Złożenie pracy online:
2022-01-29 10:35:59
Kod pracy:
13159/39073/CloudA

Sebastian Wypiór
(nr albumu: 24831)

Praca inżynierska

Platforma integracyjna wspomagająca branżę e-commerce w procesie dostawy towarów z wykorzystaniem Domain Driven Design oraz zasad czystej architektury.

Building integration platform for supporting e-commerce companies in their supply chain process using Clean Architecture and Domain Driven Design principles.

Wydział: Wydział Nauk Społecznych i Informatyki

Kierunek: Informatyka

Specjalność: programowanie aplikacji biznesowych

Promotor: dr Krzysztof Przybycień

Nowy Sącz, 2022

Pragnę złożyć najszczersze podziękowania promotorowi niniejszej pracy, dr Krzysztofowi Przybycieniowi, za poświęcony czas oraz cenne rady, a także za wsparcie i wyrozumiałość.

Streszczenie

Głównym celem pracy było zaprezentowanie wybranych zagadnień z dziedziny architektury oprogramowania na przykładzie serwisu umożliwiającego rejestrowanie przesyłek kurierskich. Projekt został stworzony w oparciu o zasady tak zwanej czystej architektury oraz z wykorzystaniem elementów Domain-Driven Design. Dzięki temu powstał w pełni funkcjonalny projekt podzielony na warstwy umożliwiający dalszą bezproblemową jego rozbudowę. W pracy starałem się przedstawić również najbardziej popularne zewnętrzne biblioteki oraz narzędzia wspomagające działanie całego rozwiązania.

Słowa kluczowe

net core, blazor, webapi, integracja, e-commerce, ddd, czysta architektura

Abstract

The main purpose of the work was to present selected topics in the field of software architecture based on the example project which is the service for registering parcels. The project was created with the usage of the so-called clean architecture principles and also with the use of Domain-Driven Design elements. As a result, a fully functional project was created, divided into layers, enabling its further trouble-free expansion. In the presented work, I tried to present the most popular external libraries and tools increasing functionality of the entire solution.

Keywords

net core, blazor, webapi, integration, e-commerce, ddd, clean architecture

Spis treści

1 Wstęp ... 3

1.1 Cel pracy ... 3

1.2 Geneza wyboru tematu ... 3

2 Charakterystyka użytych technologii ... 5

3 Zakres funkcjonalny projektu ... 8

3.1 Wymagania funkcjonalne oraz niefunkcjonalne projektu ... 8

3.2 Prezentacja działania rozwiązania ... 9

4 Implementacja ... 17

4.1 Część backendowa – WebApi ... 17

4.1.1 Czysta architektura – ujęcie teoretyczne ... 17

4.2 Implementacja czystej architektury w projekcie ... 18

4.2.1 Diagram architektury / klas ... 18

4.2.2 Diagram kooperacji komponentów aplikacji ... 20

4.2.3 Diagram sekwencji wybranego procesu ... 21

4.2.4 ShipmentService.Api ... 22

4.2.5 ShipmentService.Application ... 25

4.2.6 ShipmentService.Infra ... 27

4.2.7 ShipmentService.Persistence ... 28

4.2.8 ShipmentService.Domain ... 31

4.2.9 ShipmentService.UI ... 40

4.3 Zastosowanie dodatkowych narzędzi oraz bibliotek w projekcie ... 44

4.3.1 Seq oraz Serilog ... 44

4.3.2 RabbitMq ... 45

4.3.3 Docker ... 47

4.4 Testy jednostkowe ... 49

5 Podsumowanie 53

6 Bibliografia.. 54

1 Wstęp

1.1 Cel pracy

Jako cel pracy obrałem prezentację podejścia do tworzenia oprogramowania o nazwie Domain Driven Design oraz zasad czystej architektury poprzez zaprojektowanie oraz implementację serwisu umożliwiającego integrację oprogramowania e-commerce kontrahentów naszego potencjalnego klienta. Rozwiązanie jest przeznaczone dla firm działających w branży TSL (Transport, Spedycja, Logistyka).

Głównym zadaniem serwisu jest możliwość rejestrowania przesyłek bezpośrednio ze sklepów internetowych klientów firmy transportowej lub - jeżeli oprogramowanie klienta nie umożliwia takowej integracji – przy pomocy portalu internetowego oraz generowanie wydruków listów przewozowych.

1.2 Geneza wyboru tematu

Geneza wyboru tematu jest ściśle powiązana z moimi dotychczasowymi doświadczeniami zawodowymi. Żyjemy w czasach postępującej cyfryzacji co w naturalny sposób wymusza zapotrzebowanie na tworzenie nowego oprogramowania czy też rozwijanie i utrzymywanie dotychczasowych rozwiązań.

Towarzyszący temu niedobór odpowiednio wykształconych kadr czy też kadr w ogóle powoduje, że często mamy do czynienia z oprogramowaniem bardzo niskiej jakości. To implikuje oczywiste problemy w korzystaniu z takiego oprogramowania ale jeszcze większe związane z jego utrzymaniem oraz rozwojem. Często bywa, że źle zaprojektowanie oprogramowanie nie nadaje się do dalszego rozwoju co generuje znaczne koszty wynikające z konieczności zmiany oprogramowania bądź dostosowania procesów biznesowych do ograniczeń „wadliwego” oprogramowania.

Wprowadzenie odpowiednich zasad czystej architektury pozwala w znaczny sposób na zniwelowanie tego problemu. Przestrzeganie ustalonych konwencji umożliwia w zasadzie bezproblemowe rozwijanie produktu oraz modyfikowanie jego działających już funkcjonalności. Oczywiście osiągnięcie tych korzyści wymaga poczynienia pewnych inwestycji. Chodzi tutaj przede wszystkim o czas poświęcony na odpowiednie zaplanowanie architektury produktu oraz przestrzeganie tegoż planu w procesie implementacji oprogramowania. To w oczywisty sposób może być wystarczającym pretekstem do porzucenia

2022

idei przestrzegania koncepcji architektonicznych jednak za przestrogę niech posłuży tu stwierdzenie: „*jeżeli uważasz, że dobra architektura jest droga, to wypróbuj złą architekturę*”[1].

Po zapoznaniu się z częścią pisemną gorąco zachęcam do zobaczenia przygotowanej przeze mnie prezentacji, w której omawiam poszczególne elementy składowe projektu. Prezentacja jest dostępna pod adresem: https://drive.google.com/file/d/1557haF7oVqpQV2e-6iM_RcelTwM_lKSF/view?usp=sharing

[1] Robert C. Martin, Czysta architektura, Helion, Gliwice 2018, s. 15.

2 Charakterystyka użytych technologii

Główną technologią użytą w procesie implementacji projektu jest framework .Net 5. Jest to jedna z najnowszych wersji frameworka, będąca wynikiem unifikacji wersji .Net Framework oraz .Net Core. Dzięki temu zabiegowi możliwe stało się tworzenie aplikacji wieloplatformowych z wykorzystaniem jednej bazy kodu. Tym samym projekt może być wdrożony zarówno na platforma z systemem Windows jak i Linux.

Kolejną użytą technologią jest Blazor. Jest to framework używany do zastosowań frontendowych. Innymi słowy za jego pomocą możemy tworzyć strony internetowe. Cechą wyróżniającą ten framework jest w zasadzie całkowita eliminacja konieczności używania języka Javascript. Jest to możliwe dzięki standardowi WebAssembly. Standard ten pozwala na wykonywanie kodu natywnego w tzw. wyizolowanym środowisku przeglądarki internetowej. Dzięki temu programiści aplikacji backendowych mogą bez większych problemów zająć się tworzeniem rozwiązań frontendowych bez konieczności przyswajania języka JavaScript.

W przypadku warstwy persystencji zastosowałem sprawdzone rozwiązanie jakim jest silnik bazodanowy MS Sql Server a raczej jego zubożona odmiana – LocalDB. W przypadku tego projektu jest ona całkowicie wystarczalna. Poza tym należy zauważyć, że w przypadku podejścia DDD wybór bazy danych ma drugorzędne znaczenie[2]. Baza danych zostaje tu zdegradowana do roli „worka na dane" i użycie w niej jakiejkolwiek logiki jest niedopuszczalne.

W celu minimalizacji problemów związanych z wdrożeniem rozwiązania zastosowałem Docker. Dzięki temu narzędziu możliwe jest uruchamianie aplikacji bez konieczności instalowania na serwerze jakichkolwiek frameworków, środowisk wykonawczych, itd. Tym samym unikamy problemów związanych z niekompatybilnością bibliotek, wersji systemu czy też braku zmiennych środowiskowych. Kontener wykorzystuje jądro systemu na którym jest zainstalowany jednak dysponuje wyizolowanymi od reszty systemu zasobami takimi jak pamięć, przestrzeń dyskowa, zmienne środowiskowe, itd…

[2] Sobótka, S. (2012). Domain Driven Design krok po kroku. https://bottega.com.pl/pdf/materialy/ddd/ddd1.pdf. (data odczytu 10 grudnia 2021)

2022

Aplikacja umożliwia samodzielną rejestrację użytkownika wykorzystując w tym celu mailowe potwierdzenie procesu rejestracji. W celu minimalizacji nakładów związanych z infrastrukturą potrzebną do wysyłania maili użyłem bibliotekę firmy SendGrid. Rozwiązanie to umożliwia niezawodne wysyłanie dużej ilości wiadomości email bez konieczności dysponowania własnym serwerem pocztowym. W tym celu biblioteka łączy się z API dostawcy. Dodatkowo na portalu firmy można tworzyć szablony wiadomości w języku html. W szablonie możemy zdefiniować pewne zmienne, które przekazujemy za pomocą obiektów JSON.

W rozwiązaniu backendowym, czyli w serwisie WebApi użyłem bibliotekę MediatR. Biblioteka ta w znaczący sposób wspomaga programowanie przy użyciu wzorca Command and Query Responsibility Segregation. Wzorzec ten zostanie opisany w dalszej części pracy natomiast w dużym uproszczeniu jego celem jest rozdzielenie kodu odpowiedzialnego za odczyt danych od kodu odpowiedzialnego za ich zapis.

W celu zwiększenia wydajności aplikacji zastosowałem mechanizm cachowania. W tym przypadku użyłem rozwiązania o nazwie Redis. Jest to open-sourceowy magazyn danych działający na zasadzie słownika klucz-wartość. Przy czym godnym odnotowania jest fakt, że Redis obsługuje takie struktury danych jak: bitmapy, zestawy, listy, hashe a także indeksy geoprzestrzenne. Redis może działać również samodzielnie jako nierelacyjna baza danych.

Aby zapewnić większą wygodę i efektywność wyszukiwania ewentualnych błędów aplikacji bądź innych niepożądanych zdarzeń użyłem biblioteki Serilog. Biblioteka ta rozbudowuje standardową implementację wbudowanego w framework .Net interfejsu ILogger. Przewaga Seriloga nad standardowym rozwiązaniem jest taka, że umożliwia on logowanie strukturalne. Innymi słowy do loga możemy zapisać cały obiekt wraz z jego metadanymi, właściwościami.

Do łatwiejszego przeglądania logów użyłem narzędzia Seq. Seq w połączeniu z Serilog umożliwia wyświetlanie logów w postaci dashboardów jako stronę internetową. Tutaj ujawnia się kolejna zaleta Dockera. Tak naprawdę nie musimy zapewnić żadnego serwera WWW. Całość działa w wyizolowanym środowisku z własną podsiecią. Z naszej strony musimy zapewnić tylko wolny port na hoście Dockera aby móc przeglądać wspomniane dashboardy.

Rozwiązanie zostało również wyposażone w możliwość obsługi komunikacji asynchronej. W tym celu użyłem popularnego brokera wiadomości jakim jest RabbitMQ. Idea działania z grubsza polega na tym, że do wcześniej zdefiniowanej kolejki możemy publikować interesujące

2022

z naszego punktu widzenia wiadomości, zdarzenia jakie zaszły w naszym systemie. Po ich opublikowaniu nie interesuje nas dalszy los tychże wiadomości. Rolą systemów zainteresowanych jest zasubskrybowanie się na odbiór wiadomości z danej kolejki i obsłużenie ich we właściwy dla siebie sposób. Zakładając, że opisywany projekt jest tylko jednym z wielu systemów działających w firmie można łatwo wyobrazić sobie scenariusze wykorzystania takiego brokera wiadomości. W tym konkretnym przypadku publikowany jest fakt zarejestrowania nowej przesyłki przez klienta.

Ostatnim z rozwiązań wykraczającym poza te dostarczane przez framework są biblioteki xUnit oraz FluentAssertions. Są to biblioteki wspomagające przeprowadzanie testów jednostkowych. Osobiście uważam, że xUnit umożliwia przeprowadzanie testów w sposób bardziej zwięzły i czytelny aniżeli standardowa MS Tests. Fluent assertions dodatkowo wzmacnia to przekonanie. Jego siła tkwi w zastosowaniu wzorca Fluent Assertions co jeszcze bardziej zwiększa przejrzystość kodu.

3 Zakres funkcjonalny projektu

3.1 Wymagania funkcjonalne oraz niefunkcjonalne projektu

W celu przedstawienia wymagań funkcjonalnych projektu postanowiłem posłużyć się jedną z pochodnych przypadków użycia, czyli historyjkami użytkownika. Uważam, że w przeciwieństwie do „suchych" punktów opisowych, user stories pomagają w zrozumieniu zasadności wbudowania danej funkcjonalności w projekt. Przyjąłem tutaj standardowy szablon historyjki użytkownika:

- **Jako** – opisuje osobę bądź przypisaną jej rolę,
- **Chcę** – opisuje poszczególną cechę czy też funkcjonalność ważną z punktu widzenia użytkownika,
- **Ponieważ –** opisuje zasadność, korzyść wynikającą z dodania funkcjonalności.

Oto lista wymagań funkcjonalnych:

a. Jako potencjalny użytkownik chcę mieć możliwość zarejestrowania się w portalu ponieważ chcę zacząć korzystać z usług danej firmy kurierskiej,
b. Jako użytkownik chcę mieć możliwość zalogowania się do portalu ponieważ chcę aby wprowadzane przez mnie dany były chronione,
c. Jako użytkownik portalu chcę mieć dostęp do listy dotychczas utworzonych przeze mnie przesyłek wraz z aktualnym statusem przesyłki ponieważ chcę na bieżąco obserwować status realizacji zlecenia,
d. Jako użytkownik portalu chcę mieć możliwość definiowania własnej listy kontrahentów ponieważ nie chcę każdorazowo wprowadzać adresu stałego klienta od nowa,
e. Jako użytkownik portalu chcę mieć możliwość wydrukowania listu przewozowego ponieważ chcę aby kurier odbierający ode mnie paczkę, potwierdził na nim fakt powierzenia mu towaru,
f. Jako administrator portalu chcę mieć możliwość edycji słownika opakowań ponieważ rodzaje opakowań mogą ulec zmianie,
g. Jako administrator chcę mieć możliwość edytowania listy usług dodatkowych ponieważ jako firma staramy się poszerzać zakres świadczonych usług,

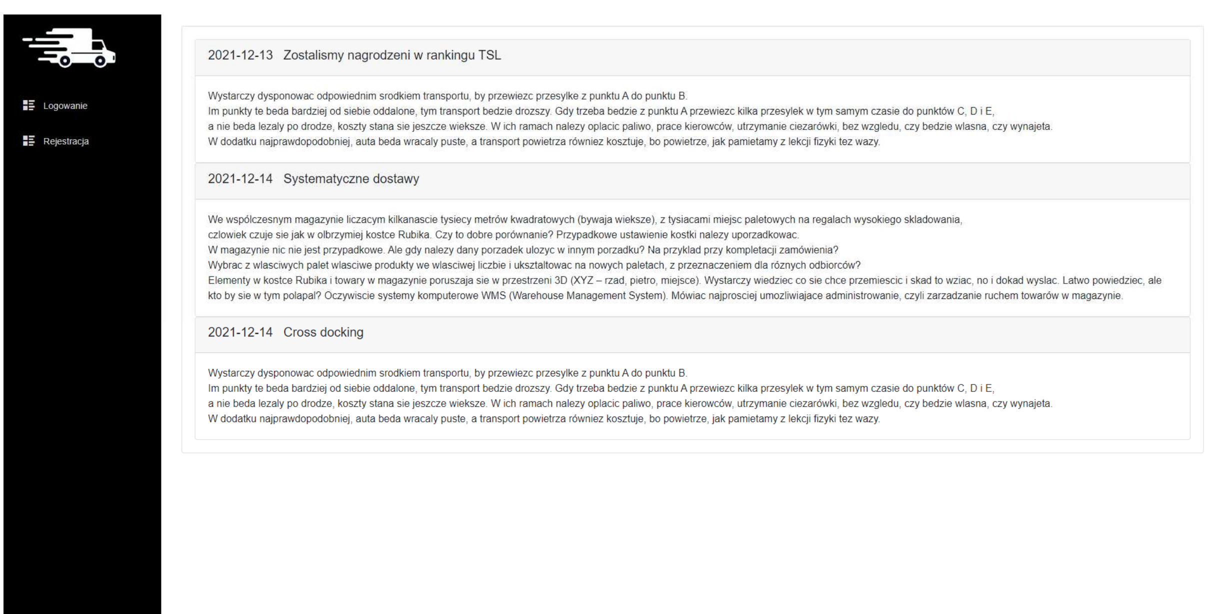
h. Jako administrator chcę mieć możliwość dodawania nowych wiadomości na stronę główną portalu ponieważ w ten sposób ułatwiamy przekazywanie nowych informacji klientom portalu

Wymaganie niefunkcjonalne projektu:

a. Aplikacja powinna być dostępna w systemie 24/7/365,
b. Aplikacja powinna móc obsłużyć nieograniczoną ilość użytkowników końcowych,
c. Maksymalny czas odpowiedzi na zapytanie użytkownika to 2000 ms,
d. Aplikacja jest przystosowana do działania w środowisku rozproszonym,
e. Aplikacja może działać w systemie Windows jaki Linux – jedynym wymogiem jest konieczność zapewnienia środowiska Docker.

