POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Karol Gzik
Nr albumu s23860

Wykorzystanie podejścia low-code w generowaniu sieci mikroserwisów

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, czerwiec 2022

Streszczenie

Celem pracy jest zaprezentowanie alternatywnego rozwiązania powtarzalnego tworzenia mikroserwisów. Praca porusza problem powtarzalnej, praktycznej implementacji projektów bazujących na architekturze mikroserwisów. Dotychczasowe rozwiązanie tego problemu polega na rozpoczęciu całości od początku samodzielnie lub skorzystanie ze zbyt ogólnych dostępnych generatorów. Rosnące zainteresowanie architekturą tego typu doprowadziło do powtarzalności i podatności na błędy ludzkie.

W dalszej części pracy przedstawione zostały konkurencyjne pomysły na wspomaganie programisty w *scaffoldingu* aplikacji. Generatory projektów wbudowane w środowiska programistyczne lub pakiety kompilacyjne oferują ograniczone możliwości konfiguracji generowanych projektów, a także pomijają wymogi dotyczące tego konceptu architektonicznego.

Przedstawione podejście low-code może rozwiązać złożoność tego problemu oraz uprościć definiowanie konfiguracji sposobu integracji mikroserwisów między sobą. W tym celu utworzono prototyp, który oferuje utworzenie projektów, połączenie siecią wewnętrzną oraz utworzenie obrazów kontenerów. Jako wewnętrzną notację konfiguracji prototypu generatora wykorzystano format JSON. W przyszłości notacja ta może być rozwinięta na obsługę edytora wizualnego co ułatwi definiowanie intencji użytkownika.

Dokonano analizy oraz porównania utworzonego prototypu wraz z konkurencyjnym rozwiązaniem JHipster. Analiza zawiera informacje o wielkości budowanych aplikacji, dostępnych funkcjonalnościach oraz możliwościach przyszłego rozwoju.

Zwieńczeniem pracy jest prezentacja ogólnych problemów każdego rodzaju aplikacji generujących projekty. Opisane zostały także problemy architektury mikroserwisów oraz problemy w generowaniu tego typu aplikacji.

Słowa kluczowe: mikroserwis, automatyzacja, parsowanie, low-code, C#, .NET, Python, SQLite, HTTP, REST, API, OpenAPI, Swagger, JSON, Docker, Docker Compose, konteneryzacja

Podziękowania

Składam serdeczne podziękowania promotorowi dr Mariuszowi Trzaska za wsparcie techniczne i merytoryczne. Anecie Kohnke która mimo braku znajomości dziedziny pomogła w stylistyce pracy.

Spis treści

1. Wstęp

Zwiększające się zapotrzebowanie na systemy informatyczne oraz rosnące wymagania logiki biznesowej powodują zwiększone zainteresowanie firm informatycznych zmniejszeniem nakładu pracy przy tworzeniu projektów od podstaw. *Scaffolding* jest narzędziem, które generuje aplikacje lub jej komponenty według zadanych przez programistę poleceń. Narzędzie to pomaga w powyższym wymaganiu poprzez automatyzację wytwarzania aplikacji wraz z podstawowymi komponentami potrzebnymi do jej działania. *Scaffolding* pozwala na odciążenie pracy programisty oraz uniknięcie błędu ludzkiego powtarzalnych zadań. Generowane aplikacje są zazwyczaj ograniczone pod względem dostępnych opcji, jak i samej struktury kodu. Tworzone aplikacje z tego względu pozwalają na wszechstronność potencjalnego przyszłego rozwoju.

Aktualnie większość dostępnych popularnych języków posiada wbudowane narzędzie do generowania podstawowej wersji aplikacji bądź ich części związanej ze specyficzną domeną. Przykładem takiego narzędzia jest generator encji w bazie danych na podstawie zaimplementowanego modelu wewnątrz aplikacji. Oznacza to, że twórca aplikacji powinien wybrać narzędzia dostępne w swoim języku programowania, które jest dostosowane do jego wymagań.

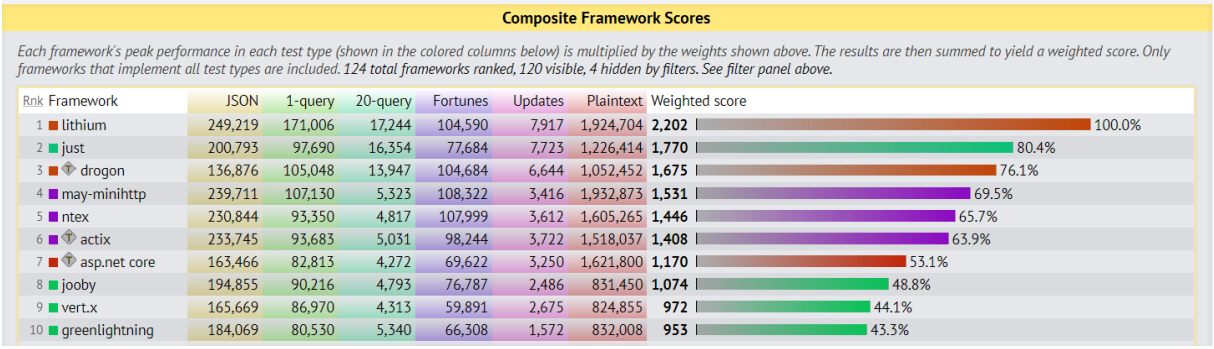
Niniejsza praca prezentuje rozwiązanie problemu ogólności narzędzi *scaffoldingu* poprzez ukierunkowanie się na mikroserwisy. Stworzony prototyp działa na wyższej warstwie logicznej. Tworzona jest sieć połączonych projektów, modeli, struktur baz danych oraz warstwa wymiany komunikatów.

1.1. Cel pracy

Problemem większości generatorów jest ukierunkowanie na tworzenie aplikacji ogólnego przeznaczenia. W przypadku architektury mikroserwisów poza podstawową aplikacją zazwyczaj wymagana jest baza danych oraz warstwa komunikacji z innymi usługami.

Celem niniejszej pracy jest zaprezentowanie konceptu *low-code* i *scaffoldingu* w zakresie architektonicznym, jakim są mikroserwisy. Prototyp generatora został przygotowany w jednym z popularniejszych języków ogólnego przeznaczenia jakim jest Python. Wybór ten był podyktowany prostotą języka i łatwością przenaszalności napisanej implementacji na inne platformy. Dodatkowo sam język pozwala na łatwe dopisywanie obsługi dodatkowych języków i funkcjonalności w tworzonych aplikacjach wyjściowych.

Projekty wyjściowe generowane są w języku C# na platformie .NET. Wybór ten był podyktowany umiejętnościami i doświadczeniem autora w tym zakresie. .NET jest nowoczesną oraz popularną platformą programistyczną rozwijaną przez społeczność *open-source* przy wsparciu Microsoftu. Aplikacje stworzone w tej platformie wymagają małej ilości przestrzeni dyskowej i pamięciowej, dodatkowo są bardzo wydajne, co zostało zaprezentowane na rysunku 1.



Composite Framework Scores

Each framework's peak performance in each test type (shown in the colored columns below) is multiplied by the weights shown above. The results are then summed to yield a weighted score. Only frameworks that implement all test types are included. 124 total frameworks ranked, 120 visible, 4 hidden by filters. See filter panel above.

Rnk	Framework	JSON	1-query	20-query	Fortunes	Updates	Plaintext	Weighted score
1	lithium	249,219	171,006	17,244	104,590	7,917	1,924,704	**2,202** 100.0%
2	just	200,793	97,690	16,354	77,684	7,723	1,226,414	**1,770** 80.4%
3	drogon	136,876	105,048	13,947	104,684	6,644	1,052,452	**1,675** 76.1%
4	may-minihttp	239,711	107,130	5,323	108,322	3,416	1,932,873	**1,531** 69.5%
5	ntex	230,844	93,350	4,817	107,999	3,612	1,605,265	**1,446** 65.7%
6	actix	233,745	93,683	5,031	98,244	3,722	1,518,037	**1,408** 63.9%
7	asp.net core	163,466	82,813	4,272	69,622	3,250	1,621,800	**1,170** 53.1%
8	jooby	194,855	90,216	4,793	76,787	2,486	831,450	**1,074** 48.8%
9	vert.x	165,669	86,970	4,313	59,891	2,675	824,855	**972** 44.1%
10	greenlightning	184,069	80,530	5,340	66,308	1,572	832,008	**953** 43.3%

Rysunek 1. Zestawienie wydajności poszczególnych operacji w przykładowych platformach webowych – źródło: [1]

Tworzenie aplikacji lub całej infrastruktury mikroserwisów od początku jest zadaniem wymagającym wielu umiejętności z zakresu programowania i architektury oprogramowania. Rosnące koszty systemów informatycznych zmuszają do optymalizacji powtarzalnych zadań. Prawie w każdym przypadku twórcy systemów internetowych powinni szukać ułatwienia budowy projektu, aby ograniczyć czas implementacji i wytworzenia dobrej podstawy do dalszego rozwoju. W przypadku zdecydowania się na architekturę mikroserwisów tworzenie podstawy jest powtarzalną operacją. Wybrany język, system bazodanowy i sposób komunikacji pozostają identyczne w każdym projekcie. Zjawisko to pokazuje potrzebę tworzenia pewnego rodzaju wzorca projektu podstawowego z predefiniowaną konfiguracją.

Rozwiązaniem powyższego problemu są *scaffoldingi*, często spotykane narzędzia dostępne w kompilatorach, środowiskach programistycznych lub jako osobne aplikacje. Aktualnie dostępne opcje *scaffoldingu* pozwalają na spełnienie tylko części wymagań tworzenia infrastruktury mikroserwisowej. Poza tym wymagane są operacje:

- tworzenia modeli bazy danych,
- tworzenia warstwy zapisu danych,
- uwierzytelniania oraz autoryzacji,
- tworzenia warstwy komunikacji między rozproszonymi aplikacjami,
- konteneryzacji.

Utworzony prototyp, który był celem tej pracy, rozwiązuje część powyższych problemów. Konfiguracja opisująca zależności między usługami jest odwzorowywana jako wiele aplikacji przechowujących dane połączone jedną siecią wewnętrzną.

Generator konkurencyjny JHipster także wspiera tworzenie infrastruktury mikroserwisów. Wykorzystano tą możliwość jako okazję do dokonania porównania. Porównano możliwość spełnienia założonego wymagania oraz pewne właściwości tworzonych projektów.

1.2. Rozwiązania przyjęte w pracy

Istnieje wiele sposobów tworzenia infrastruktury mikroserwisowej. Zdecydowano na generowanie systemu sklepu internetowego opartego o trzy mikroserwisy. Aplikacje muszą oferować pełny dostęp do API oraz mieć możliwość komunikacji między sobą. Muszą także posiadać one osobne niezależne modele. Domeny każdej z tworzonych aplikacji to:

- zarządzanie produktami,

- zarządzanie koszykiem klienta oraz możliwość zamiany koszyka w zamówienia,
- zarządzanie zamówieniami oraz tworzenie nowego zamówienia na podstawie koszyka

Te same wymagania zastosowano wobec konkurencyjnego narzędzia JHipster w celu porównania sposobu budowy projektu i możliwości przyszłego rozwoju.

1.3. Organizacja pracy

Niniejsza praca w rozdziale drugim prezentuje ogólne spojrzenie na istniejące narzędzia oferujące usługi generowania kodu.

Następny rozdział zawiera informacje o narzędziach wykorzystywanych podczas tworzenia prototypu generatora. W przypadku narzędzi mniej popularnych zaprezentowano przykłady działania.

Rozdział czwarty przybliża koncepcję low-code oraz dziedzinę działania architektury mikroserwisów. Koniec tego rozdziału został poświęcony wskazaniu wymaganych informacji od użytkownika w przypadku tworzenia aplikacji mikroserwisowych.

Rozdział następny zawiera informacje dotyczące przygotowanego prototypu. Zawarto wymagania, opisy funkcjonalności, konfigurację, proces generowania aplikacji oraz wynik końcowy.

W rozdziale szóstym zawarto porównanie właściwości projektów generowanych przez konkurencyjne narzędzie JHipster wobec tych oferowanych przez przygotowany prototyp.

Kolejne rozdziały zawierają ogólne spojrzenie na problem automatycznego tworzenia kodu oraz tworzenia mikroserwisów. Dokument kończy podsumowanie, które zawiera informacje na temat osiągniętych wyników, zasadności stosowania generatorów oraz braków funkcjonalnych utworzonego prototypu.

2. Istniejące generatory kodu

Poniższy rozdział ma na celu zaprezentowanie wybranych konkurencyjnych rozwiązań zajmujących się *scaffoldingiem*. Istnieje wiele narzędzi zajmujących się generowaniem kodu, które w przypadku mikroserwisów nie znajdą zastosowania. Działają one na jednym projekcie zamiast na całej sieci aplikacji rozproszonych.

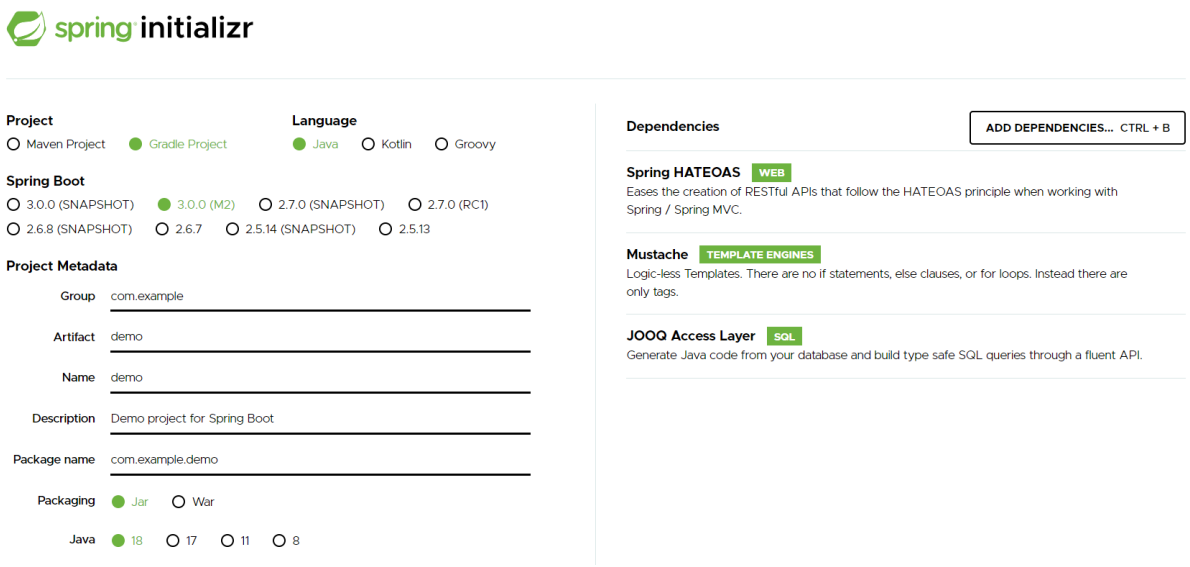
Rozdział został podzielony na charakterystykę działania generatorów. Wybrano głównie najpopularniejsze generatory dla każdego rodzaju charakterystyki. Część z nich jest powiązana ze środowiskami programistycznymi SDK (ang. *Software Development Kit*). Nowością *scaffoldingu* jest wykorzystywanie sztucznej inteligencji w celu wspomagania. Pomaga to programiście w przyspieszeniu tworzenia oprogramowania. Ostatnia grupa to generatory specjalistyczne, które są najbliższą konkurencją utworzonego prototypu.

2.1. Generowanie kodu przy użyciu środowiska programistycznego

Najpopularniejsze narzędzia to te, które są łatwo dostępne i zazwyczaj dostarczane z innymi narzędziami. W przypadku środowisk programistycznych SDK lub rozbudowanych edytorów plików źródłowych IDE (ang. *integrated development environment*) mogą być wbudowane lub dostępne jako dodatkowa opcja. Tego typu generatory zawsze muszą być najbardziej wszechstronne. Umożliwiają utworzenie bazowego projektu w celu dostosowania do własnych potrzeb.

2.1.1 Spring initializr

Spring initializr to dostarczany do pakietu Spring, generator pozwalający na utworzenie podstawowej aplikacji bez potrzeby dodatkowej konfiguracji. Wygenerowany projekt jest gotowy do uruchomienia, a także do opublikowania w chmurze. Podstawowa konfiguracja zależności i pakietów oferowanych w platformie Spring pozwala na dostosowanie aplikacji do potrzeb programisty i danej dziedziny. Dodatkowym atutem jest możliwość wybrania języka tworzonego projektu z puli języków rodziny Java Virtual Machine. Generator posiada wygodny interfejs co zostało zaprezentowane na rysunku 2. Ukryte metody obsługujące zapytania HTTP wewnątrz projektu mogą posłużyć do automatyzacji generowania projektów [2]. Projekt jest *open-source* i wraz z dostępną dokumentacją ma potencjał do dostosowania do własnych potrzeb.



Rysunek 2. Interfejs graficzny aplikacji Spring Initializr – źródło: opracowanie własne

Domyślna konfiguracja tworzy pusty projekt bez modeli i metod obsługujących zapytania. Wymaga to dużego nakładu pracy początkowej, aby móc wykorzystać go w wymaganej architekturze mikroserwisów. W celu tworzenia wielu aplikacji oprogramowanie wymagałoby dodatkowej automatyzacji wyżej wspomnianymi *endpointami* kontrolującymi.

