomenduje refaktoring kodu operacji w dwóch aspektach: przejście na wzorzec komunikacji asynchronicznej z użyciem kolejki (w adapterze do aplikacji kontrolera oraz konektorze do aplikacji parkingowej) analogicznie do usługi ParkingService oraz wytworzenie i użycie dedykowanej operacji warstwy adapterowej do aktualizacji zasobu plikowego w miejsce reużywanej obecnie operacji BillService.rechargeBill z warstwy bazowej.



Rysunek 24 Czasy odpowiedzi dla testu stabilności 2

# Podsumowanie prac

# Indeks załączników

GoogleTrends Porównanie platform integracyjnych.png 6

TEST-11\_Poprawna rejestracja konta.docx 27

TEST-12\_Błąd unikalności danych konta.pdf 27

TEST-13\_Negatywna walidacja danych wejściowych do rejestracji konta.pdf 27

TEST-14\_Brak dostępu do bazy danych podczas rejestracji konta.pdf 28

TEST-21\_Poprawne zasilenie konta.pdf 28

TEST-22\_Brak konta o podanym identyfikatorze.pdf 28

TEST-23\_Negatywna walidacja danych wejściowych do zasilenia konta.pdf 28

TEST-31\_Poprawne rozpoczęcie parkowania na określony czas.pdf 29

TEST-32\_Poprawne rozpoczęcie parkowania za określoną opłatę.pdf 29

TEST-33\_Poprawne rozpoczęcie parkowania do określonej godziny.pdf 29

TEST-34\_Poprawne zakończenie parkowania.pdf 29

TEST-35\_Negatywna odpowiedź z aplikacji obsługującej parkowanie.pdf 29

TEST-36\_Błąd aplikacji obsługującej parkowanie.pdf 30

TEST-41\_Poprawna kontrola parkowania bez wystawienia mandatu.pdf 30

TEST-42\_Poprawna kontrola parkowania z wystawieniem mandatu.pdf 30

TEST-43\_Negatywna walidacja danych wejściowych do kontroli parkowania.pdf 30

TEST-44\_Negatywna odpowiedź z aplikacji obsługującej kontrole.pdf 31

TEST-45\_Błąd aplikacji obsługującej kontrole.pdf 31

# Indeks tabel

Tabela 1 Ocena porównawcza platform integracyjnych 7

Tabela 2 Kontrakt usługi AccountService 14

Tabela 3 Kontrakt usługi BillService 16

Tabela 4 Kontrakt usługi BillService 18

Tabela 5 Założenia do testów wydajności 31

Tabela 6 Środowisko do testów wydajności 31

# Indeks rysunków

[Rysunek 1 Diagram przypadków użycia 8](#_Toc458985833)

[Rysunek 2 Wysokopoziomowa architektura logiczna integracji 10](#_Toc458985834)

[Rysunek 3 Kanoniczny model danych 11](#_Toc458985835)

[Rysunek 4 Techniczny model danych 11](#_Toc458985836)

[Rysunek 5 Diagram sekwencji operacji AccountService.registerAccount 13](#_Toc458985837)

[Rysunek 6 Model danych usługi AccountService 13](#_Toc458985838)

[Rysunek 7 Diagram sekwencji operacji BillService.rechargeBill 15](#_Toc458985839)

[Rysunek 8 Model danych usługi BillService 15](#_Toc458985840)

[Rysunek 9 Diagram sekwencji operacji ControlService.controlParking 17](#_Toc458985841)

[Rysunek 10 Model danych usługi ControlService 18](#_Toc458985842)

[Rysunek 11 Diagram sekwencji operacji ParkingService.startParking 19](#_Toc458985843)

[Rysunek 12 Diagram sekwencji operacji ParkingService.endParking 20](#_Toc458985844)

[Rysunek 13 Model danych usługi ParkingService 21](#_Toc458985845)

[Rysunek 14 Model danych usługi PersonService 23](#_Toc458985846)