3.2 Prezentacja działania rozwiązania

Po przejściu pod adres aplikacji otrzymamy stronę z zaledwie dwoma opcjami (logowanie, rejestracja) oraz listą aktualności:

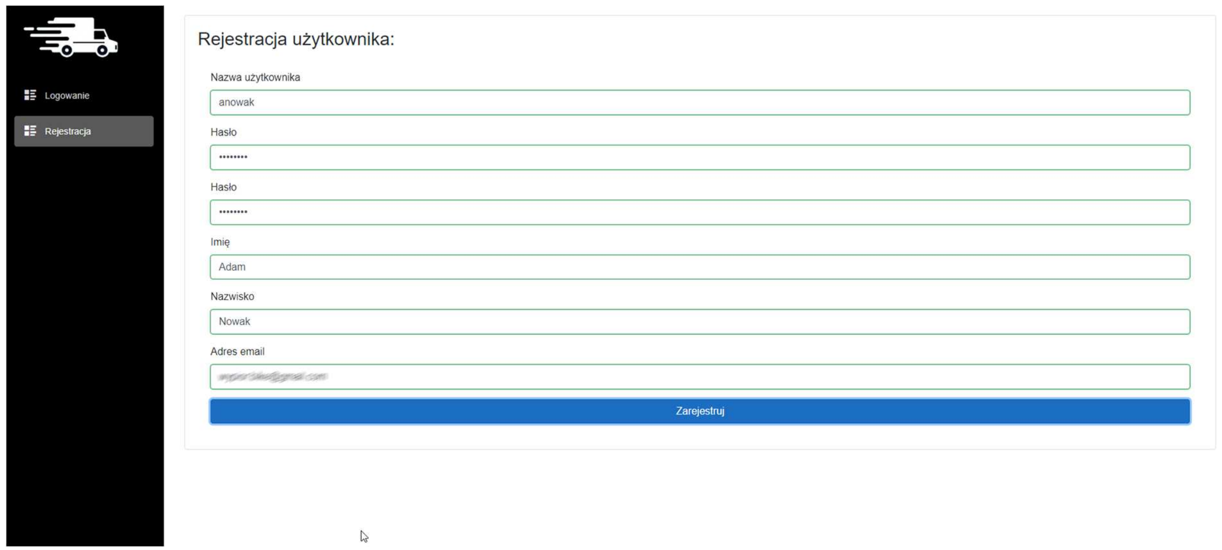


Rysunek 1. Ekran startowy aplikacji

Jeżeli wchodzimy do aplikacji po raz kolejny bardzo prawdopodobne jest, że pominiemy proces logowania się. Dzieje się tak dlatego, że w procesie autentykacji oraz autoryzacji wykorzystywany jest mechanizm JSon Web Token. Mechanizm ten zostanie

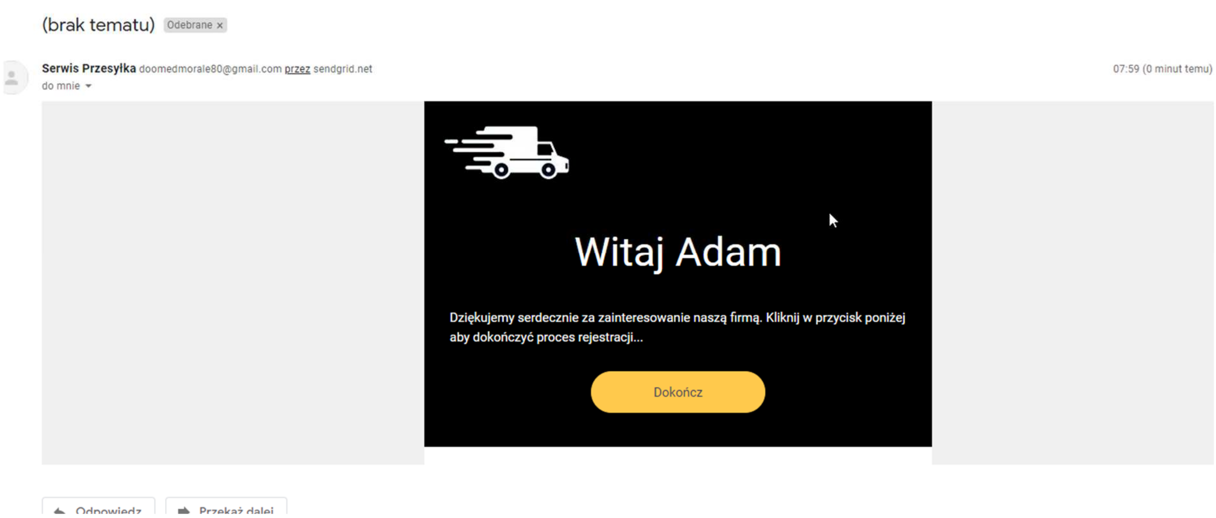
opisany w dalszej części pracy. W telegraficznym skrócie zasada działania polega na tym, że po poprawnym zalogowaniu się do WebApi zostaje zwrócony zserializowany obiekt zwany tokenem. Token ten jest zapisany po stronie przeglądarki i dalsza autoryzacja dostępów odbywa się z poprzez przekazywanie tegoż właśnie tokena w nagłówku http. Dopóki ważność tokena nie upłynie będziemy mogli swobodnie działać w ramach aplikacji webowej bez konieczności ponownego logowania się.

Proces rejestracji polega na uzupełnieniu nazwy użytkownika, dwukrotnego podania hasła (w celu uniknięcia pomyłki), imienia i nazwiska oraz adresu email.



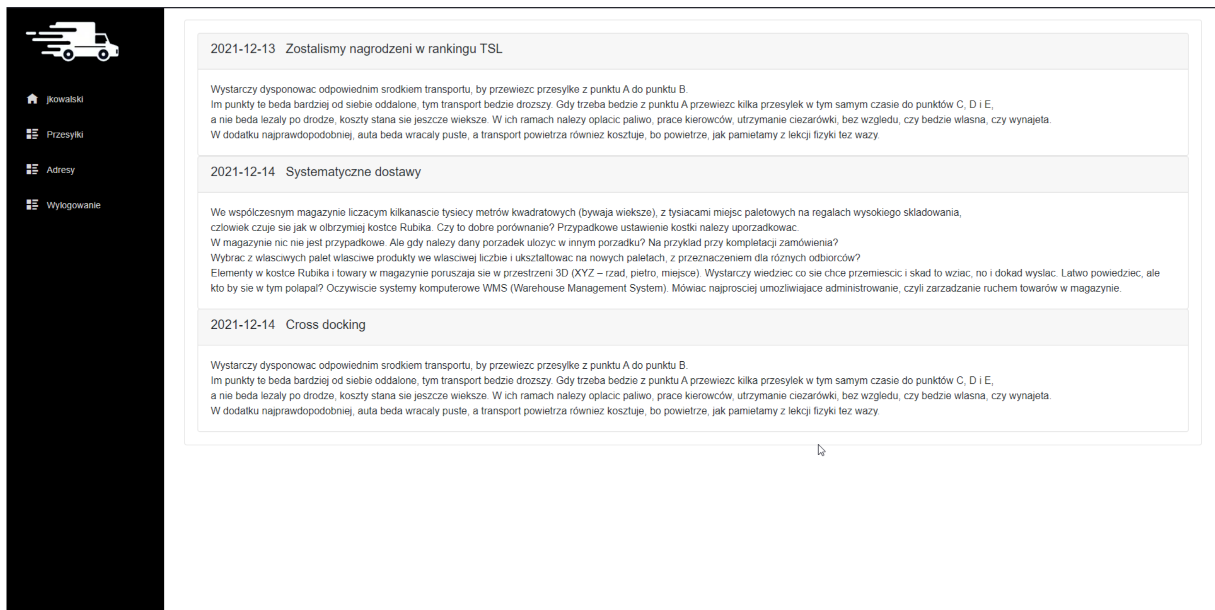
Rysunek 2. Rejestracja użytkownika

Jeżeli utworzenie użytkownika przebiegło pomyślnie na podany podczas rejestracji adres mailowy zostanie wysłana wiadomość potwierdzająca rejestrację. Potwierdzenie rejestracji polega na aktywowaniu konta użytkownika i umożliwia jego zalogowanie się do portalu.

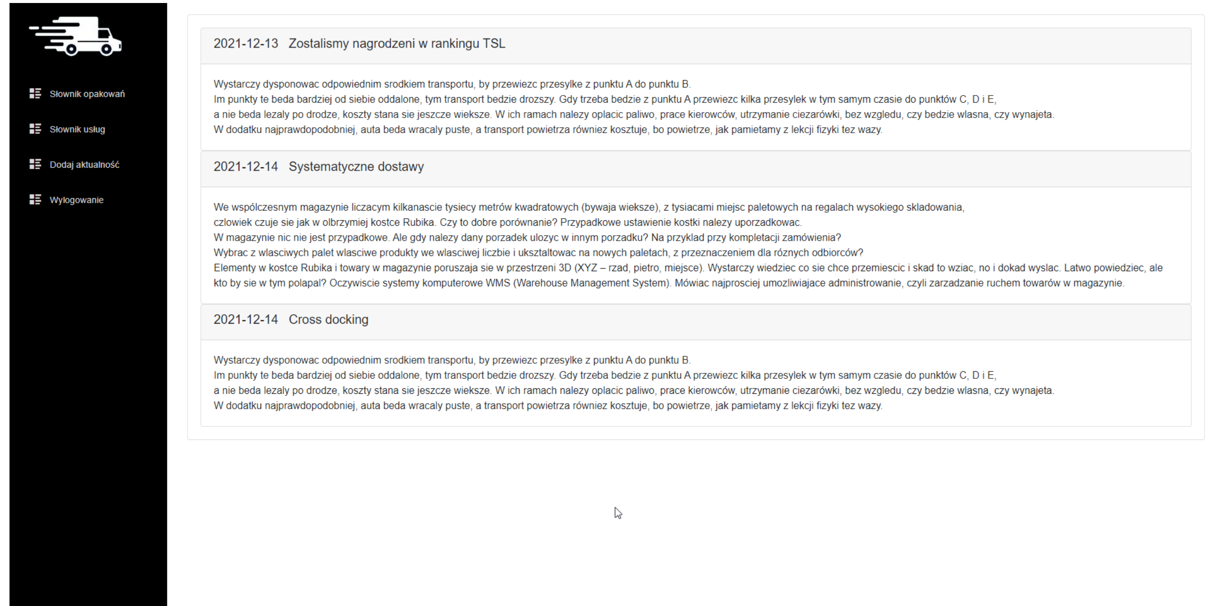


Rysunek 3. Wiadomość potwierdzająca rejestrację

Po zalogowaniu się użytkownik w zależności od przydzielonej roli ma dostępne określone pozycje menu:

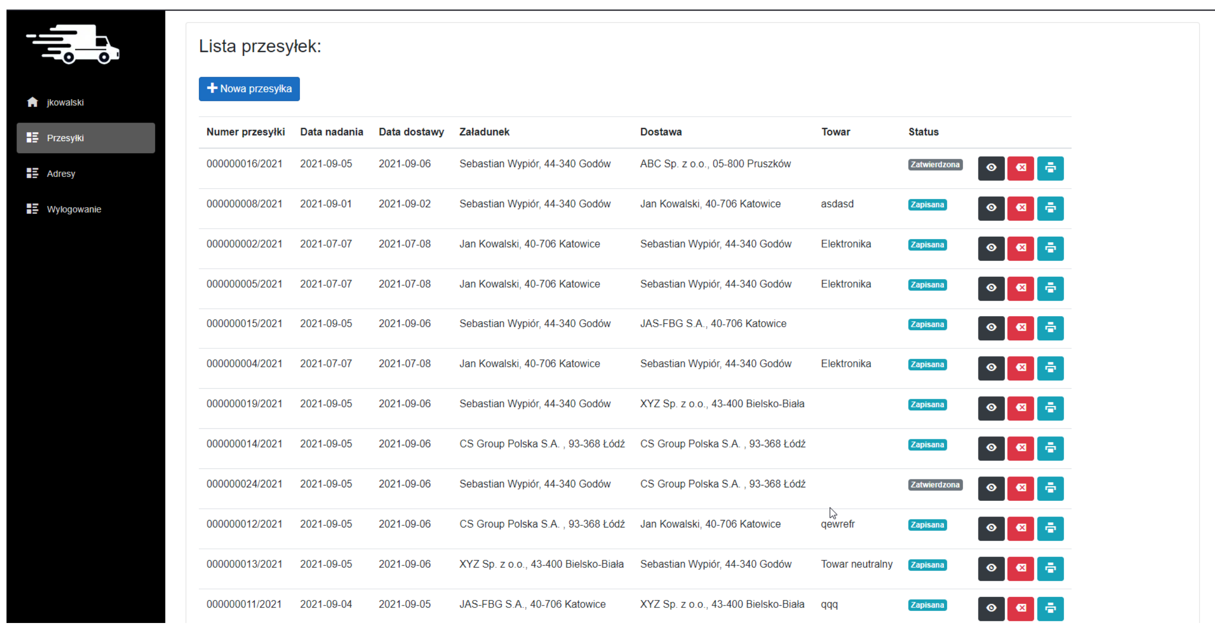


Rysunek 4. Menu użytkownika



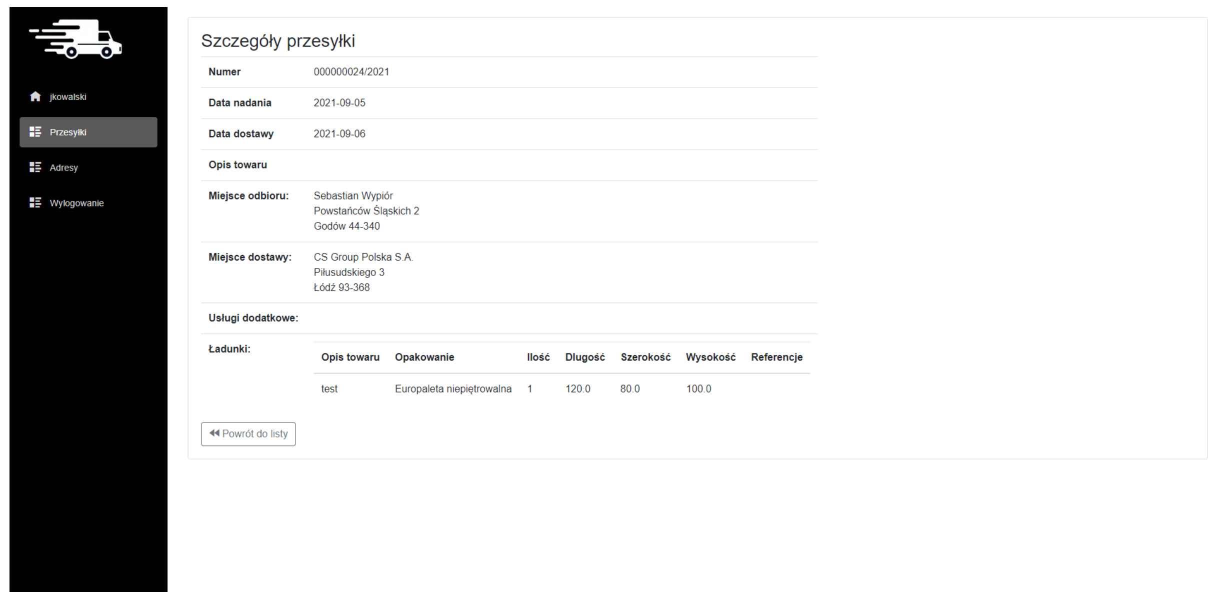
Rysunek 5. Menu administratora

Kliknięcie w pozycję „Przesyłki" przenosi użytkownika do listy utworzonych przez niego dotychczas przesyłek.



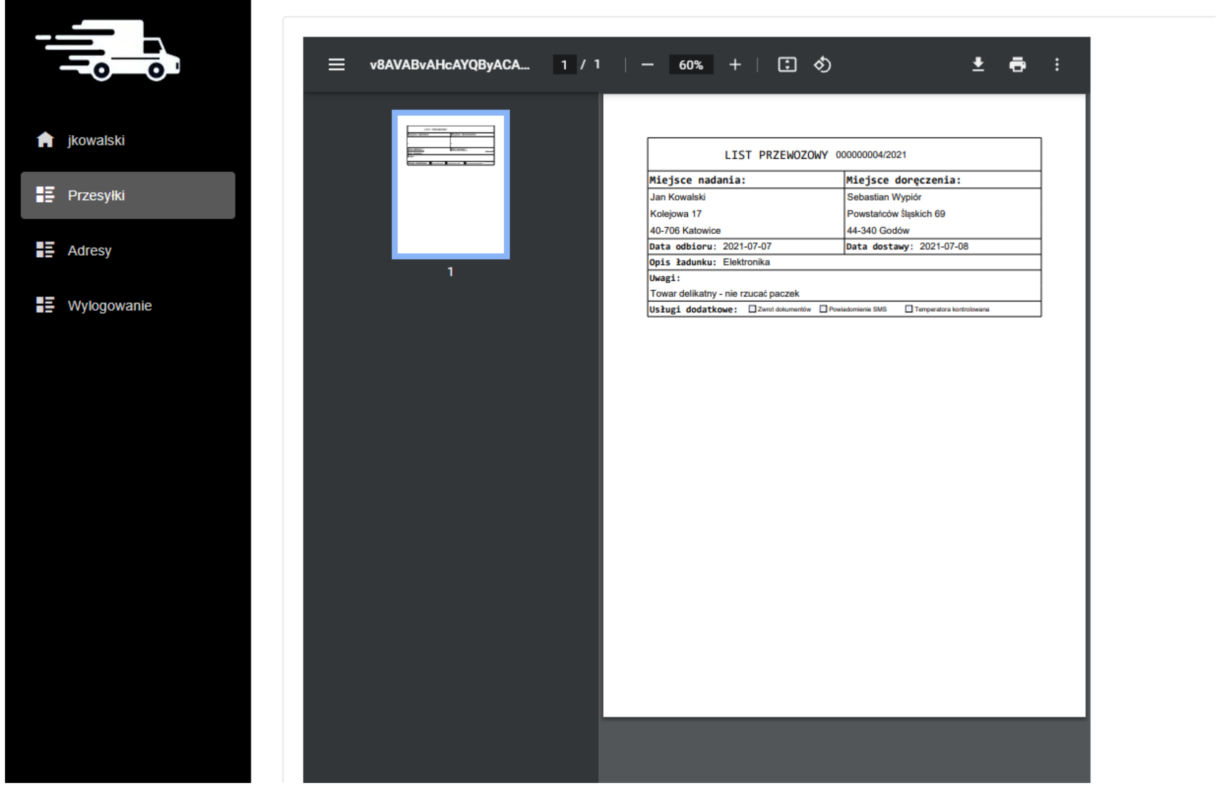
Numer przesyłki	Data nadania	Data dostawy	Załadunek	Dostawa	Towar	Status
000000016/2021	2021-09-05	2021-09-06	Sebastian Wypiór, 44-340 Godów	ABC Sp. z o.o., 05-800 Pruszków		Zatwierdzona
000000008/2021	2021-09-01	2021-09-02	Sebastian Wypiór, 44-340 Godów	Jan Kowalski, 40-706 Katowice	asdasd	Zapisana
000000002/2021	2021-07-07	2021-07-08	Jan Kowalski, 40-706 Katowice	Sebastian Wypiór, 44-340 Godów	Elektronika	Zapisana
000000005/2021	2021-07-07	2021-07-08	Jan Kowalski, 40-706 Katowice	Sebastian Wypiór, 44-340 Godów	Elektronika	Zapisana
000000015/2021	2021-09-05	2021-09-06	Sebastian Wypiór, 44-340 Godów	JAS-FBG S.A., 40-706 Katowice		Zapisana
000000004/2021	2021-07-07	2021-07-08	Jan Kowalski, 40-706 Katowice	Sebastian Wypiór, 44-340 Godów	Elektronika	Zapisana
000000019/2021	2021-09-05	2021-09-06	Sebastian Wypiór, 44-340 Godów	XYZ Sp. z o.o., 43-400 Bielsko-Biała		Zapisana
000000014/2021	2021-09-05	2021-09-06	CS Group Polska S.A. , 93-368 Łódź	CS Group Polska S.A. , 93-368 Łódź		Zapisana
000000024/2021	2021-09-05	2021-09-06	Sebastian Wypiór, 44-340 Godów	CS Group Polska S.A. , 93-368 Łódź		Zatwierdzona
000000012/2021	2021-09-05	2021-09-06	CS Group Polska S.A. , 93-368 Łódź	Jan Kowalski, 40-706 Katowice	qewrefr	Zapisana
000000013/2021	2021-09-05	2021-09-06	XYZ Sp. z o.o., 43-400 Bielsko-Biała	Sebastian Wypiór, 44-340 Godów	Towar neutralny	Zapisana
000000011/2021	2021-09-04	2021-09-05	JAS-FBG S.A., 40-706 Katowice	XYZ Sp. z o.o., 43-400 Bielsko-Biała	qqq	Zapisana

Z poziomu listy przesyłek użytkownik ma możliwość sprawdzenia aktualnego statusu przesyłki, podglądu szczegółów, anulowania przesyłki oraz wydrukowania listu przewozowego.