2.1.2 .NET CLI

.NET CLI (ang. *command-line interface)* to wieloplatformowe narzędzie ogólnego przeznaczenia do wykonywania operacji związanych z platformą .NET. Pozwala na: tworzenie, budowanie, uruchamianie oraz publikowanie aplikacji. Wiele komend pozwala na konfigurację oraz na rozszerzanie narzędzia. Komenda `dotnet new` pozwala na *scaffoldowanie* aplikacji na podstawie wzorców przygotowanych przez społeczność. Nowe wzorce mogą być doinstalowane przez pakiety NuGet. Głównym atutem tego oprogramowania jest łatwa możliwość przygotowania własnego wzorca aplikacji. Własny wzorzec projektu może być przygotowany z gotowego projektu, a następnie opublikowany w sieci jako pakiet dla innych programistów [3]. Wzorce mogą posiadać dowolną ilość podprojektów oraz plików, które nie są związane z platformą .NET. Rodzaje aplikacji, które mogą być utworzone w standardowej konfiguracji narzędzia:

- Konsolowa,
- Web App oparta o system widoków Razor,
- Web API (ang. *Application Programming Interface*),
- Aplikacje hybrydowe z aplikacją interfejsu użytkownika opartych o React lub Angular
- Projekty testów jednostkowych NUnit, XUnit, MSTest
- Windows Forms App
- WPF App

Tworzenie własnych wzorców pozwala na szerokie możliwości tworzenia mikroserwisów. Największą wadą są minimalne możliwości konfiguracji wzorca. Jedyna dostępna konfiguracja to prosta podmiana wartości w statycznym wzorcu. Brak możliwości działania na utworzonym projekcie uniemożliwia także generowania bazy danych w oparciu o dostępne modele.

2.1.3 Entity Framework Core

Jednym z dodatkowych narzędzi .NET CLI jest Entity Framework Core. W tym przypadku generowanie kodu dotyczy warstwy dostępu do danych. W połączeniu z odpowiednimi rozszerzeniami dedykowanymi do końcowej bazy danych, narzędzie to pozwala na abstrakcję zarządzania oraz wykorzystywania bazy danych.

Definicja 1

ORM (ang. *object-relational mapping*) nazywamy powiązaniem kolumn tabel bazy danych z modelami i ich właściwościami dostępnymi w klasach. Umożliwia to programiście wykorzystywanie obiektów o różnych typach w przystępny sposób bezpośrednio z aplikacji. Ukryty zostaje także mechanizm sposobu przechowywania danych, dane mogą być modyfikowane w relacyjnych bazach danych lub innych systemach bazodanowych [4]. □

Na podstawie encji w bazie danych, Entity Framework jest w stanie utworzyć strukturę w postaci klas. Zachowaniem odwrotnym jest wygenerowanie skryptów tworzenia bazy danych na podstawie modeli przygotowanych w aplikacji. Wszystkie informacje na temat struktury danych zapisywane są w migracjach co pozwala na łatwe wprowadzanie bądź wycofywanie zmian struktury bazy danych. Dzięki abstrakcji zmiana sposobu przechowywania danych może odbyć się bez zmian w głównej aplikacji.

Listing 1. Przykładowe zapytanie funkcyjne w Entity Framework Core – źródło: opracowanie własne

```
Users.Where(user => user.IsActive && user.Country == "PL")
     .Join(UsersDetail,
           user => user.Id,
           userDetail => userDetail.UserId,
           (user, userDetail) =>
              new { User = user, UserDetail = userDetail })
     .GroupBy(joinResult => joinResult.UserDetail.City)
     .Select(groupResult =>
           new {
                 City = groupResult.Key,
                 Amount = groupResult.Count()
               });
```

Narzędzie może również służyć jako warstwa ORM, która ukrywa przed użytkownikiem szczegóły komunikacji z samym systemem zarządzania danymi. Zapytania napisane w przygotowanym abstrakcyjnym języku zapytań lub języku funkcyjnym, zamieniane są na docelowe zapytania bazy danych. Zapytania pozwalają na odpytywanie, agregację, dodawanie, usuwanie oraz aktualizację danych. Przykładowe zapytanie zaprezentowane na listingu 1 zaimplementowane zostało w postaci funkcyjnej. Na listingu 2 zaprezentowano wynik działania warstwy abstrakcji. Biblioteka przetłumaczyła zapytanie funkcyjne na zapytanie docelowej bazy danych.

Listing 2. Zapytanie przetłumaczone na język zapytań SQLite – źródło: opracowanie własne

```
SELECT "u0"."City", COUNT(*) AS "Amount"
FROM "Users" AS "u"
INNER JOIN "UsersDetail" AS "u0" ON "u"."Id" = "u0"."UserId"
WHERE "u"."IsActive" AND ("u"."Country" = 'PL')
GROUP BY "u0"."City"
```

Entity Framework Core jako narzędzie konsolowe pozwala na łatwą automatyzację w skryptach. Również może to być pomocne w przypadku tworzenia wzorca projektu w narzędziu .NET CLI.

2.2. Generowanie kodu wraz ze wspomaganiem sztucznej inteligencji

Dużą częścią systemów informatycznych są projekty o otwartym źródle dostępnym dla każdego. Publiczne repozytoria tych projektów zostały wykorzystane jako duży zbiór danych dla algorytmów sztucznej inteligencji.

Wyuczony model sugeruje tworzenie kodu na żądanie. Dzięki analizie projektu i komentarzy, model jest w stanie przewidywać i zamieniać intencje programisty w rzeczywisty kod dostosowany do potrzeb danego wymagania. Utworzenie podstawowego projektu polega na wypisaniu kilku komentarzy oraz wybraniu odpowiedniej propozycji oferowanej przez model sztucznej inteligencji.

Główną wadą tego typu generatorów jest brak przewidywalności. W niektórych przypadkach sugerowane rozwiązanie może być poprawne i dostosowane do potrzeb, jednak w niektórych przypadkach może być błędne, bądź nawet niebezpieczne. Znane są przypadki, gdzie sugerowany kod zawierał krytyczne błędy lub klucze prywatne [5]. Najważniejszym elementem w tym przypadku jest, aby programista weryfikował generowany kod. Dodatkowo skuteczność narzędzia jest uzależniona od wybranego języka, wielkości społeczności i ilości publicznych repozytoriów. Problemem, który warto poruszyć jest kwestia związana z prawami autorskimi generowanego kodu. W przypadku propozycji kodu bazującego na repozytorium z kodem o ograniczonej licencji dochodzi do potencjalnych problemów legalności.

2.2.1 Github Copilot

Github Copilot został opublikowany jako prezentacja techniczna i jest najpopularniejszym generatorem, który został oparty o GPT3 oraz został wyuczony na miliardach linii kodu publicznych repozytoriów [6]. Dostęp do narzędzia jest aktualnie zarezerwowany tylko dla zarejestrowanych użytkowników. Github Copilot jako kompan programisty analizuje kontekst aktualnego kodu i przewiduje kolejne kroki. Narzędzie to jest w stanie generować całe funkcjonalne bloki co pozwala na znaczne przyspieszenie implementacji.

W przypadku mikroserwisów sam kompan musiałby być świadomy szerszej skali projektu niż tylko najbliższe klasy i aktualny język programowania. Wczesne stadium tego narzędzia pokazuje też, że jest to nowość a niefunkcjonalne narzędzie do tworzenia pewnego oprogramowania.

2.2.2 Tabnine

Tabnine to mniej popularny i bliźniaczy do Github Copilot generator oparty o sztuczną inteligencję. Podobnie jak w przypadku konkurencyjnego narzędzia model był szkolony na podstawie milionów publicznych repozytoriów. Autorzy tego narzędzia skupili się na wsparciu klientów biznesowych i mniejszych zespołów. Pozwala ono na utworzenie własnego prywatnego modelu na podstawie prywatnych repozytoriów. Model wyuczony w taki sposób podpowiada bardziej trafnie w

kontekście wewnętrznego projektu. Dodatkowo model pomaga w utrzymaniu jednego stylu wytwarzanego oprogramowania [7].

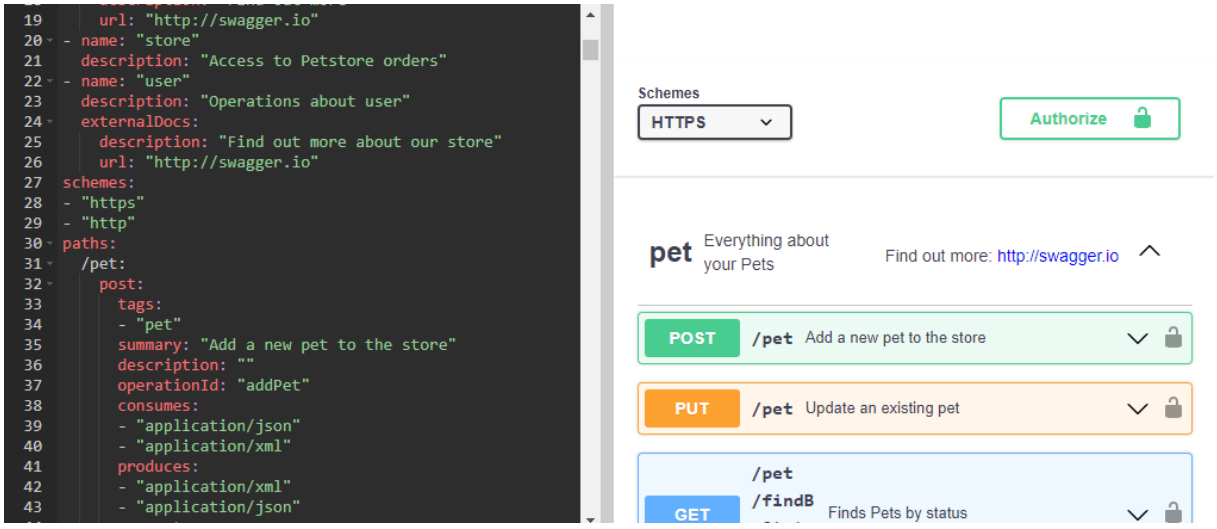
2.3. Generatory ukierunkowane

Najbardziej trafny generator jest narzędziem, które ma zdefiniowany rezultat. Większość wymienionych wcześniej generatorów ma szerokie zastosowanie. Pozwalają na tworzenie projektu gotowego do dostosowania do wybranej architektury lub platformy. Generatory ukierunkowane posiadają mniejszy zakres zastosowania tworzonego projektu. Zyskiem takiej decyzji jest większa trafność projektu w danej dziedzinie. Ukierunkowanie na daną architekturę bądź platformę, oznacza także szersze możliwości konfiguracji. Generatory tego typu zazwyczaj są powiązane z daną platformą programistyczną lub językiem programowania.

2.3.1 Microts

Microts jest generatorem z ideą rozpoczęcia projektu od opisania struktury interfejsów. Specyfikacja OpenAPI pozwala na zdefiniowanie dostępnych metod obsługujących zapytanie, modeli oraz walidacji. Dokumentacja utworzona na bazie OpenAPI jest plikiem wejściowym który jest przetwarzany na szkielet aplikacji [8]. Aplikacje są generowane w języku TypeScript oraz dodatkowo tworzone są testy oraz konfiguracja Docker. Zadaniem narzędzia jest skrócenie czasu implementacji nowych mikroserwisów oraz wymuszenie definiowania dokumentacji.

Rodzaj obsługiwanej dokumentacji wejściowej generatora jest popularnym standardem. Swagger jest grupą narzędzi implementujących ten standard. Przykładowym narzędziem przydatnym w tworzeniu dokumentacji może być Swagger Editor który został pokazany na rysunku 3. To narzędzie pozwala na wyświetlanie na żywo możliwości tworzonej dokumentacji. Edytor zaznacza błędy oraz słowa kluczowe pliku dokumentacji co przyspiesza jej tworzenie.



Rysunek 3. Swagger Editor - źródło: opracowanie własne

Generowane mikroserwisy oparte są o platformę Node oraz Express. Wszystkie pliki źródłowe tych platform znajdują się w katalogu *src*. Ścieżki zapytań HTTP są odwzorowane w strukturze katalogów. Katalogi oraz pliki źródłowe zawierające metody obsługi zapytań są generowane w katalogu *src/handlers*.

Wszystkie niefunkcjonalne aspekty aplikacji są chowane w generowanym kodzie. Programista po otrzymaniu szkieletu skupia się na logice biznesowej aplikacji. Przyspiesza to tworzenie całego projektu a także wspomaga w przypadku dostosowywania oraz poprawiania już istniejącego mikroserwisu.

Głównym atutem tego narzędzia jest zamiana dokumentacji w prawdziwą strukturę aplikacji. Zmienia to podejście programisty, dokumentacja interfejsów przestaje być niepotrzebnym i przestarzałym dokumentem a pełnoprawnym elementem tworzonej aplikacji. Umożliwia to także utrzymanie spójności struktury aplikacji. Każda nowa metoda zapytania powinna zostać najpierw dodana do dokumentacji, wygenerowana, a następnie zaimplementowana przez programistę.

Generator generuje tylko projekt, bez podstawowej logiki, obsługi zapytań lub zapisu danych. Dokumentacja OpenAPI umożliwia opisanie tylko jednej aplikacji, aby umożliwić komunikację między aplikacjami wymagana jest dodatkowa implementacja.

2.3.2 JHipster

JHipster jest jednym z najbardziej rozwiniętych generatorów aplikacji internetowych dostępnych aktualnie na rynku. Jest to platforma do szybkiego generowania, implementowania oraz wdrażania aplikacji monolitycznych lub opartych o architekturę mikroserwisów. Narzędzie wspierane jest przez ponad 600 użytkowników, posiada także wielu sponsorów.

Na rzecz generatora utworzony został specjalny język domenowy JDL (ang. *JHipster Domain Language*), który pozwala na definiowanie struktury aplikacji, wdrożeń, modeli oraz ich relacji [9]. W celu tworzenia plików konfiguracyjnych JDL powstały trzy narzędzia wspomagające:

- Plugin Visual Studio Code - JHipster JDL,
- Plugin Eclipse – JHipster IDE,
- Aplikacja internetowa JDL-Studio

Generator pozwala na tworzenie aplikacji zbudowanych z wielu komponentów. Duża część bibliotek dotyczy budowania interfejsu użytkownika panelu administracyjnego. Najpopularniejszymi bibliotekami dostępnymi do wyboru w tej części aplikacji są:

- React, Angular, Vue,
- Jest, Cypress,
- Bootstrap, Sass
- Webpack

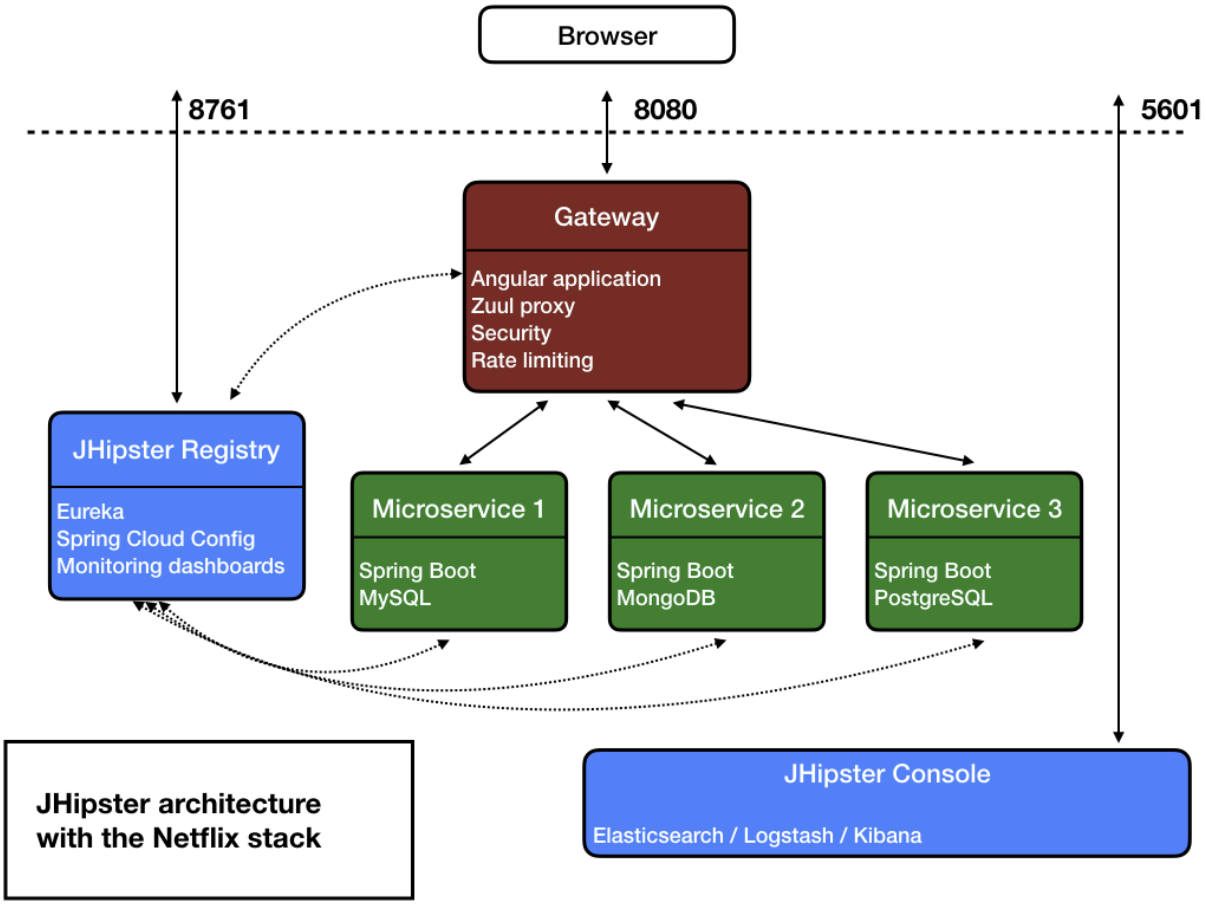
Aplikacje obsługujące zapytania oraz logikę biznesową są oparte na wielu popularnych bibliotekach i platformach programistycznych. Mogą one wykorzystywać wiele różnych dostawców baz danych np.: MySQL, PostgreSQL, Oracle, MSSQL, MongoDB. JHipster dostarcza także rozwiązania dotyczące automatyzacji integracji, budowania oraz wdrożenia aplikacji na serwisy chmurowe.