[Rysunek 15 Model danych usługi VehicleService 24](#_Toc458985847)

[Rysunek 16 Niskopoziomowa architektura logiczna integracji 25](#_Toc458985848)

[Rysunek 17 Architektura fizyczna systemu 26](#_Toc458985849)

[Rysunek 18 Raport JMeter dla testu wytrzymałości 32](#_Toc458985850)

[Rysunek 19 Średnie obciążenie dla testu wytrzymałości 32](#_Toc458985851)

[Rysunek 20 Czasy odpowiedzi dla testu wytrzymałości 33](#_Toc458985852)

[Rysunek 21 Raport JMeter dla testu stabilności 33](#_Toc458985853)

[Rysunek 22 Średnie obciążenie dla testu stabilności 34](#_Toc458985854)

[Rysunek 23 Czasy odpowiedzi dla testu stabilności 1 35](#_Toc458985855)

[Rysunek 24 Czasy odpowiedzi dla testu stabilności 2 36](#_Toc458985856)

# Bibliografia

|  |  |
| --- | --- |
| [1] | The Open Group, „Soa Source Book,” 2009. [Online]. Available: http://www.opengroup.org/soa/source-book/soa/index.htm. |
| [2] | G. Hohpe i B. Woolf, Enterprise Integration Patterns, Addison-Wesley Professional, 2003. |
| [3] | M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002. |
| [4] | T. Erl, Service-Oriented Architecture: Concepts, Technology & Design, Prentice Hall, 2005. |
| [5] | T. Erl, SOA Design Patterns, Prentice Hall, 2009. |
| [6] | T. Erl, SOA Principles of Service Design, Prentice Hall, 2008. |
| [7] | D. Chappell, Enterprise Service Bus, O'Reilly Media, 2004. |
| [8] | Software Engineering Institute, Carnegie Mellon University, „Architecting Service-Oriented Systems,” 2011. [Online]. Available: http://resources.sei.cmu.edu/asset\_files/TechnicalNote/2011\_004\_001\_15353.pdf. |
| [9] | A. Rotem-Gal-Oz, SOA Patterns, Manning Publications, 2012. |
| [10] | Arcitura Education, „SOAPatterns,” 2016. [Online]. Available: http://soapatterns.org. |
| [11] | N. Josuttis, SOA in Practice, O'Reilly Media, 2007. |
| [12] | IBM DeveloperWorks Technical Library, „The value of applying the canonical modeling pattern,” 2008. [Online]. Available: http://www.ibm.com/developerworks/data/library/techarticle/dm-0803sauter/dm-0803sauter-pdf.pdf. |
| [13] | Organization for the Advancement of Structured Information Standards (OASIS), „UDDI Version 3.0.2,” 2004. [Online]. Available: http://www.uddi.org/pubs/uddi\_v3.htm. |
| [14] | Institute of Electrical and Electronics Engineers (IEEE), Standard Computer Dictionary 610, IEEE Press, 1991. |
| [15] | The Apache Software Foundation, „Apache ActiveMQ,” 2016. [Online]. Available: http://activemq.apache.org. |
| [16] | Object Management Group (OMG), „Business Process Model and Notation,” 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/PDF. |
| [17] | IBM DeveloperWorks Technical Library, „Integration architecture: Comparing web APIs with service-oriented architecture and enterprise application integration,” 2015. [Online]. Available: http://www.ibm.com/developerworks/websphere/library/techarticles/1503\_clark/1305\_clark-pdf.pdf. |
| [18] | M. Fisher, Spring Integration in Action, Manning Publications, 2012. |
| [19] | W3C Working Group, „Web Services Glossary,” [Online]. Available: https://www.w3.org/TR/ws-gloss. |