Rysunek 6. Widok szczegółów przesyłki

Kliknięcie w symbol drukarki przy danej pozycji spowoduje wygenerowanie wydruku listu przewozowego. List przewozowy można pobrać w formacie pliku PDF bądź też wydrukować:

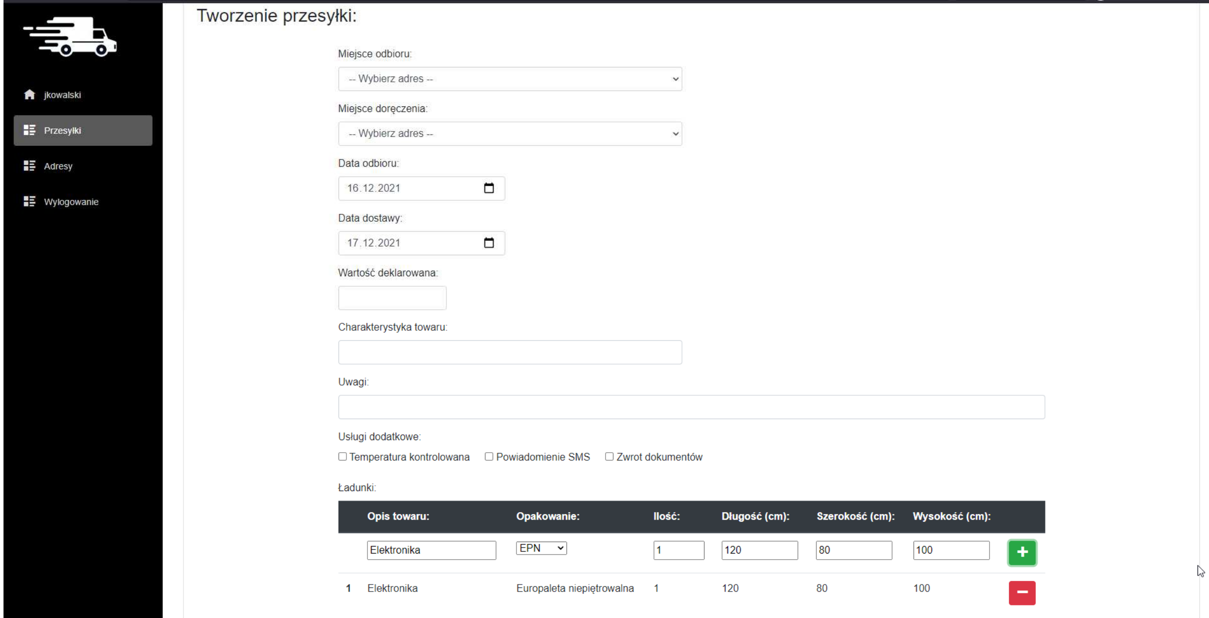


Rysunek 7. Wydruk listu przewozowego

2022

Zasadniczym elementem portalu ze strony użytkownika jest funkcjonalność tworzenia nowej przesyłki. Tworząc nową przesyłkę uzupełniamy takie elementy jak: miejsce odbioru przesyłki, miejsce doręczenia, daty odbioru i doręczenia, opis przewożonego towaru, ewentualną wartość deklarowaną a także uwagi.

Ponadto użytkownik ma możliwość zaznaczenia usług dodatkowych towarzyszących dostawie przesyłki. Chodzi tutaj o transport w temperaturze kontrolowanej (przesyłka wymaga chłodni), potwierdzenia zamiaru dostarczenia przesyłki poprzez wiadomość SMS oraz o zwrot potwierdzonych dokumentów.



Rysunek 8. Tworzenie nowej przesyłki

Każda przesyłka wymaga również zarejestrowania przynajmniej jednego ładunku. Chodzi o charakterystykę towaru, podanie takich szczegółów jak opis towaru, rodzaj opakowania, jego wymiary oraz ilość. Są to dane krytyczne dla procesu planowania i dostawy przesyłek. Oczywiście do momentu zatwierdzenia przesyłki lista może być dowolnie edytowana, czyli można dodawać bądź usuwać pozycje ładunków:

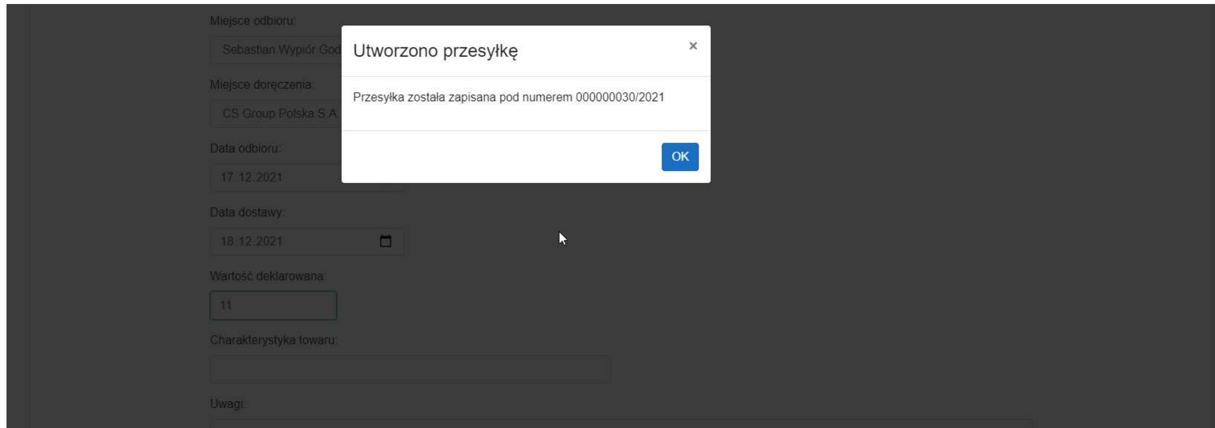
Ładunki:

	Opis towaru:	Opakowanie:	Ilość:	Długość (cm):	Szerokość (cm):	Wysokość (cm):	
	Elektronika	EPN	1	120	80	100	+
1	Elektronika	Europaleta niepiętrowalna	1	120	80	100	−

Rysunek 9. Edycja ładunków przesyłki

2022

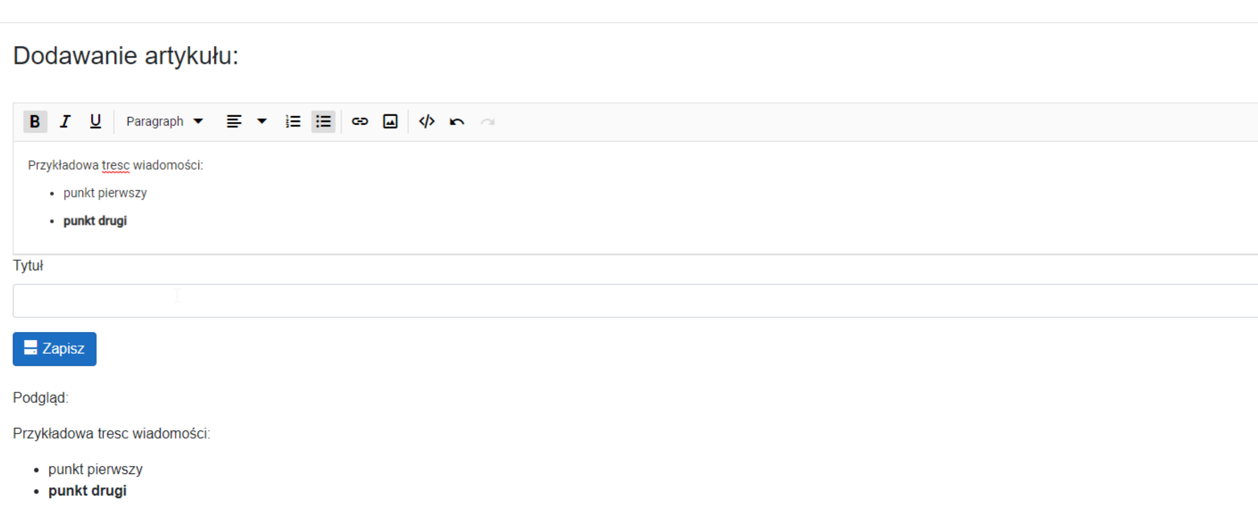
Przesyłka może zostać zapisana a w dalszym etapie zatwierdzona lub od razu zapisana i zatwierdzona. Po jej zapisie pojawia się komunikat:



Rysunek 10. Komunikat o utworzeniu przesyłki

Różnica pomiędzy zapisem a zatwierdzeniem przesyłki polega na tym, że zapisaną przesyłkę niezatwierdzoną można jeszcze anulować. Natomiast przesyłka zatwierdzona nie może być anulowana, ponieważ zostało wygenerowane odpowiednie zdarzenie i fakt ten został opublikowany w brokerze wiadomości. W związku z czym pozostałe systemie w firmie, które zasubskrybowały zdarzenia z kolejki mogą już przetwarzać dane związane z powstaniem nowej przesyłki.

Strona administracyjna programu pozwala na zarządzenie słownikiem opakowań a także słownikiem usług dodatkowych. Ponadto do systemu został wbudowany moduł umożliwiający dodawanie treści komunikatów na stronie głównej portalu. Jak wspomniałem wcześniej, zadaniem tego modułu jest ułatwienie przekazywania istotnych z punktu przedsiębiorstwa informacji klientowi. Edytor umożliwia podstawowe formatowanie tekstu oraz podgląd tworzone wiadomości w trybie rzeczywistym: bezpośrednio pod edytorem redagowanej wiadomości znajduje się obszar podglądu wiadomości. Dzięki temu mamy możliwość podejrzenia „na żywo" wyników naszej pracy:



Rysunek 11. Redagowanie nowej wiadomości

4 Implementacja

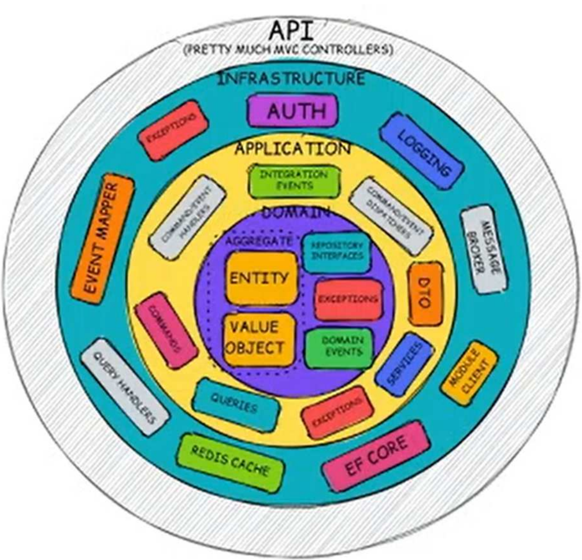
Z funkcjonalnego punktu widzenia całość rozwiązania można podzielić na dwie główne części: część backendową oraz frontendową. Przy czym należy zwrócić uwagę, że są to dwa całkowicie odrębne projekty – żaden z tych projektów nie ma jakichkolwiek referencji do drugiego. Występuje tu oczywiście powiązanie, ale jest to tak zwane luźne powiązanie.

Głównym zamysłem utworzenia takiego rozwiązania był fakt, że to warstwa backendowa, czyli projekt WebApi jest tym najważniejszym projektem. Projekt UI – strona internetowa – jest tylko klientem tego pierwszego. Biznesową przesłanką takiego rozwiązania jest fakt, że znaczna część klientów firm kurierskich dysponuje własnym oprogramowaniem do obsługi branży e-commerce i zazwyczaj warstwa prezentacji staje się zbędna. W takich sytuacjach systemy integrowane są ze sobą na poziomie komunikatów JSON wymienianych przy pomocy protokołu http.

4.1 Część backendowa – WebApi

4.1.1 Czysta architektura – ujęcie teoretyczne

Termin „czysta architektura" został przedstawiony przez Roberta C. Martina, znanego również jako Uncle Bob. Bazuje ona na podobnych założeniach jak architektura heksagonalna czy tzw. onion architecture. Głównym celem wszystkich tych podejść jest separacja zagadnień. Separacja ta jest możliwa do osiągnięcia poprzez podział oprogramowania na warstwy:



Rysunek 12 Podział warstw zgodnie z zasadami czystej architektury

Jak pokazuje powyższy rysunek, ideą czystej architektury jest podział oprogramowania na cztery warstwy:

- Domena – najważniejsza z biznesowego punktu widzenia. Zawieramy w niej wszystkie obiekty (encje, value objects), funkcje oraz reguły biznesowe,
- Przypadki użycia, czyli tak zwane use cases – mówiąc inaczej, warstwa ta pełni funkcję orkiestracji pomiędzy poleceniami wydawanymi przez użytkownika a obiektami biznesowymi,
- Warstwa adapterów interfejsów – w tej warstwie tworzymy implementacje interfejsów zdefiniowanych w warstwie wyżej,
- Warstwa frameworków i sterowników – jest to tak zwana warstwa infrastrukturalna i skupia się ona na obsłudze takich elementów jak baza danych, zewnętrzne serwisy, urządzenia, itd…

Najbardziej istotną cechą tejże architektury jest przestrzeganie reguły, że zależności są zawsze kierowane do wnętrza okręgu zaprezentowanego na powyższym rysunku. W efekcie elementy warstwy wewnętrznej nie mają żadnej wiedzy na temat sposobu działania warstw zewnętrznych. Jest to możliwe dzięki zastosowaniu interfejsów oraz wzorca dependency injection. Tym samym warstwa domenowa operuje na samym interfejsach, które jak wiemy nie zawierają żadnej implementacji. To daje nieograniczone wręcz możliwości modyfikacji i rozbudowy poszczególnych elementów oprogramowania. Przy czym zapewniamy maksymalny poziom ochrony najważniejszej z punktu widzenia biznesowego warstwy – warstwy domeny[3]. Dobrze zaprojektowana warstwa domeny powinna być modyfikowana jak najrzadziej gdyż to właśnie ona jest najbardziej podatna na wszelkie błędy generujące koszty.

4.2 Implementacja czystej architektury w projekcie

4.2.1 Diagram architektury / klas

Koncepcję czystej architektury oraz wplecionych w nią zasad Domain Driven Design przedstawia diagram przedstawiony na rysunku 14. Obrazuje on zależności pomiędzy podstawowymi obiektami aplikacji, czyli encjami. Zauważalna jest tutaj pewna schematyczność. Klasy wyposażone są w podobne interfejsy oraz metody fabrykujące swoje instancje. Konstruktory klas są prywatne, więc nie jesteśmy w stanie utworzyć instancji klas w tradycyjny sposób. Służą do tego odpowiednie metody. jeżeli To w nich zawarta jest Mocno

[3] Smith, S. (2021). Clean Architecture with ASP.NET Core. https://ardalis.com/clean-architecture-asp-net-core, (data odczytu 10 grudnia 2021)

zauważalna jest tutaj również hierarchia dziedziczenia. Skupia się ona wokół dwóch obiektów: entity oraz value object. Z diagramu w bardzo łatwy sposób można wywnioskować która obiekt pełni najistotniejszą rolę biznesową. Jest nim oczywiście klasa Shipment. To właśnie ta klasa zawiera w sobie większość pozostałych obiektów i umożliwia dostęp do nich jedynie poprzez swoją instancję. Oczywiście w procesie udział bierze szereg innych klas jednak stanowią one szczegóły implementacyjne i nie stanowią sedna projektu. Ujęcie ich na diagramie znacznie obniżyło by i tak już mocno nadwyrężoną jego czytelność.