Główną opcją generatora jest tworzenie aplikacji monolitycznych. Ten tryb generowania oferuje większość dostępnych narzędzi oraz konfiguracji. Tworzona część wizualna jest zintegrowana z częścią aplikacji serwerowej.

Generowanie aplikacji mikroserwisowych jest opcją bardziej ograniczoną ze względu na dostępne biblioteki i technologie. W przypadku tej architektury proces tworzenia został podzielony na dwie części: bramę mikroserwisów (ang. *microservice gateway)* oraz aplikację mikroserwisową (ang. *microservice application*).

Brama jest normalną aplikacją JHipster która wymaga zwyczajnej konfiguracji tak jak w przypadku aplikacji monolitycznych. Brama działa jako wejście do mikroserwisów, pozwala na kierowanie oraz balansowanie ruchem, zapewnienie bezpieczeństwa oraz stabilności całej infrastruktury

mikroserwisów. Architektura aplikacji wejściowej wraz z wykorzystywanymi komponentami została zaprezentowana na rysunku 4.



Rysunek 4. Diagram architektury tworzonych mikroserwisów przy użyciu narzędzia JHipster – źródło: [10]

Aplikacje mikroserwisowe są wyjątkiem w przypadku narzędzia JHipster. Nie posiadają one części wizualnej oraz muszą być dostępne do odkrycia przez rejestr JHipster (ang. *JHipster Registry*). Nie wspierają one także relacji między encjami z osobnych mikroserwisów. Aplikacje tego typu mogą nie posiadać bazy danych, może to być przydatne w przypadku tworzenia warstwy abstrakcji na dotychczasowych starych aplikacjach (ang. *legacy system*) [11].

JHipster jest rozbudowanym narzędziem, które pozwala na generowanie bardzo dużych projektów. Wiele dodatkowych narzędzi pozwala na łatwe monitorowanie oraz zarządzanie aplikacją. Biblioteki wykorzystywane wewnątrz utworzonych projektów są popularne i stosowane także na rzecz projektów tworzonych ręcznie od podstaw. Ułatwia to odnalezienie się programistom w utworzonych projektach. Konfiguracja bibliotek także jest udostępniona programiście. Podłączono wiele narzędzi dotyczących testów, sprawdzania bezpieczeństwa, sprawdzania jakości kodu, automatycznego budowania oraz wdrażania. JHipster umożliwia pominięcie skomplikowanego procesu konfigurowania tych usług.

Operacja generowania aplikacji ma także swoje wady. W celu umożliwienia wsparcia dużej ilości bibliotek i systemów zależnych generowane jest wiele kodu dodatkowego. Typową sytuacją jest wykorzystywanie jednego typu systemu bazodanowego wewnątrz aplikacji. JHipster w tym przypadku tworzy niepotrzebną abstrakcję obsługi każdej innej bazy danych. Implementacja tego typu nie zostanie prawdopodobnie nigdy obsłużona przez procesor. Dodatkowym efektem ubocznym jest zmniejszenie czytelności plików źródłowych. Generator pomija także tworzenie podstawowej obsługi komunikacji między mikroserwisami.

3. Opis narzędzi zastosowanych w pracy

Projekt utworzony na rzecz tej pracy składa się z wielu komponentów z różnego zakresu informatyki. Jako podstawę rozpoczętego projektu wykorzystano język Python, język ogólnego przeznaczenia idealny do tworzenia prototypów. Tworzenie prostych aplikacji mikroserwisowych umożliwiła platforma .NET. Podstawowy projekt utworzony w tej technologii jest łatwy do modyfikowania, co pozwoliło na wykorzystanie go w architekturze mikroserwisów. Narzędzie Entity Framework Core wspomaga programistę w konfiguracji obsługi bazy danych, w tym przypadku zastosowano SQLite. Visual Studio Code jest edytorem plików źródłowych o szerokich możliwościach dostosowania do własnych potrzeb. W przypadku projektu opartego o różne technologie wykorzystanie Visual Studio Code pozwala to na uniknięcie ciągłej zmiany edytora. Wbudowane okno wiersza poleceń i rozszerzenie technologii Docker pozwala na zarzadzanie konteneryzacją ostatecznych aplikacji.

3.1. Python

Do implementacji prototypu wykorzystany został język Python. Jest to język przyjazny programiście i przystosowany do prostych zastosowań. Wszechstronność języka, pomaga w tworzeniu narzędzi, projektów internetowych, skryptów, automatyzacji, aplikacji konsolowych oraz aplikacji z interfejsem użytkownika [12]. Python jest często wykorzystywany jako warstwa abstrakcji do bibliotek bardziej skomplikowanych, które są napisane w językach niskopoziomowych np. C lub C++. Python jest wieloplatformowy oraz jest ciągle aktualizowany, w celu przygotowania tej pracy wykorzystano wersję 3.10.

W przypadku generatora język ten jest najlepszym wyborem ze względu na ułatwienie prototypowania, debugowania i implementacji projektu. Python ułatwi rozwinięcie projektu o dodatkową implementację generowania projektów w innych językach docelowych. Dodatkowo w połączeniu z biblioteką Jinja tworzenie docelowych plików projektu nie wymaga dużej ilości testowania.

Największym minusem tego języka jest brak silnego typowania zmiennych i struktur danych. Obejściem tego problemu są `dataclass`, adnotacje dodane w wersji 3.7. `Dataclass` jest emulacją struktury przygotowanej bezpośrednio do przechowywania danych [13]. Dodanie parametru `frozen` do adnotacji pozwala na wymuszenie niemutowalności danych. Zastosowano ten typ w większości przypadków, aby uniknąć potencjalnych problemów z nieintencjonalną zmianą zawartości struktury. Pozwala to też na wymuszenie typów zawartych w danej strukturze co ułatwia rozwój i debugowanie aplikacji.

3.1.1 Pip

Język Python zawiera dużą ilość bibliotek i narzędzi dodatkowych wspomagających programistę w organizacji projektu. Najpopularniejszym narzędziem do rozwiązywania zależności jest Pip. Pip pozwala na instalację, usuwanie, aktualizowanie i ogólną automatyzację zarządzania pakietami zewnętrznymi [14].

Podstawowymi komendami narzędzia Pip w systemie Windows są:

- Instalacja globalna pakietu o nazwie „Pakiet”: `py -m pip install 'Pakiet'`
- Instalacja globalna listy pakietów zapisanych w pliku `requirements.txt`: `py -m pip install -r requirements.txt`
- Deinstalacja globalnego pakietu o nazwie „Pakiet”: `py -m pip uninstall 'Pakiet'`

3.1.2 Venv

Wirtualne środowiska pozwalają Pythonowi na instalację bibliotek w odizolowanym środowisku dla wybranej aplikacji. Komendy instalacji pakietów modyfikują system globalnie. Zmiany globalne mogą modyfikować zachowanie w pozostałych aplikacjach w przypadku, gdy inne aplikacje korzystają z tego samego pakietu w innej wersji [14]. Dodatkowo programista może chcieć uniknąć modyfikowania swojego systemu każdą nową biblioteką w przypadku uruchomienia nowego projektu. Wszystkie z tych problemów rozwiązują środowiska wirtualne venv.

Przykładowe komendy venv:

- Utworzenie wirtualnego środowiska w katalogu *srodowisko*: `py -m venv srodowisko`
- Uruchomienie wirtualnego środowiska w katalogu *srodowisko*: `srodowisko\Scripts\activate`

3.1.3 Jinja

Jedyną biblioteką użytą przy tworzeniu prototypu jest Jinja. Jest to szybka, rozszerzalna oraz zawierająca wiele dodatkowych funkcji biblioteka obsługi wzorców. Umożliwia zamianę wzorca oraz danych w pełny ciąg tekstowy. Specjalne znaczniki dodawane do wzorca przypominają funkcje języka Python. Wzorce są proste do utworzenia oraz zaimportowania wewnątrz programu.

Listing 3. Wzorzec zawierający procedurę tworzenia nagłówka oraz listy imion i nazwisk – źródło: opracowanie własne

```
<h1>{{ naglowek }}</h1>
<div>
      <ul>
            {% for osoba in osoby %}
            <li>{{ osoba.imie }} {{ osoba.nazwisko }}</li>
            {% endfor %}
      </ul>
</div>
```

Główną klasą pakietu jest `Environment`. Struktura ta zawiera konfigurację, filtry oraz zmienne globalne. Przekazany parametr `loader` określa w jaki sposób są odczytywane wzorce, na których wykonywane są operacje przetwarzania tekstu [15]. Klasy ładujące pozwalają na ładowanie wzorów z wielu źródeł takich jak:

1. System plików
2. Pakiet języka Python
3. Słownik języka Python
4. Prekompilowanych wzorców zapisanych w module

Następną najważniejszą klasą jest `Template`. Jest to klasa, która nie powinna być tworzona bezpośrednio, a otrzymana z przygotowanego wcześniej obiektu klasy `Environment`. W celu otrzymania wybranego zapisanego w pamięci wzorca należy uruchomić metodę `get_template` z parametrem nazwy wzorca. Wzorzec może zostać zamieniony w postać tekstową przy pomocy metody `render`. Metoda ta przyjmuje dane w postaci słownika, gdzie klucz to nazwa znacznika, który będzie podmieniany, a wartość słownika to wartość na jaką znacznik zostanie zamieniony.

Przykład wzorca został przedstawiony na listingu 3, zawiera on procedurę generowania krótkiego kodu źródłowego w formacie HTML (ang. *HyperText Markup Language*). W celu wypełnienia tego

wzorca należy do metody `render` przekazać słownik zawierający klucz `naglowek` którego wartość zastąpi znacznik `{{ naglowek }}`. Wpis w słowniku znajdujący się pod kluczem `osoby` powinien zawierać listę obiektów zbudowanych z właściwości `imie` oraz `nazwisko`. Znacznik `for` iteruje po otrzymanej liście i tworzy odpowiednią ilość znaczników `<li>`.

3.2. .NET

.NET to darmowa otwartoźródłowa platforma programistyczną do tworzenia aplikacji. .NET pozwala na tworzenie aplikacji o szerokim zastosowaniu. Platforma ta może zostać wykorzystana do:

- Aplikacji internetowych,
- Aplikacji mobilnych,
- Aplikacji komputerowych,
- Mikroserwisów,
- Bezstanowych funkcji chmurowych,
- Tworzenia gier komputerowych,
- Oprogramowania Internetu Rzeczy,

Wszystkie domyślne narzędzia dostępne w tej platformie są rozwijane przez społeczność wraz z pracownikami firmy Microsoft. Poza domyślnie oferowanymi narzędziami, dostępne są także dodatkowe tworzone przez społeczność biblioteki w postaci pakietów NuGet. Mogą one być zainstalowane przy pomocy polecenia `dotnet add package NazwaPakietuNuGet`.

Platforma .NET została zastosowana jako środowisko dla generowanych aplikacji mikroserwisowych. Narzędzie .NET CLI i wbudowane wzorce tworzą aplikacje małych rozmiarów oraz o małej ilości kodu. Utworzone w ten sposób aplikacje są łatwe do modyfikacji dla programisty, ale także i programu automatyzującego. Podstawą każdego generowanego mikroserwisu jest utworzenie projektu komendą `dotnet new web -o NazwaProjektuMikroserwisu`. Instalacja narzędzi bezpośrednio do wygenerowanego projektu wymaga dodatkowego *manifestu*. Utworzenie pliku zawierającego informację o narzędziach można dokonać komendą `dotnet new tool-manifest`. Plik `dotnet-tools.json` w katalogu `.config` poza zainstalowanymi narzędziami zawiera informację o dostępnych komendach oraz ich wersjach.

3.2.1 Entity framework

Jedynym z narzędzi instalowanych do wygenerowanego mikroserwisu jest Entity framework core. Pozwala ono na utworzenie specyficznej dla projektu warstwy komunikacji z bazą danych. Komunikacja odbywa się przy pomocy kontekstu, klasy będącej abstrakcją na sterowniku docelowej bazy danych. Na podstawie modeli oraz utworzonego kontekstu utworzone są pliki migracyjne. Zawierają one instrukcję operacji do wykonania na bazie danych w celu utworzenia wymaganej struktury. Odwzorowywane są relacje, modele, klucze podstawowe, indeksy oraz wiele innych atrybutów dostępnych w danym sterowniku bazy danych. Odzwierciedla to w pełni docelową strukturę układu modeli zapisanych w projekcie. Wykorzystanie *Entity framework* w przypadku omawianego w pracy generatora pozwoliło na skrócenie implementacji zarządzania warstwą danych projektów docelowych. Jedyne kroki, które były potrzebne do wytworzenia warstwy danych to: utworzenie kontekstu, modeli i uruchomienie komend aktualizujących bazę danych.

3.2.2 Generatory kodu oraz biblioteka Refit

Autor wybrał szóstą wersje platformy .NET ze względu na jej możliwości generowania kodu. Generatory kodu (ang. s*ource generators)* umożliwia uruchomienie specjalnie przygotowanego kodu podczas procesu kompilacji. Generator pozwala na tworzenie plików źródłowych w locie. *Source generators* ma wgląd w aktualnie utworzony projekt, na podstawie tych informacji generator może zdecydować, czy powinien utworzyć dodatkowe pliki. W przypadku C# ułatwia to działanie na platformie *.NET,* dodatkowo pozwala na kolejną optymalizację. Generator pozwala na optymalizację refleksji wbudowanego w język C# mechanizmu, którego działanie opiera się na zarządzaniu typami oraz modułami. Sam mechanizm jest powolny, lecz pozwala na zaawansowaną manipulację obiektami i powiązaniami między nimi. Generatory kodu w pewnym stopniu mogą zastąpić lub pozwolić na optymalizację większości platform programistycznych jak i ułatwić korzystanie z nich.

Przykładowym zastosowaniem generatora kodu jest tłumaczenie zapisanych wyrażeń nieregularnych na kod zapisany bezpośrednio w języku C#. Analiza wyrażeń nieregularnych podczas uruchomionej aplikacji może być nieoptymalna. Przetłumaczona wersja może być dodatkowo zoptymalizowana a także przetłumaczona dalej, dzięki kompilatorowi *JIT (ang. Just in Time)*.

Poza zastosowaniem optymalizacyjnym generator kodu ma na celu pomaganie programiście poprzez zmniejszanie ilości powtarzalnego kodu (ang. *boilerplate code*). Przykładem takiego zastosowania jest biblioteka NuGet Refit, wspomaga ona programistę poprzez skrócenie implementacji kodu dotyczącego zapytań HTTP do zewnętrznych aplikacji webowych. Jedynym zadaniem programisty jest utworzenie interfejsu zawierającego szczegóły dotyczące protokołu HTTP. Przykładowymi adnotacjami dodanymi wraz z pakietem są:

- Metody HTTP: `Get`, `Post`, `Put`, `Delete`, `Patch` oraz `Head`,
- `Headers` – określenie nagłówków wysyłanych w komunikacie HTTP,
- `Multipart` – wysyłanie wieloczęściowego typu danych,
- `Query` – dołączenie zapytań wyszukiwania,
- `Body` – określenie w jaki sposób dane przekazane do interfejsu będą przekazywane w ciele komunikatu HTTP

Programista przy użyciu adnotacji i utworzonego interfejsu specyfikuje wszystkie informacje dotyczące zapytania HTTP. Podczas procesu kompilacji na podstawie wskazanych adnotacji i interfejsów generowana jest implementacja. Implementacja opiera się na klasie wbudowanej w C# `HttpClient`, pozwala to na łatwe dostosowanie budowanych implementacji do własnych potrzeb. Refit zamienia wykorzystywanie zewnętrznego API HTTP w prosty interfejs. Ułatwia to zarządzanie dependencjami, testowanie oraz zmniejsza potencjalny błąd ludzki. W przypadku generatora przygotowanego na rzecz tej pracy *Refit* jest odpowiedzialny za utworzenie warstwy komunikacji z innymi mikroserwisami.

3.3. Repozytorium

System kontroli wersji jest to narzędzie, które służy do śledzenia historii oraz wpływu użytkowników na projekt. Docelowo projekt jest przechowywany w repozytorium, które umożliwia większej ilości programistów dostęp do danych plików oraz kolaborację w przypadku dokonywanych zmian. Nowoczesne systemy kontroli wersji pozwalają na pracę w tym samym czasie, nie blokując przy tym innych użytkowników. Klonowane repozytoria dają użytkownikowi ich własne środowisko do dokonania zmian bez wpływu innych programistów. W przypadku współpracy większej ilości programistów, system kontroli wersji oferuje mechanizmy łączenia oraz możliwość synchronizacji kodu.

Git jest przodującym systemem rozproszonego zarządzania wersją. Początkowo został on stworzony przez Linusa Torvaldsa jako system SCM (ang. *source control management*), który służył do stosowania w przypadku budowanego *kernela* systemu Linux. Aktualnie git zdominował większość rynku *open-source*, ale również jest używany przez większość organizacji do projektów prywatnych oraz własnościowych [16].

Zastosowanie tego narzędzia jest szerokie, a głównymi funkcjonalnościami są:

- Przechowywanie historii plików, autorów oraz tworzonych modyfikacji,
- Wyszukiwanie historii zmian oraz umożliwienie porównania zmian,
- Podział zmian w czasie na *commity*, gałęzie oraz *tagi*,
- Praca równoległa dzięki gałęziom i strategiom związanym z ich łączeniem,
- Zarządzanie konfliktami łączonych zmian,
- Rozproszone repozytorium umożliwia odzyskanie utraconych informacji,

Każdy stworzony projekt powinien posiadać utworzone repozytorium lokalne oraz zdalne. Usprawnia to potencjalny rozwój, kolaborację a także łatwe wycofanie błędnych zmian. W przypadku generatora, implementacja oraz jej historia jest przechowywana na prywatnym, zdalnym repozytorium na platformie Github.