1. Interoperacyjność według *IEEE Standard Computer Dictionary 610* jest definiowana jako „zdolność dwóch lub większej liczby systemów informatycznych lub ich komponentów do wymiany informacji i do jej użycia” [14]. [↑](#footnote-ref-1)
2. System integrowany może być usługodawcą (ang. *Provider*), jeśli dostarcza funkcjonalność udostępnianą przez usługę, lub usługobiorcą (ang. *Consumer*), jeśli konsumuje tę funkcjonalność. W dalszej części niniejszej pracy autor będzie posługiwał się uproszczoną terminologią „dawca” oraz „konsument”. [↑](#footnote-ref-2)
3. Skrót *VETRO* zgodnie z *Enterprise Service Bus* Davida Chappela oznacza podstawowe funkcje usługi: walidacja komunikatów (ang. *validate*), wzbogacanie treści (ang. *enrich*), transformacja danych (ang. *transform*), trasowanie ruchu (ang. *routing*), aktywowanie systemów zewnętrznych (ang. *operate*) [8]. [↑](#footnote-ref-3)
4. Szczegółowy opis wzorców wymiany wiadomości (ang. *Message Exchange Patterns*, w skrócie MEP) można znaleźć w *SOA in Practice* Nicolaia Josuttisa [12]. [↑](#footnote-ref-4)
5. Opis cech brokera komunikacyjnego można znaleźć na stronie produktu Apache ActiveMQ http://activemq.apache.org/features.html [15]. [↑](#footnote-ref-5)
6. Standard UDDI w wersji 3 jest opisany w [1]. [↑](#footnote-ref-6)
7. Koncepcja Kanonicznego modelu danych została opisana w pracy *The value of applying the canonical modeling pattern in SOA* dostępnej w IBM DeveloperWorks Technical Library [13]. [↑](#footnote-ref-7)
8. *Apache Software License version 2.0* <https://opensource.org/licenses/Apache-2.0>. [↑](#footnote-ref-8)
9. *Common Development and Distribution License 1.0* <https://opensource.org/licenses/CDDL-1.0>. [↑](#footnote-ref-9)
10. Mapowanie produktów WSO2 i Apache znajduje na stronie dostawcy WSO2 <http://wso2.com/apache>. [↑](#footnote-ref-10)
11. Strona domowa Open ESB <http://www.open-esb.net> oraz OpenMQ <https://mq.java.net>. [↑](#footnote-ref-11)
12. Strona domowa projektu Apache ServiceMix <http://servicemix.apache.org>. [↑](#footnote-ref-12)
13. Dokumentacja Apache Camel <http://camel.apache.org/enterprise-integration-patterns.html> oraz WSO2 <https://docs.wso2.com/display/IntegrationPatterns/Enterprise+Integration+Patterns+with+WSO2+ESB>. [↑](#footnote-ref-13)
14. Dokumentacja Apache Camel <http://camel.apache.org/components.html> oraz WSO2 <https://docs.wso2.com/display/ESBCONNECTORS/WSO2+ESB+Connectors>. [↑](#footnote-ref-14)
15. Standard JBI 1.0 <http://download.oracle.com/otndocs/jcp/jbi-1.0-fr-eval-oth-JSpec>. [↑](#footnote-ref-15)
16. Opis WSO2 Developer Studio <http://wso2.com/library/articles/2014/06/develop-and-deploy-esb-artifacts-using-wso2-developer-studio>. [↑](#footnote-ref-16)
17. Konsole administracyjne dla Apache Camel i ActiveMQ <http://camel.apache.org/web-console.html>. [↑](#footnote-ref-17)
18. Zestawienie wyników Google Trends znajduje się w załączniku GoogleTrends Porównanie platform integracyjnych.png. [↑](#footnote-ref-18)
19. Zestawienie jest dostępne pod adresem <https://www.openhub.net/p/_compare?project_0=WSO2+ESB&project_1=Open+ESB&project_2=Apache+Camel>. [↑](#footnote-ref-19)
20. Forum Apache Camel <http://camel.465427.n5.nabble.com/Camel-Users-f465428.html>. [↑](#footnote-ref-20)
21. Definicje pojęć orkiestracja i choreografia można znaleźć w [19]. [↑](#footnote-ref-21)