W tym miejscu pragnę zauważyć, że diagram został przygotowany przy pomocy języka C4. Zaletą takiego podejścia jest to, że źródłem diagramu jest pseudokod, który z powodzeniem można przechowywać na przykład w repozytorium typu Git. Oto fragment kodu tworzącego diagram:

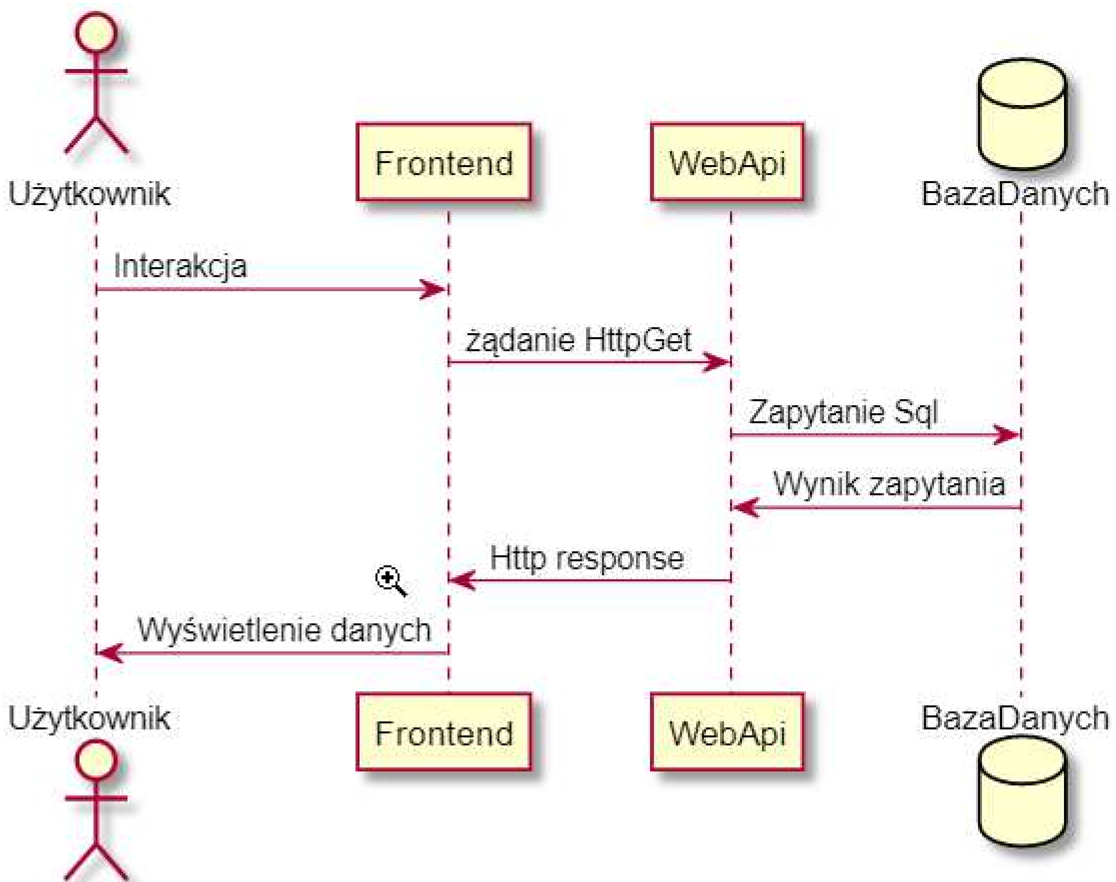
```
class ServiceBuilder {
    +Code : string
    +Name : string
    +WithCode(code : string)
    +WithName(name : string)
    +Build() : Service
}

class Article {
    +<<get, set>>Title : string
    +<<get, set>>Content : string
}

class AppUser {
    +<<get, set>> FirstName : string
    +<<get, set>> LasttName : string
}

Entity ..> AppUser
ZipCode o-- Address
EmailAddress o-- Address

Entity <|-- Cargo
Entity <|-- Address
Entity <|-- Service
Entity <|-- Article
```

Rysunek 13 Fragment kodu w standardzie C4

Rysunek 14 Diagram klas

4.2.2 Diagram kooperacji komponentów aplikacji

Dla zobrazowania przepływu danych pomiędzy poszczególnymi komponentami aplikacji postanowiłem stworzyć niewielki diagram sekwencji. Diagram ten ukazuje „z lotu ptaka” komunikację pomiędzy poszczególnymi warstwami aplikacji:



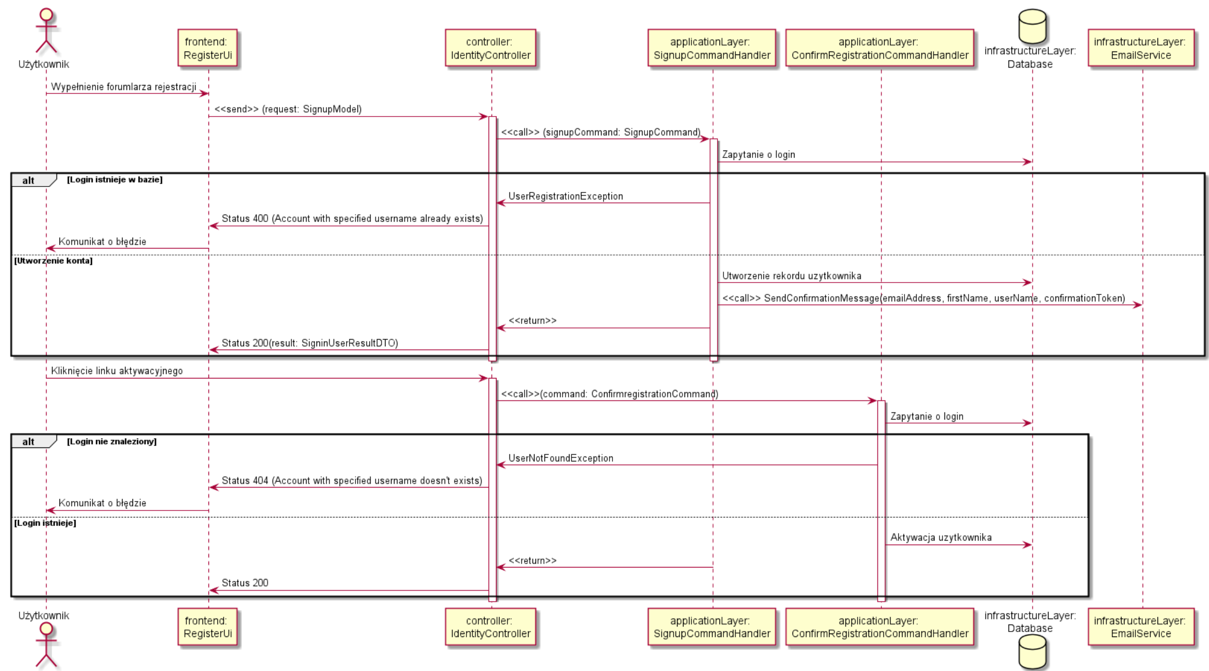
Rysunek 15 Diagram sekwencji komponentów aplikacji

4.2.3 Diagram sekwencji wybranego procesu

Poniższy diagram natomiast prezentuje drogę jaką przebywa pojedynczy komunikat od punktu startowego, mianem którego możemy określić użytkownika poprzez wszystkie komponenty aplikacji. Stosunkowo proste zadanie jakim jest proces rejestracji użytkownika pozornie wymaga użycia sporej ilości komponentów. To pozorne skomplikowanie w rzeczywistości upraszcza cały proces z punktu widzenia implementacji.

Chodzi o to, że każdy z komponentów ma tutaj jasno określoną odpowiedzialność. Przykładowo kontroler nie zawiera żadnej logiki biznesowej. Pełni on tylko rolę techniczną – przechwytuje komunikat z zewnątrz i przekazuje go w formie komendy do odpowiedniego handlera. Handler – jeżeli trzeba – wykorzystuje bazę danych bądź inny komponent warstwy infrastruktury (w tym przypadku EmailService).

2022



Rysunek 16 Diagram sekwencji procesu logowania

Stosując się do zasad czystej architektury solucję zbudowałem z następujących projektów:

4.2.4 ShipmentService.Api

Jest to najbardziej zewnętrzna warstwa, która stanowi interfejs pomiędzy logiką aplikacji a jej klientami. Głównym składnikiem projektu są kontrolery, które odbierają żądania klientów i przesyłają je do odpowiednich handlerów. Na uwagę zasługuje fakt, że kontrolery są pozbawione jakiejkolwiek logiki. Dzieje się tak głównie z dwóch powodów: pierwszym jest zamysł tworzenia kodu jak najbardziej umożliwiającego jego testowanie, co w przypadku kontrolerów nie jest najlepszym pomysłem. Drugi powód to kwestia ułatwienia „wymienialności" interfejsu, za pomocą odbywa się komunikacja z klientem.

```
[HttpGet("{id:guid}")]
[Authorize(Roles = "Customer,Administrator")]
[ProducesResponseType(typeof(ActionResult<ArticleDTO>), Status200OK)]
[ProducesResponseType(typeof(ApiProblemDetailsResponse), Status404NotFound)]
1 reference
public async Task<ActionResult<IEnumerable<AddressDTO>>> GetArticleById(Guid id)
{
    var result = await _mediator.Send(new GetArticleByIdQuery(id));

    if (result is null)
        return NotFound();

    return Ok(result);
}
```

Rysunek 17. Standardowa struktura akcji kontrolera

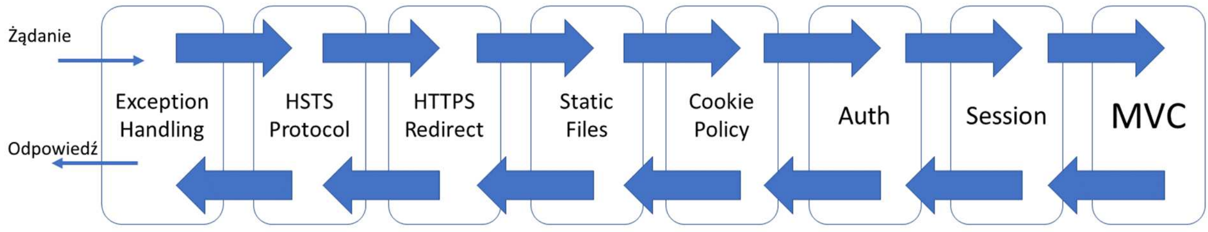
W chwili obecnej są to kontrolery z zachowaniem stylu RESTful, ale w przyszłości może zapaść decyzja o zmianie komunikacji na GraphQL. Zawierając logikę w kontrolerach musielibyśmy przepisać ją od nowa. W tym podejściu wystarczy przepisać tylko najbardziej zewnętrzną warstwę.

Kontrolery oraz ich poszczególne akcje chronione są polisami. Jak wspomniałem w początkowej części pracy, autoryzacja odbywa się poprzez wymianę obiektu JWT (Json Web Token), który uzyskujemy po pomyślnym wywołaniu metody autentykującej. Jednym z elementów tokena są tak zwane *claims*, które zawierają określone porcje informacji. W tym przypadku zawarłem w nich nazwy ról, do których został przypisany zalogowany użytkownik. Dostępne są dwie role: *administrator* oraz *customer*. Poniższy fragment pokazuje dwie metody kontrolera *PackageController* gdzie do jednej z metod ma dostęp zarówno administrator jak i klient a do metody odpowiadającej za tworzenie nowych opakowań – tylko administrator:

```
[HttpPost]
[Authorize(Roles = "Administrator")]
[ProducesResponseType(typeof(ActionResult<PackageTypeDTO>), Status201Created)]
[ProducesResponseType(typeof(ApiProblemDetailsResponse), Status422UnprocessableEntity)]
0 references
public async Task<ActionResult<PackageTypeDTO>> CreatePackageType([FromBody] CreatePackageTypeCommand createPackageTypeCommand)
{
    var result = await _mediator.Send(createPackageTypeCommand);
    return CreatedAtAction(nameof(GetPackageTypeById), new { id = result.Id }, result);
}

[HttpGet("{id:guid}")]
[Authorize(Roles = "Customer,Administrator")]
[ProducesResponseType(typeof(ActionResult<PackageTypeDTO>), Status200OK)]
[ProducesResponseType(typeof(ApiProblemDetailsResponse), Status404NotFound)]
1 reference
public async Task<ActionResult<PackageTypeDTO>> GetPackageTypeById(Guid id)
{
    var result = await _mediator.Send(new GetPackageTypeByIdQuery(id));

    if (result is null)
        return NotFound();

    return Ok(result);
}
```

Rysunek 18. Autoryzacja dostępu do metod kontrolera przy użyciu polis

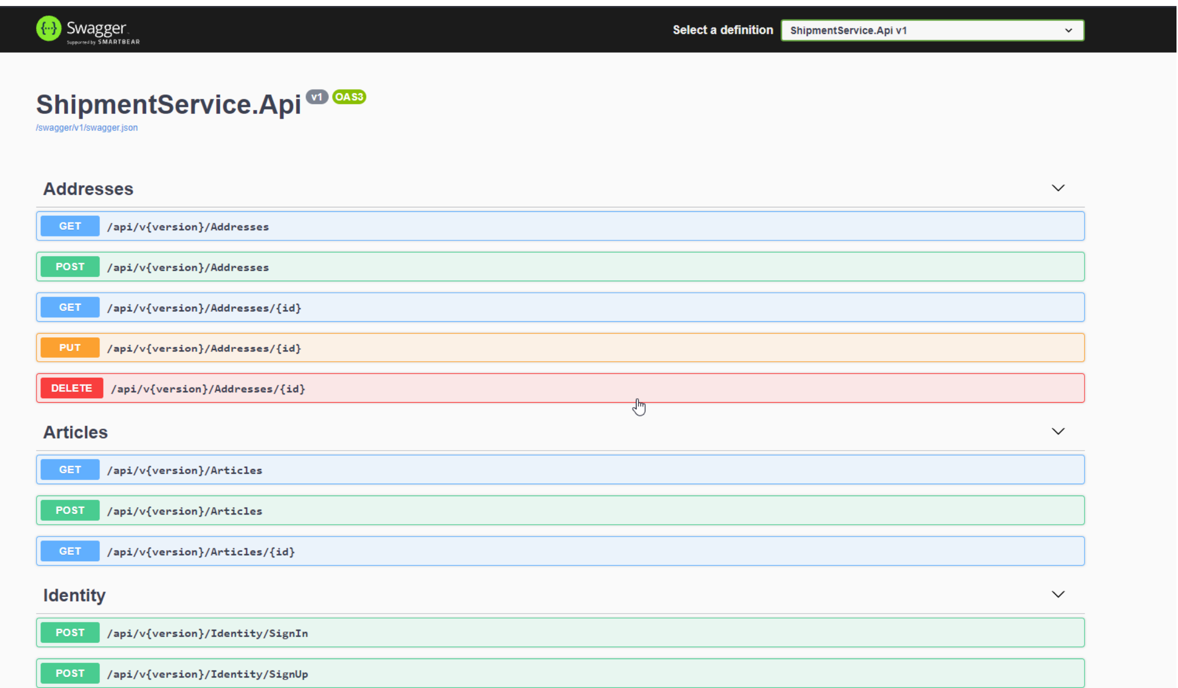
Oprócz kontrolerów projekt ten zawiera również logikę obsługi błędów aplikacji. Logika błędów aplikacji zostanie opisana szerzej w dalszej części pracy gdyż znajduje się w innych projektach natomiast w tym miejscu odbywa się tłumaczenie błędów biznesowych na komunikaty zrozumiałe dla klientów serwisu – odpowiednie kody HTTP. Aby uniknąć powtarzalności kodu zdecydowałem się na napisanie w tym celu własnego middleware. W frameworku Net 5 każde żądanie HTTP przechodzi przez tzw. pipeline (potok), w którym żądanie to jest przetwarzane przez odpowiednie metody.



Rysunek 19. Obsługa żądania http

Jeżeli żądanie zostanie z powodzeniem przetworzone przez wszystkie elementy potoku, wraca w odwrotnej kolejności do początku potoku gdzie wysyłana jest odpowiedź do klienta. Logika middleware, które napisałem polega na tym, że sprawdzam czy na danym etapie wystąpił wyjątek w aplikacji. Jeżeli tak to mapuję wyrzucony wyjątek na odpowiadający mu kod http.

Projekt został również wyposażony w narzędzie Swagger. Służy ono do zautomatyzowanego dokumentowania Api. Dzięki tej bibliotece klient ma możliwość łatwego zapoznania się z budową poszczególnych endpointów serwisu, jego parametrami wejściowymi, ze strukturą komunikatów odpowiedzi a nawet ma możliwość przetestowania poszczególnych endpointów. Dokumentacja jest dostępna jako podstrona projektu domyślnie pod adresem *adresprojektu/swagger*.



Rysunek 20. Widok strony dokumentacyjnej api wygenerowanej przez narzędzie Swagger

4.2.5 ShipmentService.Application

Ta warstwa pełni rolę orkiestracji pomiędzy kontrolerami a warstwą domeny, czyli logiką biznesową. Oczywistym jest, że logika biznesowa wymaga operowania na danych. Skoro warstwa domeny, w której powinna znajdować się logika biznesowa, nie może mieć referencji do innych projektów to warstwa przypadków użycia, czyli projekt *ShipmentService.Application* odpowiada za dostarczanie do niej danych.

Aby w znaczący sposób zwiększyć czytelność tejże warstwy, zdecydowałem się na użycie wzorca *Command Query Responsibility Segregation*. Wzorzec *CQRS* jest rozwinięciem innego wzorca – *Command Query Separation* – opublikowanego w 1988 roku przez Bertranda Meyera[4]. Chodzi tutaj o rozdzielenie operacji modyfikującymi stan obiektów (*commands*) oraz operacji będącymi zapytaniami (*queries*). To stąd wzięło się stwierdzenie, że zadanie pytania nie powinno ingerować w odpowiedź. Wzorzec *CQRS* rozszerza tę koncepcję kategoryzując komendy oraz zapytania na oddzielne byty, które zasilane są dedykowanymi tylko na ich potrzeby modelami. Stosowanie wzorca w naturalny sposób wspiera przestrzegania jednej z zasad *SOLID* a mianowicie *Single Responsibility Prcinciple* – zasady pojedynczej odpowiedzialności, w której chodzi o to, że każda klasa bądź też metoda powinna mieć dokładnie tylko jeden powód do zmiany[5].

Implementacja wzorca *CQRS* w projekcie postanowiłem posegregować wszystkie operacje wg klucza jakim jest nazwa encji, na której wykonywane są operacje. Wewnątrz folderu występują trzy kolejne: *Commands, Queries* oraz *DTOs.* W pierwszym z nich znajdują się wszystkie metody zmieniające stan danej encji, w drugim wszystkie metody dokonujące odczytu i zwracające dane, w których podstawowym obiektem jest dana encja a w trzecim obiekty DTO (*Data Transfer Objects*), które stanowią swego rodzaju nośnik dla parametrów metod odczytujących lub modyfikujących bądź są obiektami zwracanymi jako wyniki zapytań. Dla encji o nazwie *Cargo* struktura folderów projektu wygląda następująco:

[4] Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, Hoboken, 1988
[5] Robert C. Martin, Czysta architektura, Helion, Gliwice 2018, s. 81.