3.4. Visual Studio Code

Visual Studio Code jest lekkim narzędziem do edycji kodu źródłowego, które pozwala na duże możliwości rozszerzania. Aplikację można uruchomić w systemie Windows, macOS, Linux oraz wielu innych. Domyślnie wspierany jest język JavaScript, TypeScript i Node.js. Wiele dostępnych rozszerzeń pozwala na dostosowanie edytora do potrzeb każdego innego języka. Główne elementy wbudowane w edytor to [17]:

- Zarządzanie systemem kontroli wersji, domyślnie Git,
- Obsługa debugowania,
- Zadania automatyzacyjne wykonywane w wierszu poleceń,
- IntelliSense – system podpowiedzi będącym głównym wsparciem programisty,
- Rozbudowane możliwości edycji tekstu,
- Obsługa wiersza poleceń wewnątrz edytora,

Dostępne za darmo rozszerzenia umożliwiają na zmianę działania narzędzia oraz obsługę dodatkowych języków programowania. Przykładowe funkcjonalności dostępne jako rozszerzenia to:

- Wsparcie popularnych języków: Python, C++, C#, Java, HTML, CSS,
- Zaawansowane zarządzanie systemem kontroli wersji Git,
- Formatowanie plików źródłowych,
- Podpowiedzi związane z błędami, ostrzeżeniami dot. implementowanego kodu,
- Obsługa konteneryzacji,
- Obsługa systemów kompilacji,
- Zdalna edycja kodu przez tunel SSH bądź wewnątrz kontenera,

Visual Studio Code wraz z odpowiednimi rozszerzeniami ułatwiło pracę z językiem Python, modyfikacje plików źródłowych, debugowanie oraz uruchamianie generatora. Utworzone projekty w języku C# mogły także być modyfikowane oraz weryfikowane bez zmiany środowiska programistycznego. Rozszerzenie Docker pozwala na zarządzanie konteneryzacją i konfiguracją tworzonych mikroserwisów.

Edytor ten stał się jednym z podstawowych narzędzi każdego programisty. Na 82 tysiące odpowiedzi w ankiecie przygotowanej w 2021 roku przez StackOverflow około 58 tysięcy osób odpowiedziało, że wykorzystuje ten edytor regularnie oraz będzie wykorzystywać go w przyszłości [18].

3.5. SQLite

SQLite jest systemem bazodanowym, który pozwala na zapis danych w pliku. Baza danych jest prosta w obsłudze oraz jest wspierana przez systemy operacyjne macOS, Windows, Linux oraz Android. SQLite posiada wiele bibliotek, które są dostępne w większości języków programowania. Cały projekt jest dostępny publicznie oraz jest darmowy do zastosowań prywatnych jak i komercyjnych. Biblioteka działa wewnątrz procesu tworzonej aplikacji, nie wymaga dodatkowego serwera jak w przypadku innych odmian języka SQL. Dane zapisywane są bezpośrednio do pliku znajdującego się na dysku. Format pliku bazodanowego jest niezależny od systemu operacyjnego, dlatego tworzy to idealną opcją do wykorzystania jako mały system obsługi danych [19].

SQLite stał się jedną z najpopularniejszych baz danych przechowywanych wewnątrz aplikacji. W ankiecie z 2021 roku przygotowanej przez StackOverflow aż 32% uczestników z 73 tysięcy odpowiedzi wskazało, że bardzo często korzysta z tego typu bazy [20].

W przypadku prototypu, baza danych SQLite jest przechowywana wewnątrz aplikacji. Ułatwiona konfiguracja pozwoliła na uniknięcie dodatkowej implementacji logiki, tworzenia konfiguracji połączenia z bazą danych oraz tworzenia samej bazy danych.

3.6. Konteneryzacja

Większość projektów, które są aktualnie tworzone posiada wiele konfiguracji, zależności oraz dodatkowych usług potrzebnych do działania. W architekturze mikroserwisów problem ten występuję częściej ze względu na dzielenie budowanego systemu na mniejsze części. W przypadku dużych projektów, ilość mikroserwisów sięga setek lub tysięcy. Są to osobne aplikacje, które wykonują pojedynczą operację. Zazwyczaj w przypadku aplikacji opartych na tej architekturze potrzebne są też usługi zewnętrzne jak baza danych lub *cache*.

Poza problemami, które dotyczą ilości konfiguracji oraz różnorodnych aplikacji, problemem, który warto poruszyć jest także skalowanie wydajności aplikacji. Duża ilość zapytań oznacza zwiększający się czas odpowiedzi. W przypadku skalowania wertykalnego aplikacji poprawiana jest moc obliczeniowa sprzętu, na którym jest uruchamiana aplikacja. Wertykalne skalowanie ma swoje ograniczenia sprzętowe, które dodatkowo zwiększają koszt infrastruktury. Skalowanie horyzontalne w opozycji do wertykalnego skupia się na zwiększeniu ilości uruchomionych kopii aplikacji. Zapytania mogą być balansowane między aplikacjami, które wykonują tą samą pracę. Wymaga to większego nakładu implementacji oraz planu architektury systemu, czego zyskiem jest zredukowany koszt oraz zwiększona odporność na problemy działania.

Duża ilość projektów jak i zależności powoduje utrudniony próg rozpoczęcia pracy z projektem oraz proces debugowania. Jednym z rozwiązań tego problemu jest konteneryzacja.

Definicja 2

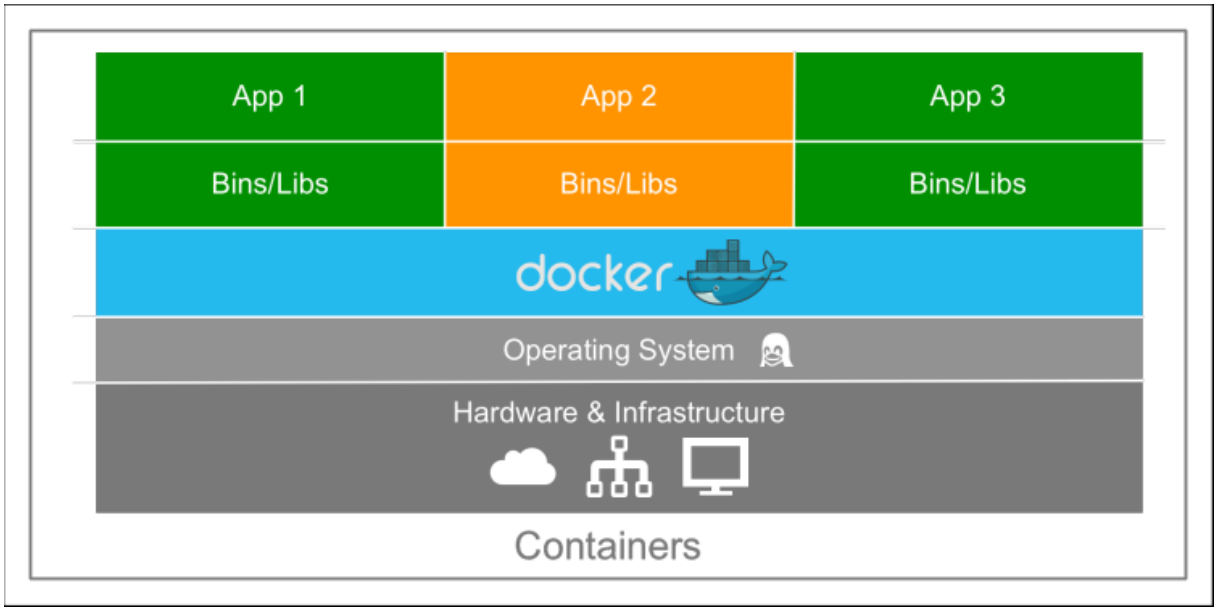
Konteneryzacją nazywamy opakowanie kodu aplikacji wraz z systemem operacyjnym, bibliotekami oraz zależnościami potrzebnymi do uruchomienia. Aplikacje utworzone na podstawie obrazów nazywane są kontenerem. Kontener działa zgodnie na każdej infrastrukturze obsługującej konteneryzację, dodatkowo są przenośne i bardziej oszczędne pod względem zasobów w porównaniu do maszyn wirtualnych [21]. □

Konteneryzacja pozwala programistom na tworzenie oraz publikowanie aplikacji szybciej i bezpieczniej. Aplikacje zawierają w sobie konfigurację oraz całą infrastrukturę potrzebną do działania. Pozwala to na łatwe przenoszenie całej aplikacji pomiędzy środowiskami a także programistami.

3.6.1 Docker

Docker jest wiodącą platformą obsługi kontenerów. Konteneryzacja jest stosowana, aby wyeliminować problem niezgodności środowiska, który występuje przy współpracy z innymi programistami. Konteneryzacja pozwala na pełne wykorzystanie mocy obliczeniowej poprzez uruchomienie kontenerów obok siebie. W przypadku dużej ilości zależności, proces skonfigurowania środowiska developerskiego, które odzwierciedlałoby środowisko produkcyjne jest prawie niemożliwe. Pierwszą próbą rozwiązania tego problemu było przygotowywanie środowisk wirtualnych. Środowisko w ten sposób jest w pełni odtworzone. Problemem pozostaje wydajność i zapotrzebowanie przestrzeni dyskowej. W przypadku maszyn wirtualnych obrazy mogą zajmować setki megabajtów przestrzeni dyskowej i pamięciowej.

Docker rozwiązuje problem niespójności środowiska poprzez utworzenie warstwy emulacji działającej wewnątrz systemu operacyjnego co zostało przedstawione na rysunku 5. W odróżnieniu od maszyn wirtualnych część wspólna kontenerów może być współdzielona np. system operacyjny oraz środowiska programistyczne.



Rysunek 5. Układ podziału warstw i aplikacji w środowisku Docker – źródło: [22]

Do utworzenia obrazu z lokalnego projektu wystarczy przygotować plik `Dockerfile` będący skryptem poleceń. Podstawowymi poleceniami są:

- FROM – wskazanie podstawowego obrazu, na którym mają zostać wykonane pozostałe polecenia,

- WORKDIR – wskazanie katalogu, w którym wykonywane są kolejne polecenia,
- COPY – kopiowanie plików z systemu operacyjnego do tworzonego obrazu,
- RUN – uruchomienie polecenia w terminalu wewnątrz kontenera,

3.6.2 Docker compose

Docker umożliwia uruchamianie dwóch lub więcej aplikacji obok siebie, posiadając różniące się wersje zależnego oprogramowania lub systemu. W przypadku mniejszych projektów uruchamianie kontenera jest proste. Uruchamianie manualne ma swoje wady, w przypadku większej ilości kontenerów do uruchomienia dodatkowo potrzebna jest instrukcja poleceń do wykonania. Przykładowo, aby uruchomić kontener1 i podłączyć do niego dostępny publicznie kontener redis należy wykonać polecenia przedstawione na listingu 4.

Listing 4. Uruchomienie kontenerów redis i kontener1 połączonych siecią – źródło: [22]

```
docker image pull redis:alpine
docker image pull kontener1
docker network create siec-kontenera-1
docker container run -d --name redis --network siec-kontenera-1 redis:alpine
docker container run -d --name kontener1-instancja --network siec-kontenera-1 kontener1
```

Odpowiedzialność Dockera powinna kończyć się na tworzeniu obrazów i uruchamianiu ich. Odseparowano w ten sposób zarzadzanie jednym kontenerem od zarządzania scenariuszami większej ilości kontenerów współpracujących ze sobą. W celu automatyzacji uruchamiania, restartowania i multiplikowania kontenerów powstał Docker compose.

Docker compose wykorzystuje pliki YAML nazwane zazwyczaj `docker-compose.yml` do zdefiniowania jak wielokontenerowa aplikacja powinna zostać zbudowana. Zapisane parametry, wolumeny, sieci oraz zależności wskazują wymagania docelowe dla Docker compose. Przykład pliku `docker-compose.yml` został zaprezentowany na listingu 5.

Listing 5. Przykład pliku docker-compose.yml – źródło: opracowanie własne

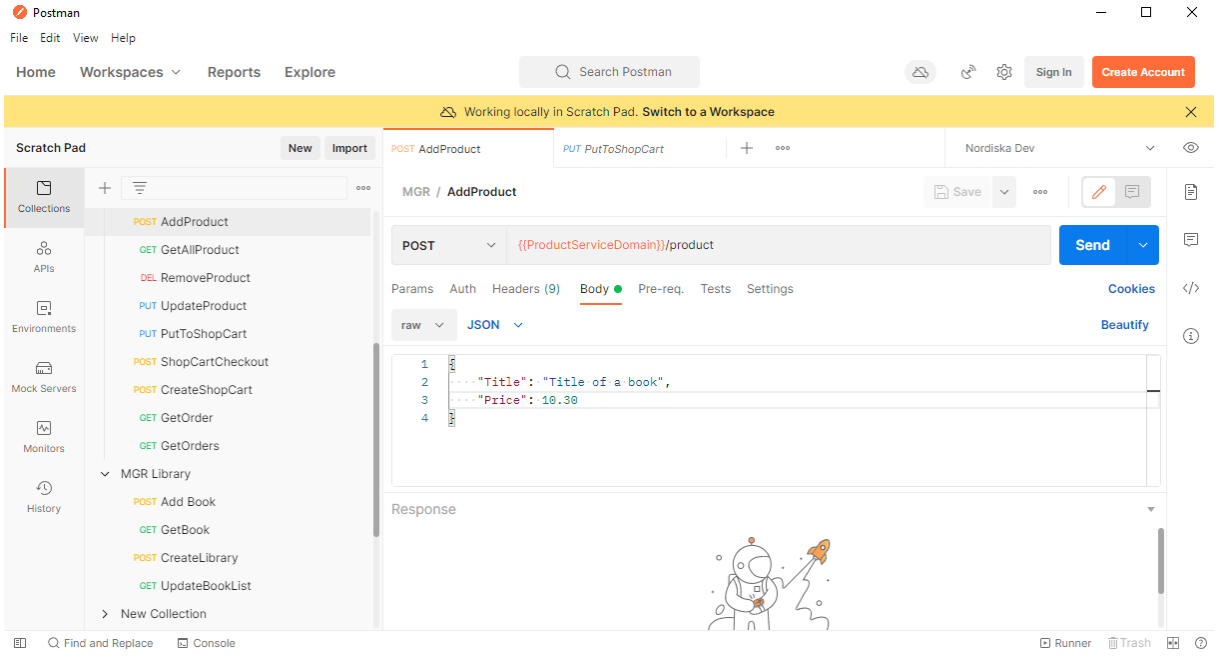
```yaml
version: "3.9"  # optional since v1.27.0
services:
  productservice:
    build: ProductService
    ports:
        - "8000:80"
  shopcartservice:
    build: ShopCartService
    ports:
        - "8001:80"
  orderservice:
    build: OrderService
    ports:
        - "8002:80"
```

Komenda `docker-compose up` wyszukuje plik, następnie uruchamia kontenery w odpowiedniej kolejności. Usunięcie kontenerów oraz sieci stworzonych przez komendę inicjalizującą może być wykonane poleceniem `docker-compose down`.

Docker i narzędzie Docker compose są podstawowymi narzędziami potrzebnymi do pracy w architekturze mikroserwisów. W celu pełnego odwzorowania środowiska tej architektury prototyp przygotowany na rzecz tej pracy tworzy inicjalne pliki `Dockerfile` oraz `docker-compose.yml`. Pozwalają one na uruchomienie całej infrastruktury jednym poleceniem.

3.7. Postman

Postman jest narzędziem, które pozwala na wykonywanie zapytań HTTP. Do takiego prostego zadania może zostać wykorzystane narzędzie takie jak cURL. Zaletą Postmana na tle konkurencyjnych narzędzi jest sposób organizacji projektów, zmiennych oraz środowisk w przyjaznym dla użytkownika interfejsie graficznym, który został przedstawiony na rysunku 6.



Rysunek 6. Interfejs użytkownika aplikacji Postman – źródło: opracowanie własne

Głównymi funkcjonalnościami aplikacji Postman są [23]:

- Obsługa zapytań protokołów np. HTTP, gRPC, WebSocket, mają one zastosowanie w aktualnie tworzonych systemach informatycznych,
- Organizowanie zapytań w kolekcje,
- Eksportowanie kolekcji dla innych programistów pracujących nad aplikacją testowaną,
- Konfiguracja zapytań: parametry, uwierzytelnianie, nagłówki, ciało zapytania, cookie,
- Dokumentowanie API,
- Zarządzanie zmiennymi dostępnymi na warstwie kolekcji lub środowiska,
- Wsparcie dla autoryzacji OAuth, Bearer Token, Basic Auth, API Key, NTLM oraz innych,

Zapytania i kolekcje mogą być tworzone od początku ręcznie bądź zostać zaimportowane z wielu źródeł. Aktualnie wspierane są OpenAPI, GraphQL, cURL, WSDL oraz pliki HAR. W przypadku OpenAPI jest to specyfikacja, która definiuje interfejs połączenia do RESTful API. Przy dobrej implementacji i opisaniu możliwości API, ułatwia ono korzystanie użytkownikowi końcowemu. Postman umożliwia importowanie ze specyfikacji OpenAPI co doprowadza do utworzenia przykładowych zapytań, zmiennych, modeli oraz dokumentacji.

3.8. Swagger

Swagger jest projektem *open-source* implementującym specyfikację OpenAPI. Umożliwia dokumentowanie udostępnianego API wewnątrz aplikacji. Ważny jest też tworzony specjalny interfejs graficzny, który umożliwia proste testowanie działania API. W celu skorzystania z domyślnie przygotowanego interfejsu należy otworzyć adres URL: `https://<domena>:<port>/swagger`

Specyfikacja pozwala na określenie:

- Dostępnych modeli,
- *Endpointów* i powiązanych z nimi informacjami np. nagłówki, odpowiedzi oraz ciało zapytania,
- Sposobów autoryzacji,
- Typów danych
- Informacji ogólnych: autorów, kontaktu, wersji, opisów,
- Licencji,

W przypadku platformy ASP .NET należy dodać odpowiednią konfigurację usługi, aby skorzystać ze Swaggera. Samo dokumentowanie końcówek API może być wykonane przy pomocy komentarzy XML co zostało przedstawione na listingu 6 [24].