2022

```
Cargos
  Commands
    CreateCargo.cs
    DeleteCargo.cs
    UpdateCargo.cs
  DTOs
    CargoDTO.cs
    CreateCargoDTO.cs
    UpdateCargoDTO.cs
  Queries
    GetShipmentCargos.cs
```

Rysunek 21. Struktura folderów z zastosowaniem CQRS

Dodatkowo zdecydowałem się na użycie biblioteki wspomagającej programowanie zgodnie ze wzorcem *CQRS* – *MediatR*. Biblioteka ta pozwalana na wprowadzenie pewnej szablonowości w kodzie co przyczynia się do zwiększenia jego czytelności oraz ogólnego zrozumienia. Kolejnym powodem, dla którego zdecydowałem się na użycie tej biblioteki jest wspomaganie obsług zdarzeń biznesowych w programie. Ostatnią z zalet tej biblioteki jaką chciałem przedstawić jest wspomaganie tak zwanego luźnego wiązania[6]. Dzięki temu unikamy referencji jednej klasy do innej. Szczególnie istotne znaczenie ma to w przypadku testowania jednostkowego gdzie testujemy pojedynczą, małą jednostkę aplikacji w związku z czym pozostałe komponenty powinny być wyłączone z działania tak aby nie zakłócać wyników testu. Takie elementy są w testach zazwyczaj mockowane, a luźne wiązanie sprzyja temu procesowi. Zasad działania biblioteki *MediatR* jest względnie prosta: nie tworzymy bezpośrednio metody odpowiedzialnej za wykonanie danego zadania. Tworzymy za to obiekt będący komendą lub zapytaniem oraz klasę nazywaną handlerem, czyli klasą obsługującą nasze komendy oraz zapytania. Dla operacji tworzenia nowego rodzaju opakowania, klasy komendy oraz handlera przedstawiają się następująco:

[6] Alexey Zimarev, Domain-Driven Design dla .Net Core, Helion, Gliwice 2021, s. 371

```
3 references
public record CreatePackageTypeCommand(string Name, string Description) : IRequest<PackageTypeDTO>;
1 reference
public class CreatePackageTypeCommandHandler : IRequestHandler<CreatePackageTypeCommand, PackageTypeDTO>
{
    private readonly IRepository<PackageType> _repository;
    private readonly IMapper _mapper;

    0 references
    public CreatePackageTypeCommandHandler(IRepository<PackageType> repository, IMapper mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }

    0 references
    public async Task<PackageTypeDTO> Handle(CreatePackageTypeCommand request, CancellationToken cancellationToken)
    {
        var entity = PackageType.Builder()
            .WithName(request.Name)
            .WithDescription(request.Description)
            .Build();

        await _repository.AddAsync(entity);

        return _mapper.Map<PackageTypeDTO>(entity);
    }
}
```

Rysunek 22. Komenda oraz jej handler realizujące tworzenie nowego rodzaju opakowania

Komenda i zapytanie muszą implementować interfejs *IRequest* natomiast klasa obsługująca je – *IRequestHandler*.

Bardzo ważną rzeczą, którą należy odnotować jest reguła w imię której obiekty domenowe (nasze encje) nie mogą „wyciekać” poza warstwę przypadków użycia/aplikacji. Uzasadnień tej reguły jest co najmniej kilka: stosując obiekty DTO unikamy konieczności oznaczania naszych obiektów domenowych atrybutami, zwiększamy elastyczność przy decydowaniu, które elementy chcemy zwrócić klientowi, dla różnych typów obsługiwanych mediów możemy wykorzystywać różne DTO.

Jednak utrzymywanie sporej ilości obiektów DTO może być czasochłonne w momencie procesu transformacji obiektu DTO na model domenowy i odwrotnie. Z pomocą przychodzi tutaj biblioteka *AutoMapper*, która dokonuje tego za nas. Naszym zadaniem jest tylko utworzenie odpowiedniego profilu, w którym wskazujemy dwa obiekty które mają być na siebie tłumaczone.

4.2.6 ShipmentService.Infra

Głównym zadaniem warstwy infrastruktury jest obsługa zewnętrznych źródeł danych takich jak serwisy, pliki, bazy danych. Innymi słowy warstwa infrastruktury odpowiada za operacje I/O. Postanowiłem nieznacznie zmodyfikować w tym miejscu projekt tworząc oddzielny folder *Infrastructure* a w nim dwa projekty: *ShipmentService.Infra* oraz

ShipmentService.Persistence. Uznałem, że ilość kodu związanego z obsługą *EntityFramework* jest na tyle duża, że warto zawrzeć ją w osobnym projekcie.

W warstwie tej zawarłem obsługę serwisu związanego z wysyłaniem wiadomości email, komunikacją z brokerem wiadomości oraz obsługą Json Web Tokens.

Dodatkowo w warstwie infrastruktury stworzyłem prostą implementację interfejsu *IDateTimeService*:

```
0 references
public class DateTimeService : IDateTimeService
{
    1 reference
    public DateTime GetCurrentDateTime() => DateTime.Now;
}
```

Rysunek 23. Implementacja własnego interfejsu dostarczającego daty

Rozwiązanie takie może na wydawać się na pozór bezcelowe skoro język udostępnia metody dostarczające obsługę dat. Implementacja nie robi w zasadzie nic innego tylko wywołuje tę właśnie metodę obudowaną tylko interfejsem. Powodem takiego rozwiązania jest ułatwienie testowania aplikacji. W sytuacji gdy korzystalibyśmy ze „zwykłych" dat – nie mielibyśmy w testach możliwości podstawiania danych testowych.

4.2.7 ShipmentService.Persistence

Projekt ten jest odpowiedzialny za obsługę bazy danych. Zdecydowałem się na użycie frameworka ORM – Entity Framework. Jest to framework mapujący obiekty bazodanowe na obiekty języka C#. Wykorzystałem tutaj podejście code first. Oznacza to, że struktura bazy danych jest odzwierciedleniem obiektów języka C#. W tym podejściu najpierw tworzymy klasy które następnie za pomocą narzędzia zwanego migracjami Przeobrażamy w tabele bazodanowe. Framework automatycznie dokonuje konwersji typów. W niektórych sytuacjach mamy poniekąd możliwość ręcznego sterowania takimi konwersjami. Zazwyczaj realizowane to jest przy pomocy adnotacji w klasach i właściwościach klas. W naszym przypadku byłoby to zaburzeniem idei Domain Driven Design, która mówi, że obiekty domenowe powinny być jak najbardziej czyste i niezależne od frameworków. Dlatego zdecydowałem się na użycie innego dostępnego rozwiązania, mianowicie napisałem domyślną implementację dostarczanej przez framework metody *OnModelCreating*, w której wskazałem konieczność użycia ręcznych konfiguracji encji. Przykładowa implementacja takiej konfiguracji wygląda jak ta poniżej:

```
0 references
public class ShipmentConfiguration : IEntityTypeConfiguration<Shipment>
{
    0 references
    public void Configure(EntityTypeBuilder<Shipment> builder)
    {
        builder.Property(x => x.Status)
            .HasConversion(
            x => x.ToString(),
            x => (ShipmentStatus)Enum.Parse(typeof(ShipmentStatus), x));

        builder.Property(x => x.Version).IsConcurrencyToken();

        builder.HasMany(x => x.Cargos)
            .WithOne(x => x.Shipment)
            .OnDelete(DeleteBehavior.Cascade);

        builder.HasMany(x => x.AdditionalServices)
            .WithMany(x => x.Shipments);

        builder.HasOne(x => x.PickupPlace)
            .WithMany()
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(x => x.DeliveryPlace)
            .WithMany()
            .OnDelete(DeleteBehavior.NoAction);

        builder.Property(x => x.SequentialNumber)
            .HasDefaultValueSql("NEXT VALUE FOR ShipmentSequentialNumber");

        builder.Property(x => x.WaybillNumber)
            .HasComputedColumnSql(@"RIGHT('000000000' + cast(sequentialnumber as varchar(10)), 9) + '/' + cast(YEAR(CreationDate) as varchar(4))");

        builder.Ignore(x => x.DomainEvents);
    }
```

Rysunek 24. Implementacja konfiguracji encji w EntityFamework

Na przykładzie encji *Shipment* pokazuję w jaki sposób można zdefiniować relacje pomiędzy obiektami. Użycie metod *HasMany* oraz *WithOne* w rezultacie powoduje utworzenie kluczy obcych.

Zmiana stanu obiektu w aplikacji nie powoduje automatycznej zmiany wartości rekordu w bazie danych. Odpowiada za to dostarczana przez *EntityFramework* metoda *SaveChangesAsync*. Zdecydowałem się również na nadpisanie i tej metody. Warstwa domeny zostanie opisana w kolejnym podrozdziale jednak wspomnę tutaj tylko, że każdy z obiektów domenowych mających swoje odzwierciedlenie w bazie danych implementuje abstrakcyjną klasę *Entity*, która to zawiera takie pola jak data utworzenia, data modyfikacji czy identyfikator użytkownika tworzącego i modyfikującego obiekt. Napisanie metody *SaveChangesAsync* pozwoliło mi na utworzenie mechanizmu audytowania. W nowej implementacji metody sprawdzam czy zapisywana encja jest tworzona czy modyfikowana i w zależności od tego ustawiam wartości odpowiednich pól:

```
5 references
public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = new CancellationToken())
{
    foreach (var entry in ChangeTracker.Entries<Entity>())
    {
        switch (entry.State)
        {
            case EntityState.Added:
                entry.Entity.CreationDate = DateTime.Now;
                entry.Entity.CreatedBy = _currentUserService.UserId;
                break;
            case EntityState.Modified:
                entry.Entity.ModificationDate = DateTime.Now;
                entry.Entity.ModifiedBy = _currentUserService.UserId;
                break;
        }
    }

    var result = await base.SaveChangesAsync(cancellationToken);
    await DispatchEvents();

    return result;
}
```

Rysunek 25.Własna implementacja metody SaveChangesAsync

Kolejnym elementem zaimplementowanym w projekcie warstwy persystencji jest wzorzec *Repozytorium*. Należy tutaj zaznaczyć, że repozytorium nie jest warstwą dostępu do danych. Z założenia repozytorium jest bytem ulokowanym pomiędzy warstwą dostępu do danych oraz logiką biznesową. Główną przesłanką do wprowadzenia wzorca repozytorium jest realizacja jednej z idei towarzyszącej *Domain Driven Design – persistence igrnorance.* Chodzi o to, że logika oraz obiekty biznesowe nie dysponują żadną wiedzą na temat sposobu przechowywania danych oraz dostępu do nich: odczytu, zapisu, modyfikacji. Dodatkowo implementacja wzorca repozytorium umożliwia nam łatwe przeprowadzenie testów jednostkowych ponieważ obiekty repozytorium można mockować dostarczając dane z pominięciem prawdziwej bazy danych. Wzorzec repozytorium pozwala na znaczną elastyczność doboru sposobu składowania danych. Dane niekoniecznie muszą być utrwalane w bazie danych. Równie dobrze mogą być zapisywane do plików *csv* czy też do innych serwisów. Z reguły każdy obiekt domenowy posiada swoje repozytorium. Repozytorium to nic innego jak zbiór metod służących do odczytu, zapisu, modyfikacji czy też usuwania danych. Ja zdecydowałem się na utworzenie generycznego repozytorium. Zapewniło mi to pewną uniwersalność i pozwoliło na uniknięcie duplikacji kodu. Zaprojektowane przeze mnie repozytorium umożliwia wykonanie standardowych operacji na każdym przekazanym do niego obiekcie. Dodatkowo wprowadziłem w interfejsie generycznym ograniczenie (*constraint*) zgodnie z którym jedynym typem jaki może być przekazany do klasy generycznej implementującej tenże interfejs może być klasa:

```
71 references
public interface IRepository<T> where T : class
{
    23 references
    Task<T> GetByIdAsync(Guid id, string includes = null);
    4 references
    Task<T> GetByIdOfCurrentUserAsync(Guid id, string includes = null);
    5 references
    Task<IReadOnlyList<T>> ListAllAsync(string includes = null);
    2 references
    Task<IReadOnlyList<T>> ListAllOfCurrentUserAsync(string includes = null);
    6 references
    Task<T> AddAsync(T entity);
    10 references
    Task UpdateAsync(T entity);
    1 reference
    Task DeleteAsync(T entity);
    4 references
    Task DeactivateAsync(T entity);
    1 reference
    Task<IReadOnlyList<T>> GetPagedReponseAsync(int page, int size);
}
```

Rysunek 26. Interfejs wzorca repozytorium

4.2.8 ShipmentService.Domain

Kolejną – ostatnią i zarazem najważniejszą warstwą – jest warstwa domeny. To tej warstwie zawarta jest cała logika biznesowa związana z przebiegiem programu. Jednak przed opisaniem tej warstwy chciałbym przedstawić krótki zarys koncepcji, którą zdecydowałem się w projekcie. Chodzi o Domain Driven Design.

4.2.8.1 Charakterystyka podejścia Domen Driven Design

DDD stało się popularne po opublikowaniu przez Erica Evansa książki „Domain-driven Design: Tackling Complexity in the Heart of Software”[7]. DDD ingeruje w tradycyjną architekturę warstwową rozdzielając ją na dwie główne warstwy: logikę aplikacji oraz logikę biznesową. Dzięki takiemu rozdzieleniu warstwa logiki biznesowej skupia się w zasadzie tylko i wyłącznie na modelowaniu dziedziny biznesowej. Uwalnia się od rozwiązywania szczegółów implementacyjnych, technicznych. Staje się wolna od frameworków. Tak więc przykładowo całkowicie niedopuszczalne jest mieszanie jest logiki biznesowej w warstwa odpowiedzialnych między innymi za persystencję danych.

Oprócz zagadnień typowo technicznych Domain Driven Design definiuje również inne podejście w kontaktach pomiędzy biznesem a osobami typowo technicznymi, odpowiedzialnymi za powstawanie i rozwój oprogramowania. Evans wprowadza pojęcie języka wszechobecnego – ubiquitous language. Jest to język, który używany jest przez wszystkich członków zespołu. Każde pojęcie specyficzne dla danej dziedziny biznesowej jest dokładnie przeanalizowane i jego znaczenie znane jest obu stronom. W związku z czym

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

unikamy nieporozumień związanych z używaniem pojęć, których rozumienie definiujemy w odmienny sposób[8].

Kolejnym pojęciem wartym krótkiego opisania jest tzw. *bounded context*, czyli kontekst ograniczony. Jest to określony fragment działalności biznesowej – pewna perspektywa spojrzenia na niejednokrotnie te same elementy z różnych stron. Przykładowo w nawiązaniu do tematyki tejże pracy kontekstem ograniczonym w firmie kurierskiej może być dział księgowości oraz dział transportu. Obydwa działy będą posługiwały się takimi pojęciami jak przesyłka czy faktura jednak dla każdego z tych działów pojęcia te będą miały różne znaczenie i inną wagę.

Encje jest to pojęcie, które stało się szerzej znane z używania z narzędzi mapowania relacyjno-obiektowych, czyli tak zwanych ORM. W przypadku DDD mówimy o encji wówczas gdy według nas dany obiekt, klasa wymaga odróżnienia swoich egzemplarzy od pozostałych. Innymi słowy encje powinny posiadać swój unikalny identyfikator tak aby każda z encji była możliwa do zidentyfikowania. Jeszcze inaczej można powiedzieć, że każda encja ma swoją tożsamość i w ciągu całego cyklu swojego życia będzie ulegała modyfikacjom. Tak więc aby dokonać porównania dwóch encji wystarczy dokonać porównania ich identyfikatorów. Nawet jeżeli pozostałe składowe encji będą identyczne – obiekty są różne. Ten wymóg implikuje konieczność wprowadzenia modyfikacji w standardową logikę porównywania obiektów zawartą w języku C# co zostanie zaprezentowane w dalszej części pracy.

Typy wartościowe są w pewnym sensie przeciwieństwem encji. *Value Objects* różnią się głównie tym od encji, że nie mają swojej tożsamości. Tak więc z punktu widzenia użytkownika tego obiektu nie ma znaczenia, którego egzemplarza użyjemy. Eric Evans w swojej książce daje na to dość obrazowy przykład: gdy dziecko rysuje jakiś obrazek przywiązuje ono sporą uwagę do koloru. Jeżeli na stole będą leżały dwie kredki o tym samym kolorze – dziecko nie będzie miało dylematu, którą konkretnie kredkę ma użyć gdyż w danej chwili najistotniejszy jest dla niego kolor[9]. Typy wartościowesą w pewnym sensie obiektami jednorazowymi. Wiąże się to z pojęciem niezmienności – *immutability*. Chodzi o to, że każda modyfikacja dokonana na składowej typu wartościowego powoduje utworzenie nowego jego egzemplarza.

---

[8] Vaughn Vernon, DDD Kompendium Wiedzy, Helion, Gliwice 2018, s. 23

[9] Eric Evans, Domain-Driven Design Tackling Complexity in the Heart of Software, Addison-Wesley, Boston 2004, s. 97

2022

Ostatnim wartym opisania pojęciem z dziedziny DDD jest agregat. Agregat jest zbiorem obiektów: encji oraz typów wartościowych. Każdy agregat składa się również z korzenia agregatu. Jest to zazwyczaj encja najważniejsza z punktu widzenia danego kontekstu ograniczonego. W pewnym sensie w przypadku pojęcia agregatów możemy mówić o zastosowaniu kompozycji – usunięcie korzenia agregatu zazwyczaj pociąga za sobą usunięcie wchodzących w skład agregatu. Bardzo ważnym elementem jest tutaj koncepcja hermetyzacji agregatu. Chodzi o to, że agregat nie powinien pozwalać na modyfikowanie swoich składowych przy pomocy zwykłych getterów oraz setterów. Takie zachowanie stanowi zazwyczaj antywzorzec zwany *anemicznym modelem* dziedziny. Model taki składa się zazwyczaj tylko i wyłącznie z getterów oraz seterów bez jakiejkolwiek logiki walidacyjnej. Modyfikacji zazwyczaj dokonujemy poprzez użycie specjalnie zaprojektowanych do tego metod biznesowych i najczęściej nie modyfikujemy bezpośrednio składowych encji danego agregatu. Metody modyfikujące encja składowe są definiowane w korzeniu agregatu gdyż zazwyczaj dopiero ujęcie całościowe daje nam odpowiedź czy poszczególna składowa – zgodnie z naszymi regułami biznesowymi – może zostać zmodyfikowana. Tym samym unikamy wycieku logiki biznesowej do zewnętrznych obiektów czy też obszarów projektu.

Podobnie sprawa ma się ze zjawiskiem tak zwanego *primitive obsession*. Zjawisko to zachodzi wtedy gdy mamy tendencję do przesadnego stosowania typów prostych. Przykładowo modelując klasę adresu potrzebny nam będzie kod pocztowy. Naturalnym odruchem wydawałoby się utworzenie pola bądź właściwości typu string. W tym momencie rodzi się jednak problem walidacji kodu pocztowego. Zachodzi pytanie kiedy i w którym miejscu powinniśmy umieścić logikę sprawdzającą czy podany kod pocztowy jest poprawnym kodem dla danego kraju. Tego typu zła implementacja prowadzi do mnożenia klas typu *util, helpers*, itd. Rozwiązanie jest banalne choć mało oczywiste – powinniśmy utworzyć odrębną klasę kodu pocztowego, w której to np. w konstruktorze dokonujemy sprawdzania poprawności kodu tym samym nie dopuszczając do powstania obiektu w nieprawidłowym stanie. Przykład ten zostanie zaprezentowany w kolejnym podrozdziale pracy.