Listing 6. Przykładowa implementacja *endpointu* ASP .NET z dokumentującym komentarzem XML – źródło: [24]

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item #1",
///        "isComplete": true
///     }
/// </remarks>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(Get), new { id = item.Id }, item);
}
```

Generator mikroserwisów tworzy konfigurację potrzebną do działania Swaggera, pomoże to w udokumentowaniu dostępnych *endpointów*, ale także w przypadku testowania dzięki możliwości importowania specyfikacji OpenAPI do Postmana. Specyfikacja OpenAPI udostępniana przez Swaggera jest dostępna pod adresem: `https://<domena>:<port>/swagger/v1/swagger.json`.

4. Metodyka low-code w generowaniu mikroserwisów

Rozdział ten zawiera dokładniejsze omówienie architektury mikroserwisów. W przeciwieństwie do monolitu aplikacje dzielą cały system na dziedziny. Opisano także popularne pomysły układu oraz komunikacji aplikacji wykorzystywane w tej architekturze.

W pozostałej części tego rozdziału przybliżono definicję hasła *low-code*, a także przykładowe narzędzia wizualne związane z tym konceptem. Podobnie jak narzędzia wizualne konfiguracja prototypu generatora opiera swoje działanie na zmniejszonej ilości konfiguracji. Szczegóły implementacyjne są schowane przed użytkownikiem co pozwala na ułatwienie tworzenia projektów. Podstawowa konfiguracja wymagana od użytkownika oraz spodziewany rezultat działania zostały przedstawione na końcu tego rozdziału.

4.1. Mikroserwisy

Dotychczas tworzone aplikacje monolityczne były oparte na zamknięciu całej logiki wewnątrz jednego projektu. Informacje usług zewnętrznych mogły być obsługiwane przy pomocy rozmaitych protokołów komunikacyjnych. Rosnąca ilość użytkowników serwisów internetowych oznacza dłuższy czas wykonywania operacji przez jedną aplikację. Częściowym rozwiązaniem tego problemu było skalowanie wertykalne. Polega ono na zwiększeniu ilości zasobów sprzętu na którym uruchomiona jest aplikacja. Zwiększenie szybkości procesora oznacza szybszą obsługę zapytania a co za tym idzie szybszą odpowiedź. W przypadku skalowania wertykalnego istnieje problem dużych skoków zapytań. Przykładem takiej sytuacji jest uruchomienie korzystnej promocji w sklepie internetowym. Jedna aplikacja po wykorzystaniu maksimum możliwości sprzętu zacznie opóźniać odpowiedź każdemu kolejnemu użytkownikowi.

Rozbicie dużych rozmiarów aplikacji na mniejsze pozwala na rozłożenie obciążenia na różne maszyny wykonujące obliczenia. Dodatkowo w przypadku poprawnego zaimplementowania komunikacji i dostępu do danych aplikacje mogą być skalowane w zależności od aktualnych potrzeb. W przypadku zwiększenia się ilości zapytań aplikacje mogą zostać uruchomione w osobnych kontenerach na osobnym sprzęcie. Podejście tego typu wiąże się z wieloma problemami implementacyjnymi. Podstawowym z nich jest wymagane odpowiednie rozbicie struktury aplikacji i komunikacji między nimi. Najbardziej perfekcyjnym scenariuszem jest uniknięcie wyścigu po dane przechowywane w bazie danych. Aplikacje wymagają dodatkowego czasu implementacji, ponieważ wymagają wzięcie pod uwagę problemu wysłania zapytania do aplikacji nie działającej lub odpowiadającej z opóźnieniem.

4.1.1 Architektura infrastruktury

Powstało wiele konceptów i ustaleń architektonicznych które pomagają w przypadku infrastruktury mikroserwisów. Pomagają one w przypadku adaptowania projektu monolitycznego lub pisania systemu opartego o tę architekturę od początku.

Przykładem tego typu wspomagania jest koncepcja kontenerów. Aplikacje są odizolowane od środowiska, w którym są uruchamiane. Mogą być uruchamiane wielokrotnie oraz w przypadku sytuacji krytycznej aplikacja może zostać zrestartowana. Powstały specjalne platformy do zarządzania tego typu problemami. Same aplikacje mikroserwisowe powinny być także świadome statusu innych aplikacji. Utworzono do tego celu specjalne biblioteki, które zajmują się rozsyłaniem statusu aktualnej aplikacji oraz odbieraniem jakości stanu aplikacji pozostałych.

W celu abstrakcji odbiorcy komunikatu w sieci mikroserwisów zaczęto wykorzystywać bramę API (ang. *application programming interface*). Jest to niewielkich rozmiarów aplikacja, która ma za zadanie oddelegowywanie otrzymanych zapytań do odpowiednich odbiorców. Zajmuje się

dyrygowaniem oraz balansowaniem ruchu otrzymanego. W sytuacji, gdy pewna aplikacja jest zwielokrotniona brama musi zdecydować o tym, żeby otrzymany komunikat trafił do najmniej zajętej aplikacji. Brama API w niektórych przypadkach może także mieć zadanie bardziej funkcjonalne. Jest zazwyczaj aplikacją tworzoną jako pierwszy krok przy migracji aplikacji monolitycznej na architekturę mikroserwisową. Aplikacja monolityczna może zostać schowana za bramą, aby wciąż umożliwić wspieraną komunikacje. Brama może zdecydować, kiedy przekierowywać komunikaty do monolitu a w jakim przypadku do nowo powstałego mikroserwisu. Zastosowanie takiego podejścia umożliwia także testy A/B. Pewna procentowa część komunikatów może być przekierowywana do nowej funkcjonalności a pozostałe do starej. W ten sposób można zweryfikować, czy wszystko działa poprawnie oraz przeprowadzać badania zadowolenia klienta.

Kolejnym elementem związanym z komunikacją są kolejki oraz szyny komunikacyjne. W przypadku dużej ilości wysyłanych komunikatów nadawca nie może być pewny, że komunikat został dostarczony i czy zostanie odpowiednio obsłużony. Dużo czynników wpływa na poprawną komunikację między aplikacjami. Kolejki są jednym ze sposobów radzenia sobie z problemem zblokowanych aplikacji oraz działania asynchronicznego. W pewnych sytuacjach odebrany komunikat nie musi być przetworzony w momencie otrzymania go. W tej sytuacji kolejka może zostać wypełniona takimi wiadomościami i obsłużona w odpowiednim momencie oraz odpowiedniej kolejności. Dzięki takiemu wzorcu można tworzyć podejście komunikacji opartej o wydarzenia [25]. Przykładem takiego wydarzenia może być zarejestrowanie się użytkownika. Poza dodaniem użytkownika do bazy danych, aplikacja rozsyła informację o wydarzeniu. Każda zainteresowana aplikacja subskrybuje oczekiwaną kolejkę w celu otrzymania informacji na wybrany temat. Wszystkie komunikaty tego typu powinny być wysyłany przy pomocy szyny komunikacyjnej.

Istnieje także podejście prostsze oraz bardziej bliskie komunikacji w architekturze monolitycznej. Zapytanie wysyłane jest bezpośrednio do aplikacji, która zajmuje się odpowiedzią w momencie przetworzenia. Rozwiązanie tego typu może oznaczać potencjalne problemy. Pierwszym z nich jest to, że nadawca musi być świadomy odbiorcy i spodziewać się, że jest aktualnie dostępny. W przypadku braku dostępności odbiorcy wykorzystuje się mechanizmy powtarzania zapytania. Długa niedostępność odbiorcy może oznaczać, że cały proces rozpoczęty przez użytkownika końcowego nie zostanie wykonany. Szyna komunikacyjna odwraca ten problem i zazwyczaj subskrybenci odbierają komunikat, kiedy chcą i wysyłają odpowiedź ponownie. To podejście zmusza aplikację do bardziej aktywnego działania i reagowania na jego rezultat.

Wadą tego typu wymiany komunikacji jest duże spowolnienie działania. Aplikacje monolityczne posiadają wszystko wewnątrz lub lokalnie. Przygotowanie odpowiedzi dla użytkownika wymaga zazwyczaj zapytanie kilku innych aplikacji o odpowiedź.

4.1.2 Układ aplikacji

Projekty mikroserwisowe są zazwyczaj dużo mniejsze niż ich monolityczne odpowiedniki. Skupiają się na jednej dziedzinie np. zamówienia, użytkownicy, dokumenty itp. Aplikacje powinny być odseparowane od innych i tylko wymagać swoich wewnętrznych zależności.

Popularnym zjawiskiem jest tworzenie specyficznej bazy danych tylko na rzecz danego mikroserwisu. Taki system bazodanowy nie powinien być dostępny dla innych mikroserwisów. W przypadku zwielokrotnienia aplikacji może oznaczać utworzenie wyścigu zapisywania danych. Popularnym rozwiązaniem jest zwielokrotnienie także bazy danych zależnej. Niespójności danych rozwiązywane są w odrębnej aplikacji nadrzędnej. Baza danych może być specyficzna natomiast obsługa jej powinna pozostać abstrakcyjna. W tym celu stosuje się wzorzec repozytorium który chowa sposób wczytywania danych, usuwania oraz ogólnej komunikacji z systemem bazodanowym.

Jak w przypadku każdej aplikacji sieciowej musi ona oferować pewnego rodzaju komunikację, najpopularniejszym standardem jest Rest API lub szyny danych. Każde z rozwiązań posiada swoje zalety i wady w tym przypadku nie powstał złoty środek, który mógłby zostać zastosowany do wszystkich projektów mikroserwisowych. Metody obsługi zapytań (ang. *endpoints*) zazwyczaj

przechowywane są w kontrolerach. Kontrolery zajmują się wstępnym sprawdzeniem poprawności danych wejściowych oraz następnie przekazują informacje do logiki biznesowej. Logika biznesowa zajmuje się częścią obsługi zapytania, wprowadzania zmian w bazie oraz zwracanie wyniku działania.

Duża ilość komunikatów w tej architekturze oznacza także dużo oczekiwania aż wiadomość zostanie nadana a potem odebrana. Popularnym elementem w tym przypadku jest zastosowanie warstwy zapisywania danych tymczasowych (ang. *cache*). Jedynym wymaganiem jest, aby programista przekazał odpowiedni klucz. Pod kluczem przechowywana jest ostateczna odebrana wartość zapytania.

4.2. Platformy programistyczne low-code

Wysoki priorytet informatyzacji w wielu regionach codziennego życia takich jak edukacja, opieka zdrowotna, bankowość, e-commerce powoduje duże zapotrzebowanie na tworzenie nowych aplikacji bądź rozwijanie aktualnych. W przypadku, kiedy popyt jest większy niż podaż istnieją dwie strategie. Zwiększenie podaży, czyli ilości specjalistów i inżynierów IT lub druga, czyli zmniejszenie zapotrzebowania na programistów. Problem podaży występuje już od dziesiątek lat i potencjalnie będzie występować dalej w najbliższym czasie. Więcej firm decyduje się na drugie rozwiązanie problemu, wykorzystanie platform do skutecznego i szybkiego budowania aplikacji bądź procesów.

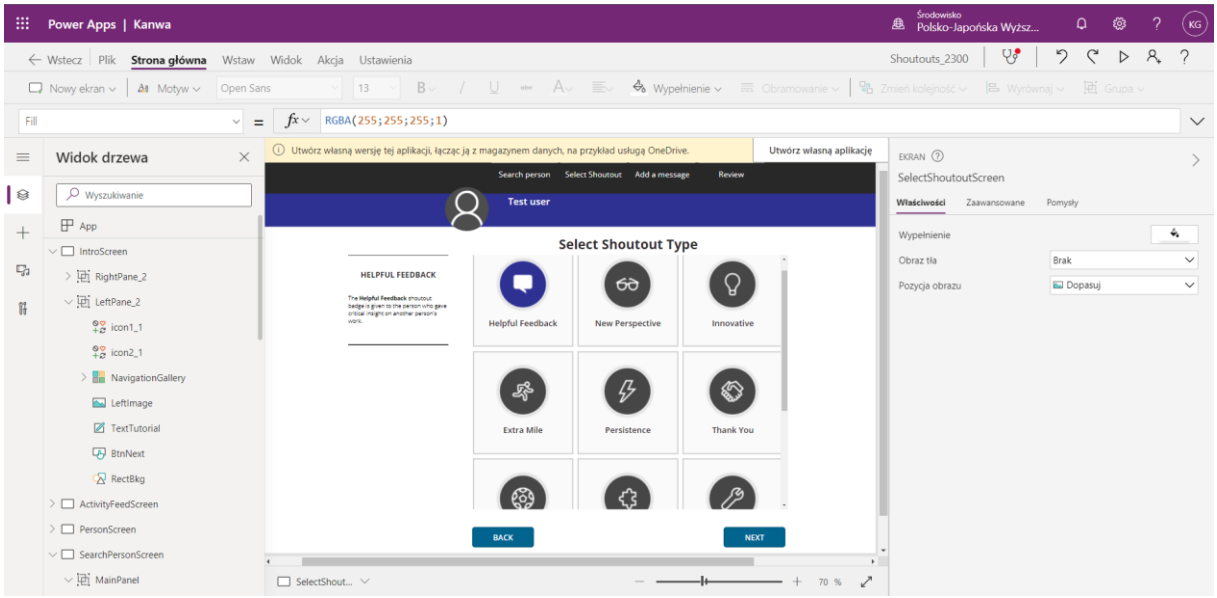
Definicja 3

Low-code development platform nazywamy platformą programistyczną używaną do tworzenia aplikacji przy użyciu małej ilości kodu lub interfejsu graficznego. Platformy tego typu tworzą aplikacje w pełni działające lub takie, które wymagają implementacji odpowiednich specyficznych procesów. Główną ideą jest redukcja czasu oraz kosztów wytworzenia aplikacji. Dodatkowym atutem jest możliwość wdrożenia większej ilości osób, które są w stanie pomóc w wytworzeniu narzędzia. Osoby nieposiadające umiejętności programowania oraz wdrażania aplikacji są w stanie utworzyć system gotowy do działania [26]. □

4.2.1 Przykłady zastosowania low-code

Platformy programistyczne, które umożliwiają budowanie kompletnych aplikacji posiadają wiele sposobów na integrację z usługami zewnętrznymi. W przypadku podstawowych procesów biznesowych np. wysyłanie maila na podstawie danych otrzymanych z innego serwisu, przydatne mogą być platformy automatyzacyjne np. Microsoft Automate lub IFTTT. Automatyzacje te posiadają ograniczoną możliwość konfiguracji.

Bardziej zaawansowanymi, ale wciąż przystępnymi dla użytkownika, który nie zna programowania, są platformy takie jak: Microsoft PowerApps, Mendix Studio, Google AppSheet [27]. Pozwalają one na tworzenie pełnych aplikacji na podstawie przygotowanych wcześniej komponentów. W przypadku aplikacji Microsoft PowerApps interfejs użytkownika kreatora jest zbliżony do innych produktów dostarczanych przez tą firmę co zostało przestawione na rysunku 7.



Rysunek 7. Widok edycji aplikacji przygotowanej w PowerApps – źródło: opracowanie własne

Uproszczony interfejs użytkownika jest potencjalną zaletą tego typu rozwiązań. Relatywnie wysoki próg zrozumienia działania takich edytorów może być potencjalną wadą tego typu aplikacji. PowerApps nie wymaga programowania, lecz posiada bardzo duże możliwości konfiguracji co może być przytłaczające dla osób mniej technicznych. W przypadku próby wykorzystania tego typu rozwiązań wraz z własnościowymi usługami dostępnymi wewnątrz firmy wymagane będzie utworzenie warstwy komunikacji np. używając protokołu HTTP.

Większość tego typu platform pozwala na utworzenie własnego komponentu. Po zdefiniowaniu wszystkich informacji dotyczących danych wejściowych, wyjściowych oraz autoryzacji, PowerApps pozwala na implementacje działania komponentu w języku programowania C# [28].

Klientami końcowymi generatorów low-code mogą być także sami programiści. Popularnymi zastosowaniami low-code wspomagającymi wytwarzanie oprogramowania są:

- generatory dokumentacji na podstawie komentarzy lub struktury klas i implementacji,
- generatory diagramów architektonicznych,
- generatory tworzące polecenia bazodanowe na podstawie utworzonego graficznego modelu danych,
- generatory układów wyglądu interfejsu użytkownika,
- generatory specyfikacji API na podstawie wystawionych *endpointów*,
- platformy automatyzacji budowania, testowania i publikowania aplikacji,
- środowiska testujące,
- platformy zarządzania kontenerami i wdrożeniem systemów,

4.2.2 JSON jako domenowy język konfiguracyjny

Popularnym rozwiązaniem zastępującym zastosowanie interfejsu graficznego na rzecz koncepcji low-code jest wykorzystanie konfiguracji zapisanej w wybranym języku. Interfejs graficzny większości generatorów posiada wewnętrzne tłumaczenie operacji wykonanej przez użytkownika na reprezentację tekstową. Przykładem tego typu tłumaczenia może być np. tłumaczenie umiejscowienia przycisku

w aplikacji graficznej na współrzędne x i y. Częstym zjawiskiem jest fakt, iż konfiguracja interfejsu graficznego posiada mniej możliwości niż modyfikacja samego pliku wynikowego.

Istnieje wiele dostępnych standardów do zapisywania danych w postaci wymienialnego pliku. Wymaganiem takiego standardu jest przejrzystość dla osoby modyfikującej i możliwość przechowywania danych obiektów zawierających pary klucz i wartość. Najpopularniejszymi standardami takich plików są: JSON, XML, YAML.

Główna popularność standardu JSON wzięła się z jego powiązania z językiem JavaScript, ponieważ jest on bazowany na jego podzbiorze. Pomimo powiązań, standard został stworzony jako niezależny od jakiegokolwiek języka programowania. Tworzenie i odczytywanie danych w tym formacie jest dostępne aktualnie w większości języków programowania [29]. Dzięki temu wykorzystanie standardu JSON jest najczęściej spowodowane prostotą integracji z aktualnie tworzonym oprogramowaniem.

Większość z plików konfiguracyjnych jest modyfikowane przez inżyniera korzystającego z aplikacji. W przypadku dużej złożoności konfiguracji szansa na błąd ludzki jest wyższa. Sam format JSON wymaga pewnych standardów zapisywania danych. Obsługiwane typy zapisanych danych to [29]:

- Obiekt – nieuporządkowany zbiór par klucz i wartość,
- Tablica – uporządkowany zbiór wartości,
- Wartość – przybiera formę wartości tekstowej, liczby, obiektu, tablicy, wartości `true` / `false` oraz `null`

W przypadku błędu, który wystąpi wewnątrz składni JSON nie będzie możliwe jego odczytanie. Sama specyfikacja tego standardu nie określa żadnych wymagań dotyczących jego długości bądź układu. Długi dokument może być podatny na błędy składni oraz struktury danych. Większość dostępnych standardów zapisywania danych posiada dodatkowy standard pisania schematów określających w jaki sposób powinna wyglądać struktura dokumentu.

Jednym z przykładów jest standard JSON Schema, który pozwala na adnotowanie i walidację struktury pliku danych. Elementy możliwe do konfiguracji w schemacie to [30]:

- typy danych,
- nazwy kluczy w obiektach oraz ich typy danych,
- ilość oraz rodzaj elementów w tablicach,
- unikalność elementów w tablicach,
- długość i wzorzec ciągów tekstowych,
- ograniczenie wartości numerycznych,
- określenie elementów jako wymagane,
- dane informacyjne dotyczące schematu,
- typy wyliczeniowe

Przykładowa zawartość pliku JSON Schema przygotowanego na rzecz prototypu została zaprezentowana na listingu 7.

Listing 7. Przykładowy skrócony schemat formatu JSON przygotowany dla prototypu generatora mikroserwisów – źródło: opracowanie własne

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
      "services": {
        "type": "array",
        "items": [
          {
            "type": "object",
            "properties": {
              "name": {
                "type": "string"
              },
              "shortName": {
                "type": "string"
              },
              "models": {
                "type": "array",
                "items": [...]
              },
              "endpoints": {
                "type": "array",
                "items": [...]
              },
              "dependencies": {
                "type": "array",
                "items": [...]
              }
            },
            "required": [
              "name",
              "shortName",
              "models",
              "endpoints",
              "dependencies"
            ]
          }
        ]
      }
    },
    "required": [
      "services"
    ]
}
```

4.3. Omówienie wymaganych parametrów od użytkownika

W przypadku każdego narzędzia *scaffoldującego* wymagane jest przekazanie intencji użytkownika, aby otrzymać oczekiwany rezultat. W przypadku sieci mikroserwisów istnieje wiele detali dotyczących pojedynczej aplikacji jak i całej infrastruktury.

4.3.1 Informacje ogólne

Każdego rodzaju aplikacja wymaga wstępnego zdefiniowania pewnych stałych. Trywialnym przykładem może być nazwa aplikacji. Sposób przechowywania danych może być odmienny i powinien być specyficzny dla domeny aplikacji. Przykładowo aplikacja zwierająca dane niezorganizowane może przechowywać dane w postaci dokumentowej w bazie danych typu *NoSQL*. Dane o stałej strukturze mogą oznaczać wykorzystanie przechowania danych w tabeli. Informacje ogólne mogą także zawierać właściwości dotyczące sposobu układu katalogów oraz systemu budowania projektu.

W przypadku większej ilości aplikacji wewnątrz infrastruktury mikroserwisowej ważnym elementem jest sposób komunikacji. Aplikacje obsługujące poszczególne protokoły wymagają zdefiniowania portów na które mogą być wysyłane komunikaty. Protokół HTTP domyślnie działa na porcie 80. Większa ilość aplikacji nie może wykorzystywać tego samego portu, oznaczałoby to konflikt. Ważnym jest zdefiniowanie unikalnych portów dla każdej aplikacji wewnątrz całej infrastruktury.

4.3.2 Opis zapisywanych danych

Każdego rodzaju aplikacja wykonuje operacje na danych, dlatego też podstawową konfiguracją jest określenie jakiego rodzaju dane będą przetwarzane oraz przechowywane. Każdy rodzaj danych powinien zostać odwzorowany jako encje. Na podstawie takiego rodzaju konfiguracji utworzona może zostać warstwa przechowania danych. Dopóki dane odczytywane oraz zapisywane są w odpowiedni sposób to nie ma znaczenia jak wygląda struktura oraz gdzie są przechowywane dane.

Modele w zależności od rodzaju danych mogą zawierać różną ilość właściwości. Ważną informacją dla generatora otrzymaną od użytkownika są nazwy oraz typy tych właściwości. Umożliwia to utworzenie odpowiedniej struktury oraz odpowiednie przechowywanie danych. Błędne zdefiniowanie właściwości może oznaczać błędy zapisu lub zapisane dane częściowe.

Dane posiadają pewną strukturę i mają pewne czynniki wiążące między sobą. Użytkownik oczekujący poprawnego odwzorowania danych powinien opisać relacje między poszczególnymi encjami. Istnieją trzy różne rodzaje relacji. Relacja jeden do wielu oznacza istnienie jednej encji, która może mieć wiele zależnych obiektów. Przykładowo użytkownik może mieć dużo zamówień. Relacja jeden do jednego oznacza powiązanie bezpośrednie pomiędzy encjami, przykładem może być kierowca oraz samochód. Relacja wiele do wielu oznacza istnienie wiele danych powiązanych różnymi wieloma danymi innego typu. Produkty zawarte w zamówieniu są przykładem tego typu relacji. Produkty mogą być w wielu różnych zamówieniach oraz zamówienia mogą zawierać wiele różnych produktów.

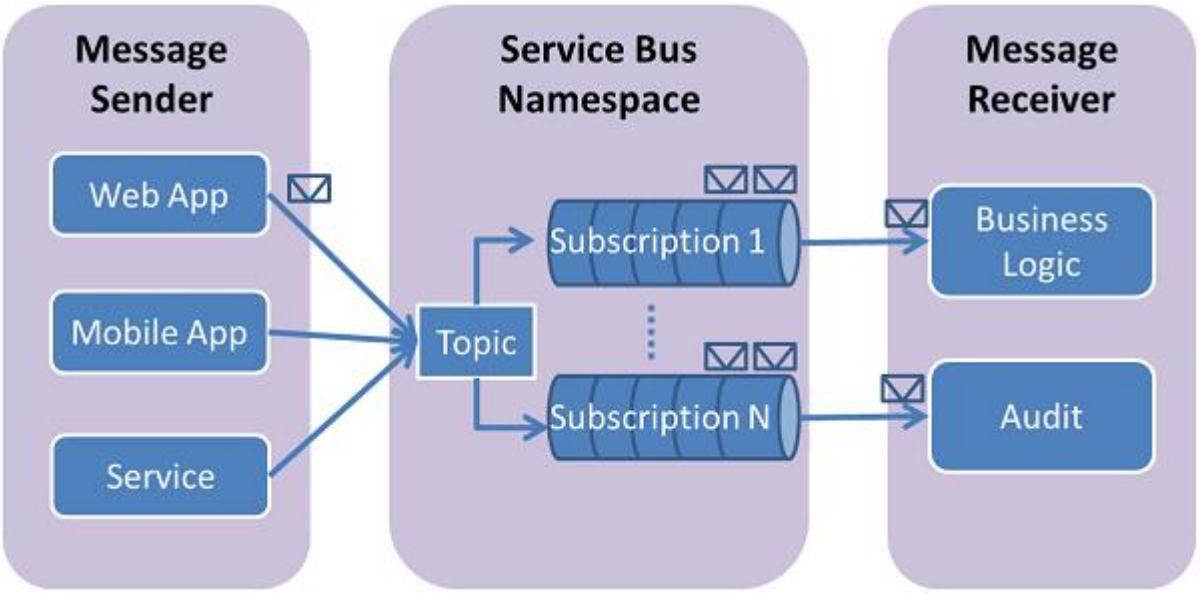
4.3.3 Definicja komunikacji z innymi mikroserwisami

Kluczowym elementem mikroserwisów jest zarządzanie wewnętrznym systemem wymiany komunikatów. Istnieje wiele rozwiązań tego problemu dostępnych także jako usługi chmurowe np. Azure Service Bus. W przypadku mikroserwisów najpopularniejsze ze stylów komunikacji asynchronicznej są [25]:

- Zapytanie/Odpowiedź – usługa wysyła zapytanie do odbiorcy i oczekuje natychmiast odpowiedzi
- Powiadomienia – nadawca wysyła wiadomość odbiorcy, ale nie oczekuje odpowiedzi,

- Zapytanie/Odpowiedź asynchroniczna – usługa wysyła zapytanie do odbiorcy i spodziewa się odpowiedzi w przyszłości,
- Publikuj/Subskrybuj – usługa publikuje wiadomość dla zera lub większej ilości subskrybentów,
- Publikuj/Odpowiedź asynchroniczna – usługa publikuje zapytanie do jednego lub większej ilości odbiorców, z których część z odbiorców wysyła odpowiedź

Na rysunku 8 zaprezentowane zostało działanie komunikacji Publikuj/Subskrybuj na przykładzie Azure Service Bus. Wybrany Nadawca wiadomości (ang. *Message Sender*) publikuje wiadomość na dany *Temat* (ang. *Topic*). Zarządca wewnątrz Service Bus tworzy duplikaty otrzymanej wiadomości i rozsyła je do zarejestrowanych subskrybentów.



Rysunek 8. Przykład komunikacji Publikuj/Subskrybuj przy użyciu Azure Service Bus – źródło: [31]

Podczas implementacji mikroserwisów programista powinien zdecydować się na styl, który najbardziej odpowiada wymaganiu biznesowemu. Każdy ze sposobów posiada swoje wady oraz coraz większe skomplikowanie implementacji. Zalecane jest wykorzystywanie jednego sposobu komunikacji, lecz nie jest to ograniczenie.

Mikroserwis który oferuje komunikacje opartą o zapytania HTTP nie będzie w stanie komunikować się z mikroserwisem wykorzystującym szynę komunikacyjną. Popularnym sposobem obejścia tego problemu jest utworzenie dodatkowej aplikacji tłumaczącej zapytania. Aplikacje pośredniczące często są stosowane w celu umożliwienia komunikacji starych systemów opartych przykładowo na protokole SOAP wraz z całą infrastrukturą mikroserwisów. Aplikacje pośredniczące mogą także tłumaczyć rodzaj komunikacji. Przykładowo wysyłane powiadomienia nie wymagają zdefiniowanych odbiorców. W przypadku komunikacji zapytanie / odpowiedź wymagane jest określenie jakie dane są wysyłane, gdzie oraz jaka jest odpowiedź wysyłana do nadawcy. W takim przypadku aplikacje pośredniczące dostosują rodzaj komunikacji do tego który jest obsługiwany w aplikacji docelowej.

4.4. Wynik działania *scaffoldingu*

Podstawowym zadaniem każdego generatora jest przetłumaczenie wymagań użytkownika na docelowy oczekiwany wynik. Powinien on być pozbawiony jakichkolwiek zbędnych plików i danych pomocniczych. Generator posiadając dane wyczytane z konfiguracji jest świadomy wymagania aplikacji, wymaganych zależności oraz struktury. *Scaffolding* powinien pomagać programiście w automatyzacji tworzenia podstaw projektu, dlatego też każda aplikacja powinna być odrębna i posiadać możliwość łatwej modyfikacji. Struktura plików i katalogów powinna być wzorowana na standardach budowania aplikacji z danej dziedziny.

Najpopularniejszym sposobem podziału struktury projektu na platformie ASP .NET jest wydzielenie katalogów dla każdej dziedziny. Przykładowymi typami klas, które można znaleźć w projekcie to:

- Kontrolery – zawierające metody obsługujące zapytania zewnętrzne,
- Modele – będące wiernym odwzorowaniem struktury warstwy danych,
- Infrastruktura – klasy będące opakowaniami na zależności zewnętrzne takie jak baza danych, mikroserwis,
- DTO (ang. *Data transfer object*) – klasy kontraktu komunikacji przychodzącej oraz wychodzącej wykorzystywanej do komunikacji z usługami zewnętrznymi,
- Usługi – będące klasami wykonującymi logikę biznesową danej aplikacji,
- Migracje – klasy, które określają wymagane operacje na kontekście bazy danych, aby otrzymać aktualną strukturę modeli,

Klasy tworzone w określonych dziedzinach powinny mieć podobną strukturę oraz pojedynczą odpowiedzialność. Pomaga to w łatwości utrzymania porządku i obsługi technicznej. Prototyp generatora stara się odwzorować te ogólnie ustalone zasady. Na listingu 8 został przedstawiony układ katalogów i plików w wygenerowanym mikroserwisie. Każda z klas służy w jednym celu i jest przechowywana w odpowiednim katalogu.

Listing 8. Wynik polecenia `tree` dla utworzonego generatorem mikroserwisu – źródło: opracowanie własne

```
ShopCartService
│   appsettings.Development.json
│   appsettings.json
│   Dockerfile
│   Program.cs
│   ShopCartService.csproj
│   ShopCartServiceContext.db
│
├───.config
│       dotnet-tools.json
│
├───Controllers
│       ShopCartServiceController.cs
│
├───Dtos
│       Order.cs
│       OrderProduct.cs
│       Product.cs
│
├───Infrastructure
│       IOrderService.cs
│       IProductService.cs
│       ShopCartServiceContext.cs
│
├───Migrations
│       20220330090638_InitialCreate.cs
│       20220330090638_InitialCreate.Designer.cs
│       ShopCartServiceContextModelSnapshot.cs
│
├───Models
│       Cart.cs
│       CartProduct.cs
│
└───Properties
        launchSettings.json
```

Jedyne usprawnienie struktury można dokonać po podzieleniu kontrolera na pliki specyficzne dla danego modelu. Tworzony kontroler posiada wszystkie możliwe metody obsługi zapytań. W tym przypadku długość pliku może szybko wzrastać przy dużej ilości *endpointów*.

Język C# pozwala na rozdzielenie implementacji klas na wiele plików. Każda z generowanych klas posiada atrybut partial. Umożliwia to programiście na dopisywanie własnego kodu do nowych plików zamiast modyfikować te wygenerowane.

5. Prototyp generatora mikroserwisów

Rozdział ten zawiera opis zakładanych wymagań do spełnienia prototypem utworzonym na rzecz tej pracy. Zostały także opisane możliwości argumentów wiersza poleceń oraz poszczególne ważniejsze funkcjonalności. Do zaprezentowania działania funkcjonalności wykorzystano pseudokod opisujący kroki algorytmów. Założeniem całego generatora jest możliwość utworzenia odrębnych aplikacji mikroserwisowych które mogą wymieniać informacje między sobą. W przyszłości narzędzie powinno zostać urozmaicone obsługą dodatkowych języków oraz baz danych.

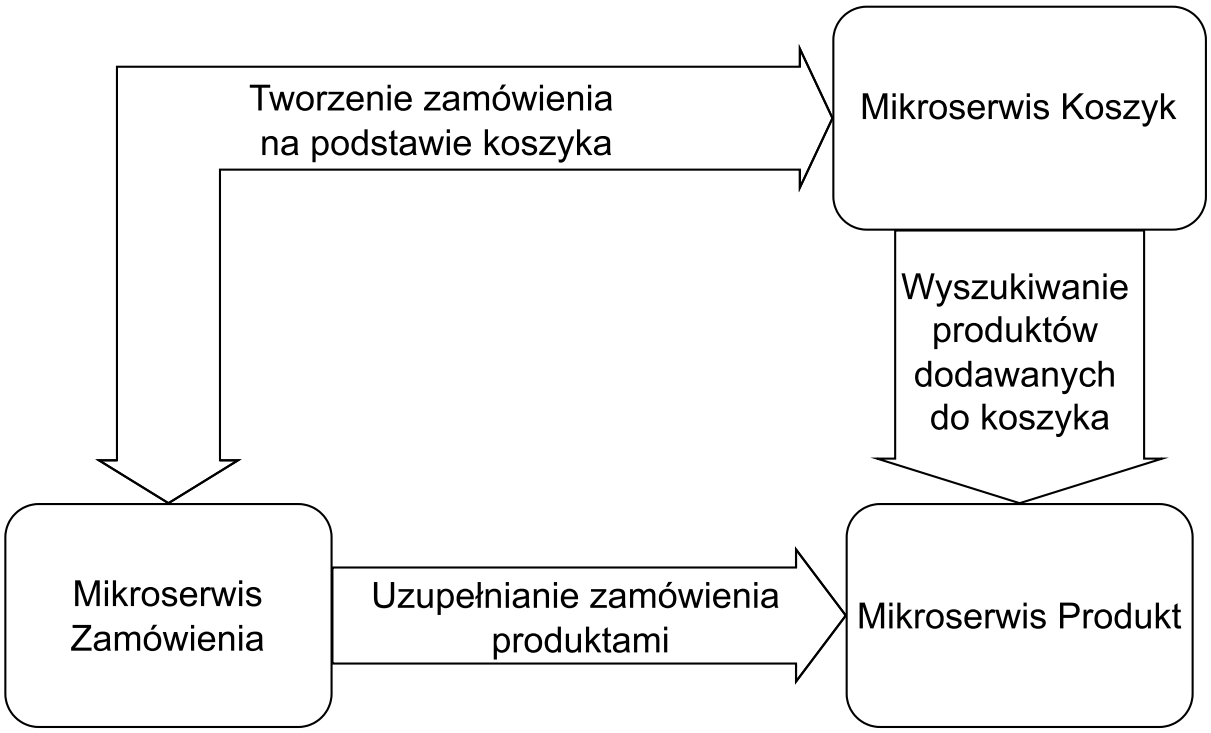
5.1. Wymagania

W przypadku każdego typu *scaffoldingu* głównym wymaganiem jest stabilność działania w zakresie oferowanych funkcjonalności. Podczas generowania mikroserwisów, istnieje duża ilość potencjalnych czynników wpływających na niepowodzenie. Problem podczas generowania jednego mikroserwisu zazwyczaj spowoduje problem utworzenia kolejnego. Powodem jest duża ilość powiązań między projektami i wartościami przekazywanymi między nimi. Inne najczęstsze problemy mogą wystąpić w przypadku przygotowania niepoprawnej konfiguracji generatora. Jednym z ważniejszych wymagań jest odpowiednie raportowanie problemów i informowanie użytkownika o postępie generowania aplikacji.