4.2.8.2 Implementacja Domain-Driven Design w projekcie

Omawianie warstwy domeny zacznę od obiektów encji. Każda z encji dziedziczy po abstrakcyjnej klasie *Entity*. Klasa ta zawiera zestaw pól, w które powinna być „uzbrojona" każda encja w projekcie. Są to między innymi takie pola jak data utworzenia, data modyfikacji metoda *Deactivate*, która służy do logicznego usuwania obiektów. Dzięki temu możliwe było

również stworzenie uniwersalnego mechanizmu audytowania operacji na encjach, o którym wspominałem w podrozdziale opisującym warstwę persystencji.

Kolejną abstrakcją jest klasa *AggregateRoot*. Klasa ta zawiera tylko dwa obiekty/ Pierwszym z nich jest właściwość *Version*. Właściwość ta jest wykorzystywana do ustalania czy dany agregat uległ modyfikacji. Jeżeli tak, wartość właściwości jest inkrementowana. Drugim składnikiem tejże klasy jest metoda *CheckRule*. Metoda ta służy do zgłaszania wyjątków biznesowych w sytuacji gdy przekazana do metoda reguła biznesowa zostanie zidentyfikowana jako złamana.

Również dla tak zwanych *Value Objects* została zdefiniowana klasa abstrakcyjna. Głównym celem przyświecającym utworzeniu tej klasy jest konieczność nadpisania reguł sprawdzających równość dwóch obiektów wartościowych. Domyślnie w języku C# dwa typy referencyjne są równe gdy ich referencje są równe, czyli wskazują na ten sam obiekt. W przypadku *Value Objects* chcemy osiągnąć nieco inny efekt. Dwa obiekty wartościowe są sobie równe gdy wszystkie ich składowe są sobie równe. Tak więc konieczne było nadpisanie metod *Equals* oraz operatorów równości i nierówności. To z kolei implikuje z zasadami programowania obiektowego przedefiniowanie metody *GetHashCode*, która wylicza wartość skrótu każdego z obiektów. Poniżej przedstawiam przykład klasy *ZipCode* będącej typowym *Value Object*:

```
8 references
public class ZipCode : ValueObject
{
    private readonly string _value;

    3 references
    public ZipCode(string value)
    {
        var regex = new Regex(@"[0-9]{2}\-[0-9]{3}");

        if (!regex.IsMatch(value))
            throw new InvalidZipCodeException(value);

        _value = value;
    }

    2 references
    public string Value
    {
        get { return _value; }
    }

    0 references
    public override string ToString()
    {
        return _value;
    }

    public static implicit operator string(ZipCode zipCode)
    {
        return zipCode.Value;
    }

    public static explicit operator ZipCode(string value)
    {
        return new ZipCode(value);
    }
}
```

Rysunek 27. Przykład implementacji Value Object

Tak więc kod pocztowy tworzony jest poprzez wywołanie konstruktora przyjmującego jako parametr wartość kodu. Bezpośrednio w konstruktorze odbywa się sprawdzanie poprawności kodu. Jeżeli przekazana wartość jest niepoprawna – zostanie zgłoszony wyjątek. Dzięki temu obiekt kodu pocztowego nie zostanie utworzony z niepoprawną wartością. W celu zwiększenia komfortu korzystania z klasy kodu pocztowego utworzyłem przeciążone definicje operatorów niejawnego rzutowania co pozwala na korzystanie z kodu pocztowego jak ze zwykłych wartości typu *string*.

Encje – jak już wspomniałem wcześniej – to obiekty posiadające swoją tożsamość i utrwalane są przy pomocy warstwy persystencji. Każda z encji – oprócz tej będącej aggregate rootem – dziedziczy po omawianej już abstrakcyjnej klasie *Entity*:

```
public class Cargo : Entity
{
    3 references
    public string Description { get; private set; }
    3 references
    public int PackageQuantity { get; private set; }
    10 references
    public CargoDimensions CargoDimensions { get; private set; }
    2 references
    public string Barcode { get; private set; }
    1 reference
    public string SSCC { get; private set; }
    2 references
    public string CustomerRefNO { get; private set; }
    2 references
    public string PackageType { get; private set; }
    1 reference
    public Shipment Shipment { get; private set; }

    0 references
    private Cargo() { }
    1 reference
    private Cargo(
        decimal length,
        decimal width,
        decimal height,
        string description,
        int packageQuantity,
        string barcode,
        string sscc,
        string customerRefNo,
        string packageType)
    {
        SetCargoDimensions(length, width, height);
        SetDescription(description);
        SetPackageQuantity(packageQuantity);
        SetBarcode(barcode);
        SetSSCC(sscc);
        SetCustomerRefNO(customerRefNo);
        SetPackageType(packageType);
    }
```

Rysunek 28. Przykład typowej encji wraz z jej właściwościami i konstruktorami

Rysunek nie pokazuje oczywiście całej encji. Zawarłem na nim tylko jej właściwości oraz konstruktory. Wszystkie setery we właściwościach encji mają prywatny modyfikator dostępu. To oznacza, że nie możemy bezpośrednio modyfikować ich z poziomu encji. Każda z encji wyposażona jest w odpowiedni zestaw metod umożliwiających modyfikowanie określonych jej składowych. Powód takiego podejścia został wyjaśniony już wcześniej, chodzi o to, że mogą istnieć określone reguły biznesowe bądź konieczność weryfikacji poprawności

danych, które chcemy przypisać poszczególnym obiektom. Przykład takiego zachowania pokazuje poniższy rysunek:

```
2 references
public void SetPackageQuantity(int packageQuantity)
{
    if (packageQuantity <= 0)
        throw new ArgumentException(null, nameof(this.PackageQuantity));
    this.PackageQuantity = packageQuantity;
}
```

Rysunek 29. Przykład sprawdzenia poprawności ilości opakowań

Otóż dzięki użyciu metody *SetPackageQuantity* możemy nie tylko ustawić ilość opakowań, ale – co ważniejsze – wcześniej sprawdzić czy podana ilość opakowań nie jest przypadkiem liczbą ujemną. W przypadku użycia modyfikatorów pozwalających na bezpośrednią zmianę wartości właściwości, musielibyśmy taką logikę pisać bezpośrednio w ciele właściwości, co pozbawia nas pewnej elastyczności – nie jesteśmy w stanie wykorzystać logiki właściwości poza nią…

Drugą rzeczą wartą odnotowania są dwa konstruktory prywatne i brak konstruktora publicznego. To oznacza, że nie jesteśmy w stanie zinstancjonować bezpośrednio danego egzemplarza klasy. W tym celu został użyty wzorzec budowniczego, którego użycie zwraca nam egzemplarz klasy. Przy czym zastosowałem tu nieco inne podejście. Przy tworzeniu obiektu posłużyłem się koncepcją tzw. *fluent interface*. W dużym uproszczeniu polega to na tym, że wywołanie danej metody klasy zwraca ją samą co z kolei w jednym bloku instrukcji pozwala na wywołanie kolejnej metody z tejże klasy. W efekcie utworzenie obiektu ładunku (*klasy Cargo*) wygląda tak:

```
var cargo = Cargo.Builder()
    .WithLength(request.length)
    .WithWidth(request.width)
    .WithHeight(request.height)
    .WithDescription(request.description)
    .WithPackageQuantity(request.packageQuantity)
    .WithPackageType(request.packageType)
    .WithBarcode(request.barcode)
    .WithCustomerRefNo(request.customerRefNo)
    .Build();
```

Rysunek 30. Przykład tworzenia obiekty przy pomocy fluent interface

O ile samo utworzenie takiego mechanizmy jest stosunkowo żmudne to korzystanie z niego jest bardzo poręczne i przyjemne. Zwłaszcza, że daje nam pewną swobodę co do tego które metody budowniczego zostaną wywołane. W tym konkretnym przypadku oparłem dodatkowo poszczególne metody budowniczego o interfejsy. Wywołanie danej metody zwracam interfejs, który zawiera definicję jednej tylko metody. Dzięki temu budowanie obiektu

prowadzi nas z góry ustaloną ścieżką wywołania. Tym samym jeżeli powyższy obiekt ładunku wymaga podania mu długości, wysokości czy szerokości to użycie tych metod budowniczego będzie wymuszone przez zwracanie interfejsów tylko dla tych metod. Z kolei na samym końcu podpowiedzą się te metody, których użycie nie jest wymagane. Przykładowo jeżeli klient nie ma potrzeby przesłania swojego numeru referencyjnego opakowania to może pominąć metodę *WithCustomerrefNo* i przejść bezpośrednio to metody *Build*, która zwróci mu gotowy egzemplarz klasy.

Najważniejszym i najbardziej rozbudowanym w naszym projekcie jest obiekt będący *aggregate rootem*. W tym projekcie jest jeden taki obiekt i jest to klasa *Shipment*. Jest to korzeń agregatu gdyż zawiera w sobie w zasadzie wszystkie pozostałe encje. Działa on na bardzo podobnych zasadach jak encje jest jednak rozbudowany o kilka mechanizmów i rozwiązań.

Jak wiadomo przesyłka zawiera listę ładunków oraz usług dodatkowych, które zostały zdefiniowane w postaci właściwości. Jednak w tym przypadku ustawienie modyfikatorów dostępu do listy będzie niezbyt pomocne bo o ile w klasach będących składowymi naszego korzenia agregatu byliśmy w stanie zahermetyzować ich składowe tak listy operują na wbudowanych we framework interfejsach. Mówiąc inaczej – ustawiając modyfikator dostępu *set* jako prywatny zapobiegamy tylko zmianie bądź usunięciu referencji do obiektu. Jeżeli mamy dostęp do listy przy pomocy akcesoria *get* to mamy dostęp do jej metod wynikających z konieczności zaimplementowania takich interfejsów jak *IEnumerable, IList, ICollection*, itd… To wymagało poczynienia pewnego zabiegu. Listy zostały zadeklarowane jako całkowicie prywatne obiekty – związku z czym możemy operować na nich tylko i wyłącznie wewnątrz instancji klasy. Natomiast „na zewnątrz" udostępniamy właściwość implementującą interfejs *IReadOnlyCollection* i korzystając z mechanizmu kowariancji zwracamy referencję do naszych prywatnych list. W rezultacie za pomocą właściwości publicznych zwracamy tylko i wyłącznie zestaw metod umożliwiających odczyt list – bez możliwości ich modyfikacji:

```
private readonly List<Cargo> _cargos = new List<Cargo>();
private readonly List<Service> _additionalServices = new List<Service>();
4 references
public IReadOnlyCollection<Cargo> Cargos => _cargos.ToList();
2 references
public IReadOnlyCollection<Service> AdditionalServices => _additionalServices.ToList();
4 references
```

Rysunek 31. Udostępnienie list tylko do odczytu

Dodatkowo korzeń agregatu dysponuje metodami umożliwiającymi operowanie na obiektach, których jest właścicielem. Takie rozwiązanie jest konieczne gdyż dopiero z punktu widzenia korzenia agregatu mamy ogląd na całokształt obiektu na którym operujemy. Chodzi o to, że zmiana właściwości danej klasy w oderwaniu od całości kontekstu jakim jest korzeń

agregatu mogła by nie dać nam kompletu informacji potrzebnych do oceny czy dana zmiana jest poprawna zarówno z technicznego jak i biznesowego punktu widzenia. Przykładem takiego podejścia jest zdefiniowana reguła biznesowa o nazwie *ShipmentMustHaveAtLeastOnecargoRule:*

```
2 references
public class ShipmentMustHaveAtLeastOneCargoRule : IBusinessRule
{
    private readonly IReadOnlyCollection<Cargo> _cargos;

    1 reference
    public ShipmentMustHaveAtLeastOneCargoRule(IReadOnlyCollection<Cargo> cargos)
    {
        _cargos = cargos ?? throw new ArgumentNullException(nameof(cargos));
    }

    4 references
    public string Message => "Shipment must have at least one cargo";

    2 references
    public bool IsBroken() => _cargos.Count == 0;
}
```

Rysunek 32. Przykład reguły biznesowej

Jak sama nazwa wskazuje reguła ta sprawdza, czy w przesyłka zawiera w sobie choć jeden ładunek. Jeżeli nie to znaczy, że nie wiedzielibyśmy co tak naprawdę przyjmujemy do przewozu, co z punktu widzenia operacyjnego jest niedopuszczalne. Reguła ta jest przykładem przedstawianej wcześniej metody *CheckRule* będącej częścią abstrakcyjnej klasy *AggregateRoot*. Dzięki temu mechanizmowi możemy definiować niezliczoną ilość reguł biznesowych, których sprawdzanie w przypadku korzenia agregatu odbywa się przykładowo w metodzie zatwierdzającej przesyłkę:

```
1 reference
public bool ApproveShipment()
{
    CheckRule(new DeliveryDateMustBeLaterThanPickupDateRule(this.PickupDate, this.DeliveryDate));
    CheckRule(new PickupPlaceMustBeDifferentThanDeliveryPlaceRule(this.PickupPlaceId, this.DeliveryPlaceId));
    CheckRule(new ShipmentMustHaveAtLeastOneCargoRule(this.Cargos));

    this.Status = ShipmentStatus.Approved;

    DomainEvents.Add(new ShipmentApprovedEvent(this));

    return true;
}
```

Rysunek 33. Sprawdzanie reguł biznesowych

Poprzez reguły biznesowe w naturalny sposób przechodzimy do omówienia kolejnej składowej warstwy domeny jakim są wyjątki. Generalnie panuje zasada, w myśl której nie powinno się opierać logiki działania programu na wyjątkach. Z drugiej strony framework udostępnia mechanizm definiowania własnych wyjątków. Kwestią odpowiedniej interpretacji pozostaje zagadnienie czy złamanie reguły biznesowej może być potraktowane jako wyjątek aplikacji. W przypadku podejścia DDD – jak najbardziej. Jeżeli złamanie danej reguły

biznesowej spowodowałoby możliwość utworzenia obiektu w nieprawidłowym stanie to jest to sytuacja jak najbardziej wyjątkowa i powinniśmy wszelkimi metodami dążyć do zapobieżenia jej.

Każdy z wyjątków dziedziczy po abstrakcyjnej klasie *DomainExceptionBase*, która to definiuje kilka przeciążeń konstruktorów oraz wirtualną właściwość *Code.* Właściwość ta pozwala na oznaczenie danego wyjątku unikalnym kodem co w teorii powinno ułatwić w przyszłości na przykład wyświetlanie treści wyjątków w innych językach – w przypadku regionalizacji naszej aplikacji:

```
15 references
public abstract class DomainExceptionBase : Exception
{
    12 references
    public virtual string Code { get; }

    0 references
    public DomainExceptionBase()
        : base()
    {
    }

    7 references
    public DomainExceptionBase(string message)
        : base(message) { }
}
```

Rysunek 34. Bazowa klasa wyjątku

Zdecydowałem się na użycie wyjątków również z dwóch innych powodów. Jednym z nich jest łatwiejsze testowanie aplikacji. Sprawdzając dane zachowanie aplikacji będziemy wiedzieć jakiego konkretnie obiektu mamy się spodziewać w przypadku testowania tzw. *unhappy path*, czyli scenariusza niepowodzenia. Drugim powodem była obsługa błędów. Wspominałem już przy okazji opisywania warstwy API mechanizm *middleware* dzięki któremu możliwe było tłumaczenie poszczególnych wyjątków na kody HTTP.

Pokazana na rysunku nr 29 metoda *ApproveShipment* zawiera jeszcze jeden element, o którym chciałbym wspomnieć. Chodzi mianowicie o zdarzenia domenowe. W tym konkretnym przypadku zdarzeniem takim jest fakt zatwierdzenia przesyłki. Obsługa zdarzeń w projekcie realizowana jest z użyciem opisywanej już biblioteki *MediatR* i działa na bardzo zbliżonych zasadach jak obsługa wzorca CQRS poprzez tę bibliotekę. Czyli deklarujemy określone zdarzenie a następnie poprzez odpowiedni handler implementujący wbudowany interfejs obsługujemy dane zdarzenie. W tym konkretnym przypadku wywołanie zdarzenia *ShipmentApprovedEvent* powoduje w konsekwencji wyemitowanie poprzez broker wiadomości *RabbitMQ* komunikatu o pojawieniu się nowej przesyłki w systemie. Dzięki temu pozostałe systemy w firmie są w stanie zareagować na fakt utworzenia nowej przesyłki. Oczywiście jeżeli tylko zasubskrybowały się na odczyt komunikatów z danej kolejki.

4.2.9 ShipmentService.UI

Ostatnim elementem projektu jest tak zwana warstwa prezentacji. Jest to strona internetowa, która działa jako klient opisanego powyżej serwisu WebApi. Na uwagę zasługuje fakt, że strona została napisana w stosunkowo nowej technologii jaką jest *Blazor WebAssembly*, która teoretycznie pozwala na prawie całkowite uniknięcie programowania w języku *JavaScript*. WebAssembly to technologia zbliżowana do maszyny wirtualnej, która pozwala na wykonywanie plików binarnych, które zostały już skompilowane. Taki zabieg pozwala na osiągnięcie wydajności zbliżonej do natywnej. Poniższy rysunek prezentuje jedną z możliwości użycia technologii Blazor. Tworząc przycisk dodajemy obsługę naciśnięcia przycisku wskazując metodę C# a nie – jak byłoby to w przypadku tradycyjnych technologii – JavaScript:

```
<button type="button" @onclick="AddCargo" class="btn btn-success">
    <span class="oi oi-plus"></span>
    </button>
</td>
```

Rysunek 35. Wywołanie metody C# podczas obsługi naciśnięcia przycisku

Z racji tego, że projekt strony jest poniekąd tylko narzędziem dzięki któremu chciałem zaprezentować działanie omówionego na powyższych stronach serwisu, skupię się tylko na jego najważniejszych elementach.

4.2.9.1 Wykorzystanie HttpClientFactory

Naturalnym mogłoby wydawać się użycie do komunikacji obiektu HttpClient jednak takie podejście jest groźne z jednego zasadniczego powodu: pomimo iż klasa ta implementuje interfejs *IDisposable* nie uwalnia ona natychmiastowo socketów sieciowych. To – przy intensywnym użyciu naszego serwisu – mogłoby w pewnym momencie całkowicie zablokować ruch TCP na naszym serwerze. Problem ten całkowicie rozwiązuje użycie klasy HttpClientFactory.