Drugim wymaganiem jest zweryfikowanie działania generatora na przykładowym projekcie. W celu spełnienia wymagania, prototyp powinien utworzyć w pełni działającą komunikację między dwoma lub większa ilością mikroserwisów.

Przykładowy projekt zawierający trzy mikroserwisy został przedstawiony na rysunku 9.

- Mikroserwis Produkt - jest najbardziej podstawowy, umożliwia dodawanie, usuwanie, modyfikację oraz odczyt danych zapisanych w wewnętrznej bazie danych,
- Mikroserwis Koszyk – pozwala na tworzenie koszyka oraz dodawanie do niego produktów, po wysłaniu zapytania na *endpoint* `checkout` przesyła koszyk z produktami do mikroserwisu Zamówienia, który tworzy zamówienie,
- Mikroserwis Zamówienia – obsługuje zapytanie utworzenia zamówienia z koszyka oraz wyświetla utworzone zamówienia



Rysunek 9. Diagram wymaganej struktury oraz opis komunikacji między mikroserwisami – źródło: opracowanie własne

Trzecim wymaganiem jest utworzenie kontenerów dla każdego wygenerowanego projektu. Umożliwi to wykorzystanie Docker compose w celu uruchomienia wszystkich mikroserwisów obok siebie.

Ostatnim wymaganiem jest wykorzystanie technologii Swagger. Interfejs graficzny Swagger UI pozwala na wyświetlenie dostępnych *endpointów*, dostępnych modeli oraz przykładowych zawartości zapytań. Interfejs pozwala także na wykonanie testowego wysłania wybranej wiadomości. Standard OpenAPI dostarczany wraz z Swaggerem może zostać zaimportowany do aplikacji zewnętrznej Postman.

5.2. Opis funkcjonalności

Przygotowany prototyp został dostosowany do spełnienia założonych wymagań. Pozwala na stworzenie działających projektów od zera. Konfiguracja w formacie JSON jest wstępnie weryfikowana pod względem semantycznym. W celu utworzenia pełnej weryfikację konfiguracji wejściowej należałoby dodać sprawdzanie logiki i zależności.

5.2.1 Początkowa weryfikacja

Oprogramowanie *scaffoldujące* zazwyczaj przyjmuje konfigurację na początku i wykonuje działanie, aż do utworzenia projektów. Prototyp także zaczyna swoją pracę od wstępnej weryfikacji ustawień. Sprawdzane są:

1. parametry wejściowe oprogramowania,
2. poprawność zapisanego pliku konfiguracyjnego,
3. poprawność strukturalna przy pomocy JSON Schema

Argumenty wejściowe mogą zostać wykorzystane przez programistę do konfiguracji dodatkowej aplikacji. Na listingu 9 pokazano wynik uruchomienia generatora z atrybutem -h, wynik zawiera pomoc dotyczącą wszystkich dostępnych opcji oraz przykładowe wartości. Parametry możliwe do skonfigurowania to:

- `-h, --help` - Wyświetlenie pomocy przedstawionej na listingu 9.
- `--configPath` - Ścieżka do pliku konfiguracyjnego JSON.
- `--output-path` - Ścieżka do katalogu, w którym zostaną utworzone projekty.
- `--clean-output` - Ustawienie czy katalog, w którym utworzone zostaną projekty powinien być czyszczony przed uruchomieniem. Wartości możliwe do ustawienia: *`true`*, *`false`*.
- `--single-service` - Możliwość wygenerowania jednego mikroserwisu z wszystkich dostępnych w pliku konfiguracyjnym JSON. Argument wyczytuje nazwę mikroserwisu podaną przez użytkownika.
- `--start-services` - Definiuje, czy usługi powinny zostać uruchomione po zakończonym generowaniu. Wartości możliwe do ustawienia: *`true`*, *`false`*.
- `--dry-run` – Określa czy uruchomienie generatora powinno wykonywać komendy i dokonywać zmian w systemie plików. Przydatny tryb do weryfikowania poprawności konfiguracji. Wartości możliwe do ustawienia: *`true`*, *`false`*.
- `--log-level` – Informuje generator jak zaawansowane informacje powinien wyświetlać użytkownikowi. Wartości możliwe do ustawienia: *`CRITICAL`*, *`FATAL`*, *`ERROR`*, *`WARNING`*, *`INFO`*, *`DEBUG`*.

Listing 9. Lista dostępnych opcji generatora wyświetlana po wykonaniu polecenia `python connectis.py -h` – źródło: opracowanie własne

```
usage: connectis.py [-h] [--configPath CFGPATH] [--output-path OUTPUTPATH] [--single-service
SINGLESERVICE] [--clean-output CLEANOUTPUT] [--start-services STARTSERVICES] [--dry-run DRYRUN] [--
log-level LOGLEVEL]


Prototype of application that is scaffolding microservices based on configuration file.


options:

 -h, --help                  show this help message and exit

 --configPath CFGPATH  Config file path. Default "config.json"

 --output-path OUTPUTPATH

                Output path where projects will be stored. Default "output"

 --single-service SINGLESERVICE

                Create only single service, provide name

 --clean-output CLEANOUTPUT

                Clean output directory before start. Default: true

 --start-services STARTSERVICES

                Start all services after building them. Default: true

 --dry-run DRYRUN      Running application without doing any changes. Default: false

 --log-level LOGLEVEL  Log level: ['CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG',
'NOTSET']. Default: INFO
```

5.2.2 Utworzenie projektów

Mikroserwisy tworzone są na bazie podstawowego projektu dostarczanego z narzędzia .NET CLI. Polecenie `dotnet new web -o NazwaMikroserwisu` buduje podstawową strukturę plików potrzebnych do późniejszej modyfikacji. Głównym elementem projektu jest plik o rozszerzeniu *csproj* przedstawiony na listingu 10, zawiera on konfigurację SDK (ang. *Software Development Kit*) i wersję środowiska, w którym aplikacja może zostać uruchomiona. Pliki `appsettings.json` i `appsettings.Development.json` zawierają konfigurację, która jest mapowana do obiektów podczas uruchamiania aplikacji.

Listing 10. Plik *csproj* utworzonego projektu przy pomocy narzędzia .NET CLI – źródło: opracowanie własne

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Plik źródłowy `Program.cs` zawiera definicje budujące serwis internetowy do obsługi zapytań HTTP. Na listingu 11 przedstawiono zawartość pliku, najbardziej bazowa wersja pliku zawiera tylko 4 linie kodu.

Listing 11. Podstawowy plik `Program.cs` aplikacji ASP .NET – źródło: opracowanie własne

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Plik ten można podzielić na dwie grupy. Inicjalizacja aplikacji oraz konfiguracja przetwarzania potokowego każdego zapytania.

Inicjalizacja aplikacji jest kodem wykonywanym przed uruchomieniem polecenia `builder.Build();`. W tej grupie definiowane są usługi, fabryki oraz kontrolery. Każda utworzona klasa może zostać zarejestrowana w kolekcji usług. W zależności od użytej metody zmienia się okres istnienia zarejestrowanej usługi. W celu zarejestrowania usługi w systemie budującym serwis internetowy na obiekcie klasy `WebApplicationBuilder` należy uruchomić metodę: `builder.AddScoped<TypKlasyRejestrowanej>();`. Zarejestrowana w ten sposób usługa jest dostępna tylko na czas wykonywania zapytania. Klasa zostanie usunięta z pamięci po wysłaniu odpowiedzi do użytkownika [32]. Rejestrowane interfejsy i ich implementacje będą wstrzykiwane do kontrolerów i każdej innej zarejestrowanej usługi. Pomaga to w rozdzieleniu budowanej aplikacji na mniejsze klasy. Dodatkowo przyspieszone jest testowanie oraz klasy mogą być łatwiej podmieniane atrapami.

Warstwy pośredniczące (ang. *middleware)* są obsługiwane podczas przetwarzania potokowego w sytuacji odebrania zapytania. Warstwa decyduje, czy wiadomość powinna zostać przekazana do kolejnej warstwy przetwarzania potokowego, może ona także wykonywać pracę przed i po następnej warstwie. Przetwarzanie potokowe pozwala na zarządzanie zachowaniem aplikacji na różnorodne rodzaje zapytań. Przykładem tego typu manipulacji mogą być tłumaczenie odpowiedzi, obsługa plików lub autoryzacja.

Prototyp modyfikuje plik `Program.cs`, aby rozszerzyć działanie mikroserwisu oraz zachować strukturę czystości kodu. Dodawane są dodatkowe usługi bazy danych, infrastruktury komunikacji zewnętrznej oraz kontrolery.

5.2.3 Utworzenie warstwy struktury danych

Jednym z głównych sposobów przechowywania danych w aplikacjach internetowych jest wykorzystanie bazy danych. Głównym elementem decydującym o wyborze systemu warstwy danych jest domena działania mikroserwisu. Konteneryzacja i systemy zarządzające konteneryzacją mogą powodować sytuację, gdy aplikacja jest uruchomiona wielokrotnie obok siebie. W celu dobrania odpowiedniego sposobu przechowywania danych należy wziąć wyżej wymieniony czynnik pod uwagę.

Na rzecz prototypu zdecydowano zastosowanie najprostszego systemu bazy danych SQLite. Wszystkie dane są przechowywane wewnątrz pliku znajdującego się wewnątrz kontenera. Jest to mało optymalne rozwiązanie. Wielkość kontenera nie powinna być zależna od danych przechowywanych wewnątrz aplikacji. Zalecane jest utworzenie wolumenów i przechowywanie pliku bazy danych wewnątrz systemu plików gospodarza kontenerów.

Warstwa struktury danych jest tworzona na podstawie modeli otrzymanych z konfiguracji. Tworzone są klasy na podstawie wyspecyfikowanych przez użytkownika informacji oraz modele pośrednie wymagane np. w relacji jeden do wielu, wiele do wielu.

Wymaganą klasą do obsługi bazy danych jest kontekst. Przykładowy kontekst wygenerowany dla mikroserwisu Zamówienia został przedstawiony na listingu 12. Kontekst jest dostarczany przez Entity Framework Core jako warstwa abstrakcji na systemie bazodanowym. Kontekst zawiera generyczne kolekcje `DbSet`, które są odwzorowaniem przechowywanych danych w systemie bazodanowym. Dodatkowe metody konfiguracyjne używane są do wskazania informacji na temat sterownika bazy danych oraz relacji między modelami. Na podstawie klasy kontekstu Entity Framework tworzy migracje oraz dokonuje aktualizacji struktury bazy danych.

Listing 12. Wygenerowana klasa kontekstu dla mikroserwisu Zamówienia – źródło: opracowanie własne

```
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using OrderService.Models;

namespace OrderService.Infrastructure;

public partial class OrderServiceContext : DbContext {

    public DbSet<OrderProduct> OrderProducts => Set<OrderProduct>();

    public DbSet<Order> Orders => Set<Order>();

    public string DbPath { get; }

    public OrderServiceContext()
    {
        DbPath = System.IO.Path.Join(".", "OrderServiceContext.db");
    }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite($"Data Source={DbPath}");
}
```

Problem danych zapisywanych wewnątrz kontenera aplikacji zostanie usunięty w przypadku zastosowania innej bazy danych np. PostgreSQL. System bazodanowy mógłby zostać uruchomiony w zewnętrznym kontenerze, obok aplikacji mikroserwisowej. Rozwiązanie to zostało pominięte w prototypie z powodu utrudnionego wykonywania migracji na bazie danych.

5.2.4 Utworzenie warstwy komunikacji

Najtrudniejszą funkcjonalnością jest zapewnienie komunikacji wychodzącej do pozostałych mikroserwisów oraz komunikacji przychodzącej zapisanych w kontrolerze.

Wygenerowany na podstawie konfiguracji kontroler zawiera metody obsługujące zapytania. Każda z metod może posiadać innego rodzaju logikę obsługi zapytania, np. dodanie otrzymanego modelu do bazy lub przesłanie modelu do innego mikroserwisu. Przedstawiona na listingu 13 wygenerowana metoda zawiera logikę `mappedPost`. Rodzaj logiki, ciało zapytania otrzymywanego oraz ścieżka URI została wyczytana z konfiguracji. Ciało logiki jest tworzone na podstawie kontekstu aplikacji, *endpointu* oraz typu logiki wyczytanego z pliku JSON. `MappedPost` zamienia wejściowy typ danych na obiekt o typie danych obsługiwanych w danym mikroserwisie.

Listing 13. Utworzona metoda obsługująca zapytanie typu `mappedPost` – źródło: opracowanie własne

```
[HttpPost("order/create")]
public async Task<ActionResult<Order>> createOrder(Cart body)
{
    var mapped = new Order{
        OrderProduct =
            body.CartProduct.Select(x =>
                new OrderProduct{
                    ProductId = x.ProductId
                }).ToList(),};
    orderservicecontext.Orders.Add(mapped);
    await orderservicecontext.SaveChangesAsync();
    return Ok(mapped);
}
```

Komunikacja wychodząca może zostać utworzona na podstawie informacji o utworzonych metodach kontrolera innych mikroserwisów. Generator wie o istnieniu wszystkich mikroserwisów oraz ich API. Te informacje wystarczą do utworzenia interfejsu i odpowiednich metod z adnotacjami. Prototyp nie musi tworzyć implementacji, ponieważ wykorzystana biblioteka Refit wygeneruje go podczas kompilacji projektu.

5.2.5 Przygotowanie skryptów konteneryzacji

Środowisko Docker pozwala na tworzenie kontenerów na podstawie obrazów. Obrazy dostępne w publicznych repozytoriach mogą zostać uruchomione ręcznie lub wykorzystane jako podstawa do budowy innych obrazów. Projekty tworzone na platformie .NET mogą wykorzystać obraz `mcr.microsoft.com/dotnet/aspnet:6.0` jako podstawę do uruchomienia aplikacji.

`Dockerfile` wygenerowany dla każdego projektu zawiera instrukcję tworzenia obrazu aplikacji. Wygenerowany plik przedstawiony na listingu 14 został podzielony na grupy `base`, `build`, `publish`, `final`. W celu optymalizacji wielkości kontenera projekt jest budowany w grupach `build`

lub `publish`. Następnie pliki mogą zostać przeniesione na mniejszy obraz `base`. Grupy `build` oraz `publish` bazują na obrazie `mcr.microsoft.com/dotnet/sdk:6.0` który zawiera dodatkowe narzędzia kompilacji.

Kontenery domyślnie udostępniają na zewnątrz porty TCP (ang. *Transmission Control Protocol*) w tym standardowy port HTTP 80 oraz port do komunikacji HTTPS 443.

Listing 14. `Dockerfile` mikroserwisu Koszyk – źródło: opracowanie własne

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["ShopCartService.csproj", "."]
RUN dotnet restore "./ShopCartService.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "ShopCartService.csproj" -c Release -o /app/build

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

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

5.2.6 Uruchomienie kontenerów

Generator pozwala opcjonalnie na uruchomienie kontenerów po ich wygenerowaniu. Przygotowany plik *docker-compose.yml* zawiera informacje o wszystkich mikroserwisach. Najważniejszą informacją dla użytkownika jest wartość portu przekierowanego. Port lokalny jest odwzorowaniem portu 80 wewnętrznego kontenera. Użytkownik może wykorzystać port przekierowany do rozpoczęcia komunikacji z mikroserwisem. Przekierowane porty zaczynają się od wartości 8000. Na listingu 15 przedstawiono przykładowe kontenery `ProductService` oraz `ShopCartService`. W celu wysłania wiadomości na port 80 mikroserwisu `ShopCartService`, użytkownik musi wysłać wiadomość na port 8001.

Listing 15. Fragment pliku docker-compose.yml – źródło: opracowanie własne

```
services:
  productservice:
    build: ProductService
    ports:
        - "8000:80"
  shopcartservice:
    build: ShopCartService
    ports:
        - "8001:80"
```

5.3. Wymagana konfiguracja

Na rzecz utworzonego generatora wykorzystano format JSON w celu zdefiniowania wymaganych parametrów zamienianych w gotowe projekty. Umożliwiło to stworzenie notacji, która zawiera informacje dotyczące struktury, dostępnych modeli, relacji między modelami oraz sposobu komunikacji. Utworzono w ten sposób DSL (ang. *domain-specific language*) który jest wyspecjalizowany w domenie opisywania struktury i zależności mikroserwisów. Przygotowany plik JSON Schema, którego fragment został przedstawiony na listingu 7 pozwala na weryfikację poprawności syntaktycznej przygotowanej konfiguracji. Poprawność semantyczna jest weryfikowana wewnątrz prototypu.