Również w przypadku tego projektu pokusiłem się o utworzenie swego rodzaju repozytorium generycznego, którego zadaniem jest przeprowadzania operacji CRUD na przekazywanych obiektach. Ponadto stworzyłem statyczną klasę z zadeklarowanymi publicznymi zmiennymi typu string, z których każdy przechowuje adres tak zwanego endpointu do wykonywania poszczególnych akcji kontrolera naszego serwisu.

2022

4.2.9.2 Obsługa Json Web Tokens

Jak już zostało wspomniane wcześniej, obiekty JWT służą do autentykacji klientów serwisów bez konieczności każdorazowo przesyłania drogą sieciową danych typu nazwa użytkownika i hasło co znacząco poprawia bezpieczeństwo aplikacji.

Implementacja JWT jest stosunkowo prosta. Przy pomocy *HttpClientFactory* wysyłam żądanie http typu *POST* z poświadczeniami użytkownika. Jeżeli w odpowiedzi otrzymam kod http 200 (ok) to zapisuję obiekt tokena w tzw. storage przeglądarki i następnie wykorzystuję tenże obiekt dodając go do każdego żądania http:

```
if (!response.IsSuccessStatusCode)
    return false;

var content = await response.Content.ReadAsStringAsync();
var token = JsonConvert.DeserializeObject<ResponseModel<SignInResultModel>>(content);

await _localStorage.SetItemAsync("authToken", token.Result.Token);
await ((ApiAuthenticationStateProvider)_authenticationStateProvider).SignedIn();
```

Rysunek 36.Zapisanie w pamięci przeglądarki tokena uzyskanego w wyniku poprawnego zalogowania się

Oczywiście każdy token ma swoją ważność. Po skończeniu się jego ważności kolejne żądanie http zwróci kod błędu 401 (unauthorized) co wymusza na nas ponowne wykonanie żądania do endpointu autentykacyjnego serwisu.

4.2.9.3 Poprawa wydajności aplikacji klienckiej przy użyciu mechanizmu cachowania

Często zdarza się, że przy dużych obciążeniach doświadczamy spadku wydajności naszej aplikacji. Jest to szczególnie uciążliwe dla użytkownika końcowego, szczególnie w przypadku branż gdzie czas odgrywa znaczącą rolę. Oczywiście jest kilka metod zwiększenia wydajności działania aplikacji: poprawa zapytań SQL, skalowanie wertykalne czy horyzontalne. Jednak możemy pokusić się o mniej kosztowną metodę – cachowanie. Cachowanie to technika pozwalająca na przechowywanie często pobieranych danych w źródłach szybszych od ich pierwotnego pochodzenia. Najczęściej stosowane są dwie techniki cachowania: in-memory caching oraz distributed caching. Pierwsza pozwala na przechowywanie danych w pamięci serwera. Druga na zewnętrznym serwerze. W przypadku pamięci RAM wydajność wzrasta drastycznie natomiast napotykamy to szereg ograniczeń: pamięć RAM jest zazwyczaj mocno ograniczona, więc musimy ostrożnie dobierać dane podlegające cachowaniu, reset serwera usuwa cachowane dane z pamięci w związku z czym

aplikacja bezpośrednio po restarcie serwera może działać bardzo wolno. Dlatego też w przypadku omawianego projektu zdecydowałem się na drugie rozwiązanie. Do tego celu wybrałem bazę Redis Cache. Jest to baza przechowująca dane jako pary klucz, wartość. Implementacja mechanizmu jest dość prosta. Pobierając dane z mało wydajnego źródła danych najpierw sprawdzam czy dane zostały wcześniej zapisane w pamięci cache:

```
10 references
public async Task<IList<T>> Get(string url)
{
    var cacheKey = $"List_{typeof(T).ToString()}";
    var res = await _cacheService.GetCacheValueAsync<T>(cacheKey);

    if (res is { })
    {
        var cachedResult = JsonConvert.DeserializeObject<ResponseListModel<T>>(res);
        return cachedResult.Result;
    }
```

Rysunek 37. Próba odczytu danych z pamięci podręcznej

Jeżeli dane zostały odnalezione w pamięci podręcznej, automatycznie zostają zwrócone a dalsze wykonywanie metody zostaje przerwane. W sytuacji gdy dane nie zostały jeszcze zbuforowane, następuje ich odczyt z oryginalnego – wolniejszego – źródła danych oraz zapis w pamięci podręcznej:

```
if (responseMessage.StatusCode == System.Net.HttpStatusCode.OK)
{
    var content = await responseMessage.Content.ReadAsStringAsync();
    var result = JsonConvert.DeserializeObject<ResponseListModel<T>>(content);

    if (result.Result is { })
        await _cacheService.SetCacheValueAsync(cacheKey, result);

    return result.Result;
}
```

Rysunek 38. Zbuforowanie odczytanych danych

Dane następnie są zwracane. Tym samym przy kolejnym zapytaniu o te same dane schemat zostanie powtórzony jednak tym razem zostaną zwrócone zbuforowane dane.

Stosując technikę buforowania powinniśmy pamiętać o kilku zasadach. Nie staramy się buforować wszystkiego. Buforujemy tylko dane odczytywane z dużą częstotliwością ale stosunkowo rzadko modyfikowane. Zazwyczaj są to dane słownikowe. Powinniśmy starannie dobierać czasy tak zwanej inwalidacji cachu. Chodzi tutaj o czas po którym zbuforowane dane tracą swoją ważność i wymagane jest odświeżenie bufora. To zapobiega sytuacji gdy posługujemy się danymi które nie są już aktualne. Zazwyczaj ustawiamy w tym celu dwie zmienne: *sliding expiration* – czas pomiędzy odczytamy danych z bufora. Jeżeli w danym przedziale czasu zbuforowane dane nie zostaną wykorzystane – bufor traci ważność. Natomiast zmianna *absolute expiration* definiuje bezwzględny termin ważności zbuforowanych danych.

Chodzi o to, że w przypadku bardzo częstych odczytów danych z bufora, *sliding expiration* może nigdy nie zostać przekroczona co będzie skutkowało korzystaniem ze zdezaktualizowanych danych. Dlatego należy ustawić wartość bezwzględną, po której bufor i tak musi zostać odświeżony.

4.2.9.4 Zwiększanie odporności aplikacji przy pomocy mechanizmów ponawiania

Kolejną bolączką intensywnie używanego serwisu może być jego niepełna dostępność. Chodzi o sytuacje gdy wysyłamy żądanie do serwisu zewnętrznego i nie otrzymujemy od niego odpowiedzi. Czasami serwis zewnętrzny od razu zwraca komunikat o błędzie. Zazwyczaj mamy do czynienia z błędami przejściowymi. W takiej sytuacji warto zastanowić się nad mechanizmem ponowienia zapytanie do serwisu. Przeciwnym razie daną sytuację trzeba będzie i tak w jakiś sposób obsłużyć – zazwyczaj poprzez pokazanie komunikatu o błędzie. Z reguły jednak użytkownik danej aplikacji i tak ręcznie ponowi dane działanie.

Możemy pokusić się o samodzielne napisanie własnej obsługi takich błędów. Można by do tego wykorzystać algorytm *exponential backoff*, którego idea polega na tym, że po kilku błędach losowana jest określona liczba z zadanego przedziału, która to stanowi czas po którym nastąpi ponowienie wysłania żądania.

Mamy jednak dostępne gotowe biblioteki. Jedną z nich jest *Polly*. Dzięki niej możemy zaimplementować dwa scenariusze

```
services.AddHttpClient("shipmentApiClient")
    .AddTransientHttpErrorPolicy(policy => policy.WaitAndRetryAsync(3, _ => TimeSpan.FromSeconds(2)))
    .AddTransientHttpErrorPolicy(policy => policy.CircuitBreakerAsync(5, TimeSpan.FromSeconds(5)));
```

Rysunek 39. Implementacja reguł ponawiania wywołań

Zgodnie z powyższym rysunkiem w projekcie użyłem dwóch polis, reguł: pierwsza z nich pozwala na automatyczne trzykrotne ponowienie żądania po odczekaniu dwóch sekund. Druga polisa implementuje wzorzec *Circuit Breaker*. Jego zasada działania – jak wskazuje nazwa – polega na przerwaniu obwodu. Innymi słowy jeżeli wykonane operacje ponowień nie przyniosą skutku to zostanie przerwany obwód, czyli nie będziemy miało miejsca ponowienie żądania tylko automatycznie zostanie wyświetlony błąd o chwilowej niedostępności danego serwisu. Taki błąd z reguły ustawia się na kilka sekund. Po upływie tego czasu możliwe będzie ponowne wysłanie żądania pod wskazany adres. Wszystkie te zabiegi mają na celu zwiększenie komfortu pracy aplikacji klienckiej.

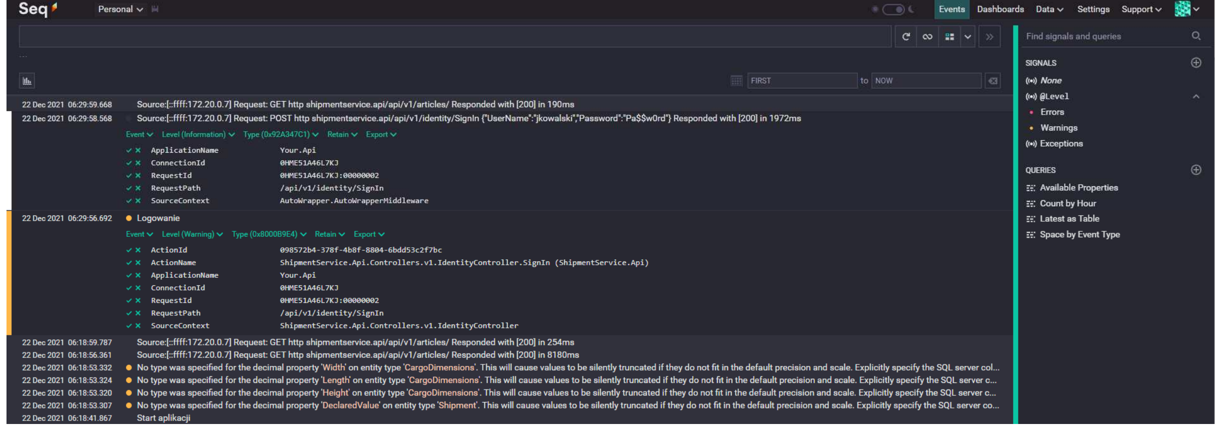
4.3 Zastosowanie dodatkowych narzędzi oraz bibliotek w projekcie

Oprócz omówionych powyżej narzędzi zastosowałem jeszcze kilka innych, których zastosowanie warte jest odnotowania:

4.3.1 Seq oraz Serilog

Dwie pierwsze należy opisać w parze gdyż ich zastosowanie jest sprowadza się do obsługi tego samego zakresu funkcjonalnego aplikacji a mianowicie logowania. Jak wspominałem wcześniej Serilog to biblioteka rozszerzająca możliwości logowania standardowego interfejsu wbudowanego we framework. Dzięki niej mamy możliwość logowania strukturalnego czyli możemy zalogować całe obiekty. Jest to nieocenione w przypadku obsługi logowania w serwisach web api gdzie często zachodzi potrzeba zalogowania całego żądania jakie wysyła do nas klient. Użycie biblioteki jest w zasadzie bezobsługowe gdyż implementuje ona standardowy interfejs frameworka – *Ilogger*.

Natomiast *Seq* to narzędzie do wizualizacji naszych logów. Zazwyczaj logi zapisywane do pliku, który w razie potrzeby jest przeszukiwany ręcznie bądź przy pomocy przeznaczonych do tego narzędzi. Użycie biblioteki *Serilog* daje nam możliwość definiowania tak zwanych *Sinks* czyli miejsc do których będą spływały logowane dane. Jednym z nich może być właśnie *Seq*. Z perspektywy użytkownika *Seq* to strona internetowa przy pomocy której możemy śledzić logi naszej aplikacji. Możemy również w odpowiedni dla nas sposób filtrować, itd… Poniższy rysunek pokazuje okno aplikacji *Seq* śledzące działanie naszej aplikacji – gdzie odnotowana jest próba zalogowania się do serwisu web api:



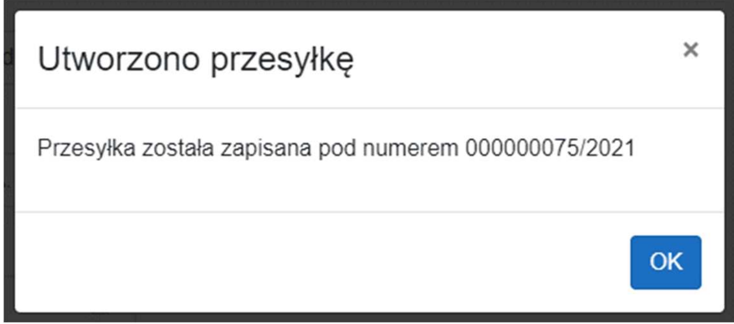
Rysunek 40. Okno aplikacji logującej Seq

4.3.2 RabbitMq

RabbitMq to narzędzie które służy do obsługi komunikatów w zawartych w postaci kolejek. Cała idea działania opiera się na udziale w procesie wymiany trzech ról: producenta komunikatu, serwera obsługującego kolejki z komunikatami oraz konsumentów komunikatów. Zadaniem producenta jest utworzenie oraz wysłanie do kolejki istotnych z punktu widzenia całości rozwiązania informacji. Zadaniem serwera jest przyjęcie tego komunikatu i przechowywanie go w określonej kolejce. Natomiast zadaniem konsumenta jest zasubskrybowanie się do odpowiedniej kolejki w celu oczekiwania na interesujące go wiadomości.

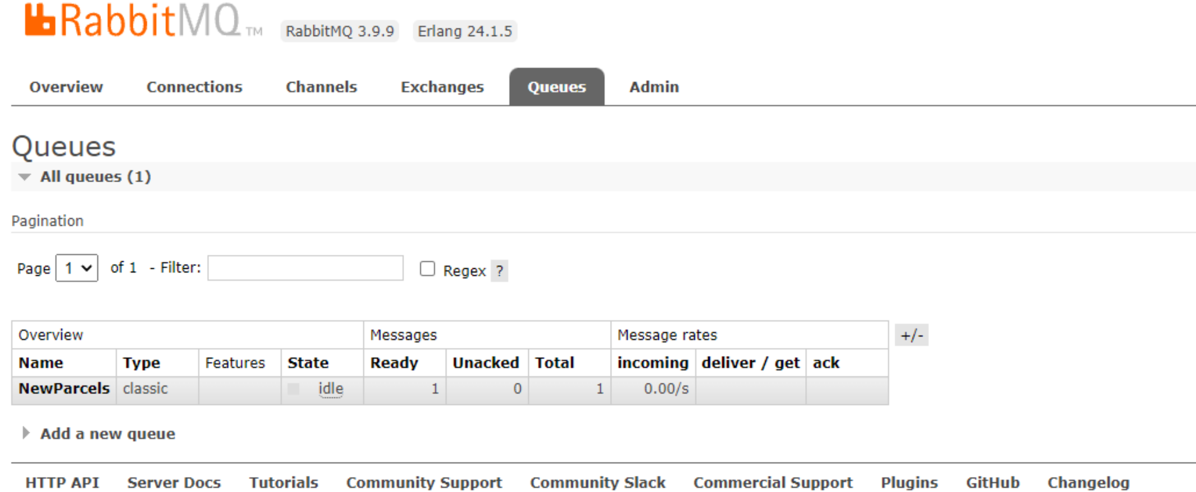
Narzędzie to doskonale sprawdza się w systemach rozproszonych gdzie możliwe czy wręcz wymagane jest zastosowanie tak zwanej asynchroniczności. Chodzi o to, że jedynym zadaniem producenta jest wygenerowanie odpowiedniej wiadomości. Producenta nie interesuje dalszy los wiadomości oraz to czy zostanie ona przetworzona albo wręcz czy ktokolwiek z niej skorzysta. Zadaniem subskrybenta jest zatroszczenie się o zdobycie odpowiedniej informacji.

W przypadku omawianego projektu zastosowałem bardzo banalne rozwiązanie polegające na tym, że w momencie utworzenia przesyłki zostaje opublikowany na kolejkę komunikat z jej numerem. Jako, że w chwili obecnej nie mamy wiedzy z jakich systemów korzysta nasz potencjalny klient po zakupie naszego oprogramowania będzie miał możliwość zasubskrybowania się do danej kolejki i dalsze przetwarzanie informacji o nowej przesyłce, która pojawiła się w systemie. Poniżej utworzyłem na stronie WWW nową przesyłkę:

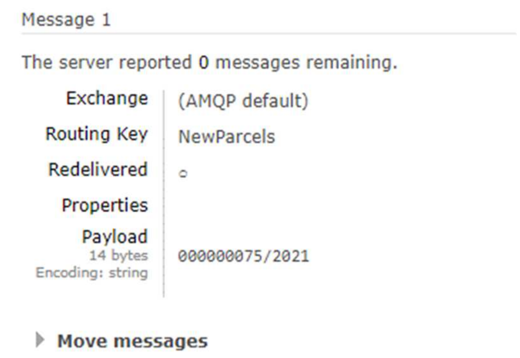


Rysunek 41. Popup na stronie informujący o nadaniu numeru zatwierdzonej przez nas przesyłce

Po jej zatwierdzeniu został wygenerowany nowy komunikat i wysłany na odpowiednią kolejkę do RabbitMq:



Rysunek 42. Dashboard RabbitMQ pokazujący kolejkę NewParcels z nową wiadomością



Rysunek 43. Wiadomość z numerem nowo utworzonej przesyłki

Z punktu widzenia projektu wiadomości została opublikowana w momencie obsługi umawianych wyżej zdarzeń domenowych. Takim zdarzeniem było zatwierdzenie przesyłki. W efekcie jego wystąpienia została wywołana odpowiednia metoda interfejsu, której implementację utworzyłem – zgodnie z zasadami czystej architektury – w projekcie infrastruktury.

4.3.3 Docker

Zastosowanie powyższych narzędzi było by trudne z punktu widzenia wdrożenia aplikacji na serwerach klienta. Do prawidłowego i w pełni funkcjonalnego działania projekt potrzebuje zapewnienia sobie takich narzędzi jak RabbitMq czy Seq. Te rozwiązania wymagają instalacji oraz hostowania na serwerze WWW co mogłoby znacznie podnieść koszty wdrożenia aplikacji. Z pomocą przyszło narzędzie Docker.

Jest to narzędzie które pozwala na utworzenie dodatkowego w pełni odizolowanego systemu operacyjnego. Zaletą Dockera jest to, że wykorzystuje on system operacyjny hosta, czyli maszyny na której jest zainstalowany. To daje znaczne korzyści wydajnościowe gdyż – w przeciwieństwie do tradycyjnych rozwiązań do wirtualizacji systemów – nie tworzy wszystkiego całkowicie od nowa.

Standardowo kontenery działają w całkowitym odseparowaniu od siebie. Oczywiście do momentu gdy sobie tego życzymy. W przypadku omawianego rozwiązania taka sytuacja byłaby niedopuszczalna.

Kontenery działają na podstawie obrazów. Są to w pewnym sensie szablony czystych systemów operacyjnych z zazwyczaj zainstalowaną na nich jedną interesującą nas aplikacją. Do uruchomienia kontenera potrzebna jest aplikacja Docker, która dostarcza nam interfejsc w postaci poleceń konsoli dzięki którym możliwe jest uruchomianie kontenerów.

Drugim, znacznie wygodniejszym sposobem uruchomiania kontenerów są pliki *Dockerfile*, dzięki którym mamy możliwość zdefiniowania pewnych stałych ustawień naszego kontenera. Poniższej przedstawiam plik Dockerfile dla projektu webapi:

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["ShipmentUI/ShipmentUI.csproj", "ShipmentUI/"]
RUN dotnet restore "ShipmentUI/ShipmentUI.csproj"
COPY . .
WORKDIR "/src/ShipmentUI"
RUN dotnet build "ShipmentUI.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "ShipmentUI.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ShipmentUI.dll"]
```

Rysunek 44. Plik Dockerfile dla projektu ShipmentService.Api

W przypadku tego pliku zastosowałem mechanizm tzw. *multistage builds.* Chodzi o to, że wykorzystuję w nim dwa obrazy dockera. Jeden to obraz *SDK* do którego kopiuję pliki źródłowe projektu a następnie wykonuję polecenie *dotnet build*. Po zbudowaniu projektu kopiuję utworzone binaria do nowego obrazu – tym razem do czystego środowiska wykonawczego dotnet. To pozwala mi na znaczne zaoszczędzenie miejsca, gdyż obraz *SDK* z racji zawartych w nim bibliotek i narzędzi do budowania projektów jest znacznie większy. Do prawidłowego działania aplikacji te rzeczy są nam niepotrzebne.

Drugim istotnym z punktu widzenia projektu plikiem jest plik *docker-compose*. Jest to plik, który pozwala na pewną orkiestrację zawartych w projekcie obrazów Docker. W pliku tym konfigurujemy między innymi porty, które będą wykorzystywały poszczególne kontenery:

```
version: '3.4'

services:
  db:
    image: "mcr.microsoft.com/mssql/server"
    ports:
      - "5433:1433"
    volumes:
      - C:\TEMP\DATABASES:/var/opt/mssql/data
    environment:
      SA_PASSWORD: "Your_password123"
      ACCEPT_EULA: "Y"
  shipmentservice.api:
    image: ${DOCKER_REGISTRY-}shipmentserviceapi
    build:
      context: .
      dockerfile: ShipmentService.Api/Dockerfile
    ports:
      - "7000:80"
    depends_on:
      - db

  shipmentui:
    image: ${DOCKER_REGISTRY-}shipmentui
    build:
      context: .
      dockerfile: ShipmentUI/Dockerfile
    ports:
      - "8800:80"
    depends_on:
      - shipmentservice.api
  redis:
    image: redis
    restart: always
    ports:
      - "6379:6379"
  rabbitmq:
    image: rabbitmq:3-management-alpine
    container_name: 'rabbitmq'
    ports:
        - 5672:5672
        - 15672:15672
  seq:
    image: datalust/seq:latest
    ports:
      - 8099:80
      - 5341:5341
    environment:
      ACCEPT_EULA: Y
```

Rysunek 45. Plik dockerfile orkiestrujący działanie całej solucji

Za przykład niech posłuży pierwszy zdefiniowany na powyższym rysunku kontener – baza danych. Nadałem kontenerowi nazwę ***db***, która stanowi w pełni funkcjonalną nazwę pod kątem komunikacji z innymi kontenerami. Dowodem tego jest wartość *connection string* użyta do połączenia z bazą danych w projekcie web api:

```
"ConnectionStrings": {
  "ShipmentDbConnectionString": "Server=db;Database=shipment;User=sa;Password=Your_password123;"
},
```

Rysunek 46. Connection string odwołujący się do skonteneryzowanej bazy danych

W sekcji *volumes* definiujemy zasoby na komputerze hosta do których będzie miał dostęp kontener. W tym przypadku są to pliki bazy danych. O ile sama baza danych działa w całkowicie odizolowanym kontenerze, tak pliki muszą być zapisywane na maszynie hosta gdyż w przeciwnym wypadku, po restarcie kontenera uległy by nadpisaniu nową wersją plików.

W sekcji *ports* definiujemy porty które dostępne będą z zewnątrz kontenera. W tym konkretnym przypadku port 5433 na maszynie hosta będzie przekierowywany na port 1433 kontenera. Jak wiadomo port 1433 to standardowy port serwera SQL. W efekcie uzyskujemy dostęp do danych z zewnątrz kontenera.

4.4 Testy jednostkowe

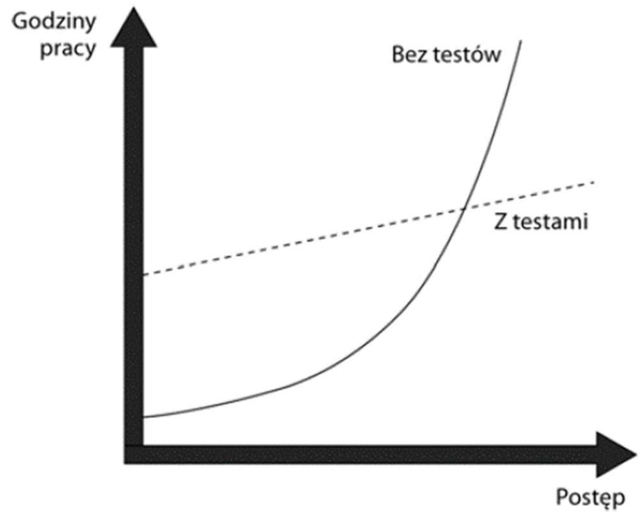
Naprawdę trudno wyobrazić sobie stworzenie obecnie aplikacji działającej od pierwszych chwil jej istnienia bez zarzutu. Poziom skomplikowania współczesnej inżynierii oprogramowania powoduje, że praktycznie każda aplikacja wymaga rzetelnego jej przetestowania przed oddaniem do jakiegokolwiek użytku.

Jednym z najbardziej skutecznych a zarazem najbardziej popularnych narzędzi są testy jednostkowe. Mianem testu jednostkowego możemy określić „fragment kodu, który wywołuje jednostkę pracy i sprawdza końcowy wynik tej jednostki pracy”.[10]

Jedną z zasadniczych zalet testów jednostkowych – oprócz wykrywania błędów – jest poprawa jakości produkowanego kodu. Często zdarza się, że pisząc kod jednostkowy dla danego fragmentu oprogramowania okazuje się, że napisanie testu jest niemożliwe. Jest to jasna wskazówka że kod, który próbujemy przetestować jest słabej jakości i wymaga natychmiastowej refaktoryzacji.

[10] Roy Osherove, Testy jednostkowe – świat niezawodnych aplikacji, Helion, Gliwice 2014, s. 29.

Testy jednostkowej przyczyniają się również do zwiększenia wydajności wytwarzania oprogramowania. To może wydawać dziwne, ale poniższy wykres uświadamia nam pewną zależność. Jak można było się spodziewać oprogramowanie powstaje znacznie szybciej bez konieczności pisania testów jednostkowych jednak po przekroczeniu pewnej granicy skomplikowania kodu szybkość ta drastycznie maleje na korzyść testów. Dlaczego ? Otóż przyczyna jest banalnie prosta – brak testów jednostkowych powoduje, że w kodzie pojawia się coraz więcej wad ukrytych, które wraz z przyrostem nowego kodu zaczynają się ujawniać i wymagają bieżącej naprawy. Zjawisko to nosi miano *entropii oprogramowania*[11]



Rysunek 47. Porównanie dynamiki wytwarzania oprogramowania z zastosowaniem testów jednostkowych i bez

Testy powinny cechować się kilkoma właściwościami: powinny być zautomatyzowane, można je włączać dowolną ilość razy, powinny działać szybko oraz powinny działać w izolacji.

Ostatnia cecha wymaga krótkiego omówienia. Otóż większość klas, metod składa się z różnych zależności. Zazwyczaj mają one sporo referencji do innych klas, metod, itd… testując daną jednostkę – na przykład metodę – testujemy tylko jej działanie. Reszta zależności nie powinna mieć wpływu na wyniki testu. Tak więc wszystkie zależności powinny zostać zastąpione tak zwanymi atrapami bądź zaślepkami (obiekty *mock*). Dlatego tak istotne jest pisanie dobrego kodu umożliwiającego w ogóle przetestowanie danej funkcjonalności. Główną zasadą jest luźne wiązanie – powinniśmy jak najwięcej kodu opierać o interfejsy bądź abstrakcje.

[11] Vladimir Khorikov, Testy jednostkowe. Zasady, praktyki, wzorce, Helion, Gliwice 2020, s. 22.

Zazwyczaj struktura testu oparta jest na szablonie *AAA*, czyli *Arrange, Act, Assert*. Są to przyjęte ogólnie nazwy faz występujących w testowaniu. Pierwsza opisuje fragment przygotowujący do testu – jest to zazwyczaj deklaracja zmiennych, klas, strap, itd… Druga faza – *Act* – opisuje proces przeprowadzenia danej akcji, której wyniki będziemy oceniać w fazie *Assert*.

Jest to oczywiście ogólnie przyjęta koncepcja a nie sztywno narzucona norma. Zwłaszcza, że w niektórych przypadkach jej wykorzystanie może być niemożliwe. Za przykład niech posłuży jeden z przypadków testowych z projektu:

```
[Fact]
0 references
public void Passing_wrong_zipcode_throws_exception()
{
    //Arrange
    Func<ZipCode> sut = () => new ZipCode("1212345");

    //Act & Assert
    sut.Should().Throw<InvalidZipCodeException>();
}
```

Rysunek 48. Test logiki kontrolera

Jak widać test ten łączy w sobie fazy *Act* oraz *Assert*. Chodzi o to, ze w tym przypadku testuję działanie konstruktora klasy oczekując wrzucenia wyjątku w przypadku podania kodu pocztowego w złym formacie.

Pisząc testy jednostkowe projektu zdecydowałem się na framework *xUnit* oraz biblioteki *FluentAssertions*. Uważam, że narzędzia te zwiększają czytelność testów i na podstawie własnych doświadczeń wiem, że bywają zrozumiałe nawet dla osób spoza kręgu programistów. Podczas pisania testów starałem się przestrzegać zasady testowania pojedynczej jednostki a wręcz pojedynczego scenariusza.

Przykładowo tworzenie nowego ładunku wymaga podania kilku różnych prawidłowych danych. Tworząc testy jednostkowe postanowiłem przetestować każdy scenariusz z osobna. Wynikiem tego jest pokazany na poniższym rysunku jeden z testów napisanych dla procesu tworzenia ładunku a mianowicie test wykrywający czy podanie pustego opisu ładunku spowoduje wyrzucenie określonego typu wyjątku. W tym przypadku zamiast standardowego dla frameworka atrybutu *Fact* pojawił się atrybut *Theory*, który wskazuje że przekazujemy do testu kilka różnych danych testowych. W tym przypadku przekazujemy obiekt *null* oraz pusty string gdyż obydwa te przypadki powinny zostać wykryte jako błędne.

```
[Theory]
[InlineData(null)]
[InlineData("")]
0 references
public void Passing_empty_description_value_throws_ArgumentNullException(string description)
{
    //Arrange
    Func<Cargo> sut = () => Cargo.Builder()
    .WithLength(1)
    .WithWidth(1)
    .WithHeight(1)
    .WithDescription(string.Empty)
    .WithPackageQuantity(1)
    .WithPackageType("CLL")
    .Build();

    //Act + assert
    sut.Should().Throw<ArgumentException>();
}
```

Rysunek 49. Test sprawdzający wykrycie podania pustego opisu ładunku

Pisząc testy przestrzegałem konwencji polegającej na tworzeniu nazw folderów oraz plików odpowiadającym nazwom obiektów testowanych tak aby było łatwe do określenia co jest przedmiotem testu. Ponadto starałem się unikać w testach wykorzystywania nazw testowanych metod. Powód jest prosty – w przypadku refaktoryzacji kodu głównego może nastąpić konieczność zmiany nazw niektórych testów co może być uciążliwe i prowadzić do błędów. Uważam, że nazwy testów powinny odzwierciedlać testowane działanie a nie nazwy obiektów które to działanie definiują.

2022

5 Podsumowanie

Głównym celem pracy była prezentacja dwóch koncepcji związanych z architekturą programowania – *Clean Architecture* oraz *Domain-Driven Design.* Drugim – obocznym – celem było stworzenie działającej, w pełni funkcjonalnej aplikacji umożliwiającej rejestrowanie przesyłek kurierskich. Jak wiemy branża e-commerce rośnie w siłę. Jest to jeden z niewielu obszarów gospodarki raczej odporny na zawirowania gospodarcze, geopolityczne czy tak nietypowe to z którym zmagamy się obecnie – pandemiczne. W związku z czym również rynek transportowy a w szczególności jego gałąź – dostawy kurierskich – przeżywa rozkwit. To pociąga za sobą konieczność ciągłego dostarczania szybkiego i niezawodnego oprogramowania wspomagającego obydwie branże. Uważam, że bez dobrze zaplanowanej architektury tworzenie takich rozwiązań staje się bardzo trudne lub wręcz całkowicie niemożliwe. Na przestrzeni powyższych rozdziałów starałem się zawrzeć skondensowaną dawkę wiedzy dotyczącą architektury oprogramowania oraz możliwości użycia narzędzi wykraczających poza standardowe rozwiązania zawarte w omawianym frameworku programowania co pozwala mi stwierdzić, że obydwa cele pracy zostały osiągnięte.

6 Bibliografia

Evans, E. (2003). *Domain-Driven Design: Tackling Complecity in the Heart of Software.* Boston: Addison Wesley.

Khorikov, V. (2020). *Testy jednostkowe. Zasady, praktyki, wzorce.* Gliwice: Helion.

Martin, R. C. (2018). *Czysta Architektura.* Gliwice: Helion.

Meyer, B. (1988). *Object-Oriented Software Construction.* Hoboken: Prentice Hall.

Osherove, R. (2014). *Testy jednostkowe – świat niezawodnych aplikacji.* Gliwice: Helion.

Smith, S. (2021, 11 30). *Ardalis - become a better developer with me.* Pobrano z lokalizacji https://ardalis.com/clean-architecture-asp-net-core/

Sobótka, S. (2012, Lipiec 20). *Bottega IT minds.* Pobrano z lokalizacji https://bottega.com.pl/pdf/materialy/ddd/ddd1.pdf

Vernon, V. (2018). *DDD. Kompendium wiedzy.* Gliwice: Helion.

Zimarev, A. (2021). *Domain-Driven Design dla .Net Core.* Gliwice: Helion.

Spis ilustracji

Rysunek 1. Ekran startowy aplikacji 9
Rysunek 2. Rejestracja użytkownika 10
Rysunek 3. Wiadomość potwierdzająca rejestrację 11
Rysunek 4. Menu użytkownika 11
Rysunek 5. Menu administratora 12
Rysunek 6. Widok szczegółów przesyłki 13
Rysunek 7. Wydruk listu przewozowego 13
Rysunek 8. Tworzenie nowej przesyłki 14
Rysunek 9. Edycja ładunków przesyłki 14
Rysunek 10. Komunikat o utworzeniu przesyłki 15
Rysunek 11. Redagowanie nowej wiadomości 16
Rysunek 12 Podział warstw zgodnie z zasadami czystej architektury 17
Rysunek 13 Fragment kodu w standardzie C4 19
Rysunek 14 Diagram klas 20
Rysunek 15 Diagram sekwencji komponentów aplikacji 21
Rysunek 16 Diagram sekwencji procesu logowania 22
Rysunek 17. Standardowa struktura akcji kontrolera 22
Rysunek 18. Autoryzacja dostępu do metod kontrolera przy użyciu polis 23
Rysunek 19. Obsługa żądania http 24
Rysunek 20. Widok strony dokumentacyjnej api wygenerowanej przez narzędzie Swagger . 24
Rysunek 21. Struktura folderów z zastosowaniem CQRS 26
Rysunek 22. Komenda oraz jej handler realizujące tworzenie nowego rodzaju opakowania . 27
Rysunek 23. Implementacja własnego interfejsu dostarczającego daty 28
Rysunek 24. Implementacja konfiguracji encji w EntityFamework 29
Rysunek 25.Własna implementacja metody SaveChangesAsync 30
Rysunek 26. Interfejs wzorca repozytorium 31
Rysunek 27. Przykład implementacji Value Object 34
Rysunek 28. Przykład typowej encji wraz z jej właściwościami i konstruktorami 35
Rysunek 29. Przykład sprawdzenia poprawności ilości opakowań 36
Rysunek 30. Przykład tworzenia obiekty przy pomocy fluent interface 36
Rysunek 31. Udostępnienie list tylko do odczytu 37
Rysunek 32. Przykład reguły biznesowej 38

2022

Rysunek 33. Sprawdzanie reguł biznesowych 38
Rysunek 34. Bazowa klasa wyjątku 39
Rysunek 35. Wywołanie metody C# podczas obsługi naciśnięcia przycisku 40
Rysunek 36.Zapisanie w pamięci przeglądarki tokena uzyskanego w wyniku poprawnego zalogowania się 41
Rysunek 37. Próba odczytu danych z pamięci podręcznej 42
Rysunek 38. Zbuforowanie odczytanych danych.. 42
Rysunek 39. Implementacja reguł ponawiania wywołań.. 43
Rysunek 40. Okno aplikacji logującej Seq.. 44
Rysunek 41. Popup na stronie informujący o nadaniu numeru zatwierdzonej przez nas przesyłce 45
Rysunek 42. Dashboard RabbitMQ pokazujący kolejkę NewParcels z nową wiadomością... 46
Rysunek 43. Wiadomość z numerem nowo utworzonej przesyłki 46
Rysunek 44. Plik Dockerfile dla projektu ShipmentService.Api 47
Rysunek 45. Plik dockerfile orkiestrujący działanie całej solucji 48
Rysunek 46. Connection string odwołujący się do skonteneryzowanej bazy danych 49
Rysunek 47. Porównanie dynamiki wytwarzania oprogramowania z zastosowaniem testów jednostkowych i bez 50
Rysunek 48. Test logiki kontrolera 51
Rysunek 49. Test sprawdzający wykrycie podania pustego opisu ładunku............................ 52