Aktualna struktura notacji jest wystarczająca do przygotowanego prototypu, lecz posiada ograniczone możliwości modelowania mikroserwisów. W celu umożliwienia większej elastyczności należałoby rozwinąć notację o:

- obsługę języka w jakim tworzony jest mikroserwis,
- informacje dotyczące sposobu przechowywania danych,
- adnotacje autoryzacyjne,
- definicję obsługi zewnętrznych zależności

5.3.1 Definicje struktury modeli oraz relacji

Utworzona notacja pozwala na wykorzystanie większości popularnych typów dostępnych w językach programowania:

- `guid` – typ globalnie unikalnego identyfikatora w postaci tekstu
- `int` – typ całkowity liczbowy
- `string` – typ tekstowy
- `decimal` – typ o postaci liczby dziesiętnej
- `list<typ_przechowywany>` - lista elementów o wskazanym typie
- `Ref` – typ referencyjny który pozwala na odwołanie się do innej stworzonej encji

Na listingu 16 został przedstawiony przykład modelu *Produkt*, który może być reprezentacją istniejącego produktu w sklepie. Encja utworzona na podstawie podanej konfiguracji będzie zawierała unikalny identyfikator, tytuł o typie tekstowym oraz cenę zapisaną w postaci liczby dziesiętnej.

Listing 16. Definicja modelu Produkt – źródło: opracowanie własne

```
{
    "modelName": "Produkt",
    "properties": [
        {
            "type": "guid",
            "name": "Id"
        },
        {
            "type": "string",
            "name": "Tytul"
        },
        {
            "type": "decimal",
            "name": "Cena"
        }
    ],
    "modelReference": {
        "name": "ProduktRef",
        "propertyName": "Id"
    }
}
```

W celu umożliwienia odwołania się do danego modelu podczas tworzenia relacji jeden do wielu lub wiele do wielu należy zdefiniować dodatkowy obiekt przypisany do właściwości `modelReference`. Pierwszą wartością tego obiektu jest nazwa zgłaszanego identyfikatora modelu. Powinien to być unikalny identyfikator wewnątrz konfiguracji. Nazwa ta może zostać wykorzystana jako powiązanie z modelem z innego mikroserwisu. Druga wartość przypisana do `propertyName` zawiera informację, która z dostępnych właściwości encji zdefiniowanych w liście `properties` może zostać wykorzystana jako identyfikator odwołania.

5.3.2 Konfiguracja obsługiwanych *endpointów*

W przypadku generatora mikroserwisów zastosowano najprostszy styl komunikacji między mikroserwisami. Zapytania i odpowiedzi są wysyłane za pomocą protokołu HTTP. Generator spodziewa się, że każdy odbiorca jest w danym momencie dostępny oraz jest w stanie odpowiedzieć na zapytanie innego mikroserwisu. Opis podstawowego odbiorcy zapytania (*endpointu*) został przedstawiony na listingu 17.

Listing 17. Definicja konfiguracji odbiorcy zapytania – źródło: opracowanie własne

```
{
    "type": "get",
    "method": "get",
    "name": "wezKoszyk",
    "uri": "cart/{Id:guid}",
    "body": "none"
}
```

Dostępne właściwości wewnątrz konfiguracji pozwalają na określenie większości informacji dotyczących sposobu obsługi zapytania.

Właściwość `type` ma za zadanie poinstruowanie generatora w jaki sposób ciało metody obsługującej zapytania powinno zostać utworzone. W aktualnej implementacji dostępne jest osiem różnych sposobów obsługi komunikatów. Lista dostępnych wartości tego atrybutu prezentuje się następująco:

- `get` – zwracanie jednego obiektu encji według otrzymanego identyfikatora,
- `getAll` – zwracanie wszystkich modeli danego typu,
- `post` – tworzenie nowego modelu,
- `put` – aktualizacja modelu,
- `delete` – usunięcie modelu,
- `call` – wysłanie obiektu aktualnego mikroserwisu do metody obsługującej zapytanie innego mikroserwisu,
- `putReferenced` – utworzenie relacji między wysłanym modelem a modelem dostępnym w warstwie danych,
- `mappedPost` – odebranie modelu wejściowego i tłumaczenie na model dostępny wewnątrz mikroserwisu

Komunikacja HTTP wymaga określenia sposobu wykonywanego zapytania. W przypadku tworzonych metod obsługi zapytania typ obsługi nie oznacza zawsze metody HTTP wykorzystywanej do komunikacji. Właściwość konfiguracji `method` umożliwia na określenie w jaki sposób komunikaty powinny być odbierane. Obsługiwane wartości to: *POST*, *GET*, *PUT*, *DELETE*.

Właściwość `name` oznacza nazwę metody obsługi zapytania. Wykorzystywana jest w przypadku tworzonej dokumentacji Swagger oraz wewnątrz tworzonego kodu.

Następnym elementem konfiguracji jest `uri`. Zapisane wewnątrz informacje są bardzo ważne. Przetworzona ścieżka URI (ang. *Uniform Resource Identifier*) pomaga w zrozumieniu działania *endpointu*. Zawarte wewnątrz ścieżki klamry zawierają informacje na temat parametrów oraz ich nazw. Przykładem takiego typu parametru jest `{Id:guid}`. Przetworzona ścieżka informuje generator o tym, że powinien dodatkowo utworzyć parametr generowanej metody o nazwie `Id` i typie `guid`. Aktualnie ilość atrybutów nie jest ograniczona. Wygenerowana metoda obsługi zapytania wykorzysta tylko te które są potrzebne.

`Body` jest atrybutem, który określa informację jakiego ciała komunikatu generowana metoda powinna się spodziewać. W przypadku typu `post`, który służy do obsługi dodawania obiektu ciałem powinien być model zdefiniowany w danym mikroserwisie. `Body` może także przyjmować jako wartość

odwołania do modelów innych mikroserwisów. Może zostać to wykorzystane do komunikacji między mikroserwisowej.

W sytuacji generowania metody typu `call` wymagane jest dodanie dodatkowej właściwości o takiej samej nazwie. Wymagane jest zdefiniowanie informacji na temat nazwy *endpointu* odbierającego oraz nazwy mikroserwisu w którym ta metoda się znajduje. *Endpoint* o typie `call` przesyła obiekt dostępny w bazie do mikroserwisu który jest odbiorcą i ma zdefiniowany typ `mappedPost`.

5.3.3 Informacja o zależnościach

W przypadku konfiguracji zawierającej więcej niż jeden mikroserwis wymagane jest określenie zależności. Każde odwołanie do innego mikroserwisu przy pomocy modelu referencyjnego lub *endpointu* wymaga dopisania do listy nazwy zależności źródłowej. Przykładem jest mikroserwis zamówień, którego lista dependencji została przedstawiona na listingu 18. Do utworzenia relacji między modelami jeden do wielu potrzebuje encji Produkt z zależności `ProduktMikroserwis` oraz modelu `Koszyk` do obsługi *endpointu* tworzenia zamówienia.

Listing 18. Lista zależności mikroserwisu Zamówienia – źródło: opracowanie własne

```
"dependencies": [
    "ProduktMikroserwis",
    "KoszykMikroserwis"
]
```

5.4. Generowanie mikroserwisów

Pomyślne wczytanie konfiguracji oznacza dla prototypu rozpoczęcie procesu analizy i grupowania projektów. Mikroserwisy zawierające duże ilości zależności muszą zostać utworzone najpóźniej, te które jeszcze nie zostały wygenerowane a wymagają innych usług także nie wygenerowanych mają zależność kołową.

Definicja 4

Zależność kołowa (ang. *circular dependency*) nazywamy relacją między dwoma lub większą ilością modułów, które bezpośrednio lub pośrednio zależą od siebie nawzajem, aby działać poprawnie [33]. □

W odpowiedni sposób do przydzielonej grupy aplikacje są *scaffoldowane*. Utworzone projekty zawierające wszystkie komponenty są zapisywane jako obrazy Docker i uruchamiane przy pomocy Docker compose.

Aktualna implementacja prototypu wykorzystuje narzędzia SDK dostępne na maszynie lokalnej w celu wygenerowania podstawowego projektu, bazy danych lub kompilacji. Każda operacja na SDK została schowana za abstrakcyjną klasą. Umożliwi to w przyszłości dodanie implementacji obsługi narzędzi zewnętrznych dostarczanych z w kontenerze. Abstrakcja tej logiki dodatkowo umożliwia tryb `dry-run` generatora, który wyłącza wprowadzanie zmian w systemie plików. Jest to tryb testowy dla konfiguracji wprowadzonej przez użytkownika.

5.4.1 Rozwiązywanie zależności

Generowane mikroserwisy mogą być się dzielić na bazowe (ang. *base*) zawierające proste zależności oraz kompleksowe (ang. *complex*) zawierające zależności kołowe. Pierwszym krokiem

wykonywanym po wyczytaniu w konfiguracji jest rozwiązanie zależności. Określenie które mikroserwisy mogą zostać zbudowane w pełni bez dodatkowych operacji i częściowego generowania.

Bazowe mikroserwisy wymagają jedynie generowania ich w odpowiedniej kolejności. Przykładem tego typu mikroserwisów mogą być:

1. „Książki” nieposiadający żadnych zależności
2. „Pracownicy” nieposiadający żadnych zależności
3. „Biblioteka” wymagający zależności „Książki” oraz „Pracownicy”

Aby utworzyć końcowy mikroserwis Biblioteka wymagane wcześniej jest utworzenie aplikacji „Książki” oraz „Pracownicy”.

W tym celu utworzony prosty i mało optymalny algorytm pokazany na listingu 19. Algorytm sprawdza wielokrotnie czy wymagania projektu zostały spełnione na podstawie przeanalizowanych projektów wcześniej projektów. Algorytm powinien być ograniczony pod względem ilości operacji. Optymalizacja w przypadku zastosowania w generatorze nie jest potrzebna, ponieważ ilość mikroserwisów generowanych nie będzie duża.

Listing 19. Pseudokod ustalania kolejności tworzenia projektów bez zależności kołowych

```
wczytaj mikroserwisy bez zależności jako bezZal
wczytaj mikroserwisy z zależnościami jako zZal

bezZaleznosciKolowych := bezDep
Posortuj listę zZal według ilości zależności

dla i := 0, 1, ... , len(zZal) - 1:
    dla każdego elementu listy zZal:
        jeżeli aktualny element znajduje się w bezZaleznosciKolowych
            usuń z listy zZal aktualny element
            kontynuuj

        jeżeli aktualny element nie może być utworzony
               na podstawie elementów listy bezZaleznosciKolowych
            kontynuuj

        dodaj do bezZaleznosciKolowych aktualny element

zwróć bezZaleznosciKolowych
```

Wynikiem całej logiki rozwiązywania zależności jest podzielona lista usług na te które mogą zostać zbudowane w całości na podstawie zależności wybudowanych wcześniej oraz na te które muszą być generowane jednocześnie.

5.4.2 Przygotowanie projektu bez zależności kołowych

Bazowe projekty mogą być wygenerowane w całości bez potrzeby oczekiwania na wygenerowania części innego projektu. Jedynym wymaganiem jest spełnienie zależności używając wygenerowane wcześniej projekty.

Początkowym krokiem jest utworzenie projektu podstawowego, który w przypadku platformy .NET jest tworzony przy pomocy narzędzia .NET CLI. Na bazie tego dodawane są dodatkowe katalogi oraz konfiguracje dotyczące Swaggera.

Każdy z modeli dostarczonych wewnątrz konfiguracji zostaje przetworzony na klasę z tymi samymi właściwościami. Modele zawierające relację z innymi modelami dodatkowo wymagają utworzenia klas pośrednich do przechowywania kluczy pierwszych relacji. Przykładowo encja `ShopCart` będąca reprezentacją koszyka, który może zawierać w sobie wiele produktów. Dodatkowym wygenerowanym modelem przedstawionym na listingu 20 jest `CartProduct` który przechowuje klucz podstawowy modelu `ShopCart` oraz modelu `Product`.

Listing 20. Model pośredni utworzony na podstawie skonfigurowanej relacji

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;

namespace ShopCartService.Models;

public partial record CartProduct
{
    [Key]
    public Guid Id { get; set; }

    public Guid ProductId { get; set; }
}
```

Jeśli projekt ma zależności to musi mieć też możliwość na komunikowanie się z nimi. Każda zależność jest zamieniana w odpowiednik interfejsu wraz z odpowiednimi dla adnotacjami. Przykład wygenerowanego interfejsu przedstawiono na listingu 21, Adnotacja `Get`, `Post`, `Put` oznaczają metodę, która powinna zostać wykorzystana do dokonania komunikacji. Biblioteka Refit wspomaga w tej sytuacji poprzez automatyczne generowanie obsługi komunikacji przy użyciu protokołu HTTP.

Listing 21. Wygenerowany interfejs do obsługi komunikacji z mikroserwisem Produkt

```
using Refit;
using ShopCartService.Dtos;
using ShopCartService.Models;

namespace ShopCartService.Infrastructure;
public partial interface IProductService
{
    [Get("/product")]
    Task<List<Product>> getAllProduct();

    [Get("/product/{Id}")]
    Task<Product> getProduct(Guid Id);

    [Post("/product")]
    Task<Product> createProduct(Product body);

    [Put("/product")]
    Task<Product> updateProduct(Product body);

    [Delete("/product/{Id}")]
    Task<Product> removeProduct(Guid Id);
}
```

Każdy model mikroserwisu zależnego zamieniany jest w klasy DTO (ang. *data transfer object*). Każda zależność musi dodatkowo zostać zarejestrowana jako nowa usługa w pliku `Program.cs`. Przykładowa rejestracja mikroserwisu zależnego `BookService` została przedstawiona na listingu 22.

Listing 22. Wpis w pliku `Program.cs` na temat komunikacji poprzez Refit z zewnętrzną usługą

```
builder.Services
       .AddRefitClient<IBooksService>()
       .ConfigureHttpClient(c => c.BaseAddress =
                                 new Uri("http://BooksService"));
```

Następnym krokiem jest utworzenie warstwy bazy danych. Projekty .NET w wersji podstawowej wymagają zainstalowania dodatkowo:

- Narzędzia *dotnet-ef*
- Pakietu NuGet `Microsoft.EntityFrameworkCore.Sqlite`
- Pakietu NuGet `Microsoft.EntityFrameworkCore.Design`

Wewnątrz katalogu `Infrastructure` tworzony jest kontekst na podstawie pliku wzorca pokazanego na listingu 23. Wzorzec jest plikiem, który wykorzystuje biblioteka Jinja. Znaczniki

oznaczone klamrami wewnątrz wzorca są podmienione na podstawie danych przekazanych bibliotece. Przykładem takiego znacznika jest `{{ className }}`, przekazanie słownika z kluczami powiązanymi ze znacznikami powoduje wypełnienie wzorca odpowiednimi danymi.

Listing 23. Wzorcowy plik `context.template` obsługiwany przez bibliotekę Jinja

```
{% extends 'base_src_file.template' %}
{% block usings %}{% for using in usings %}{{using}}
{% endfor %}{% endblock %}
{% block body %}
public partial class {{ className }} : DbContext {
{% for property in properties %}
      {{property}}
{% endfor %}
      public string DbPath { get; }

      public {{ className }}()
      {
            DbPath = System.IO.Path.Join(".", "{{ className }}.db");
      }

      protected override void OnConfiguring(DbContextOptionsBuilder
                                            options)
      {
            options.UseSqlite($"Data Source={DbPath}");
      }
}
{% endblock %}
```

Znacznik `{% for property in properties %}` pozwala na iterowanie po liście elementów przekazanych do wzorca. Umożliwia to zapisanie wewnątrz kontekstu każdego modelu mikroserwisu. Przykładowy wynik końcowy transformacji wzorca w plik został przedstawiony na listingu 12.

Ostatnim z brakujących elementów całego mikroserwisu jest obsługa komunikacji przychodzącej. Całość budowana jest na podstawie trzech wzorców: `controller.template`, `endpoint.template` oraz jednego z wzorca logiki. Każdy *endpoint* zdefiniowany w konfiguracji jest odwzorowywany na jedną metodę wewnątrz kontrolera.

5.4.3 Przygotowanie projektów z zależnościami kołowymi

Zaimplementowany prototyp posiada możliwość tworzenia projektów z zależnościami kołowymi. Jest to sytuacja, której należy unikać w przypadku ustrukturowania kodu, w przypadku architektury mikroserwisów jest to normalna sytuacja. Generowanie takich aplikacji jest utrudnione, informacje potrzebne do utworzenia aplikacji A są niedostępne do momentu utworzenia aplikacji B. Pełne utworzenie aplikacji B także nie jest możliwe do zrealizowania, bo np. wymaga informacji na temat komunikacji oferowanej przez aplikację A.

Aplikacje z zależnościami kołowymi nazwane później kompleksowymi tworzone są po utworzeniu aplikacji bazowych. Procesy tworzenia komponentów aplikacji są niezmienione,

zmodyfikowana jest kolejność wykonywania operacji. Kolejność wykonywanych operacji została przedstawiona na listingu 24.

Listing 24. Pseudokod tworzenia projektów z zależnościami kołowymi

```
wygenerowaneProjekty = []

dla każdego mikroserwisu kompleksowego w konfiguracji:
    utwórz projekt kompleksowy z dostępnych zależności bazowych
    utwórz modele zawierające powiązania z innymi mikroserwisami
    zapisz projekt w wygenerowaneProjekty

dla każdego elementu wygenerowaneProjekty:
    uzupełnij aktualny projekt zależnościami wygenerowanymi
    utwórz kontekst warstwy danych
    utwórz endpointy oraz kontroler

dla każdego elementu wygenerowaneProjekty:
    utwórz usługi komunikacji z innymi mikroserwisami
```

Zaprezentowane rozwiązanie na listingu 24 posiada ograniczenie, modele mikroserwisów nie mogą mieć zależności kołowych. Problem ten nie został rozwiązany z powodu braku potencjalnego zastosowania tego typu sytuacji w realnym wykorzystaniu mikroserwisów. Opisana sytuacja oznacza błędne ustrukturyzowanie architektury wszystkich aplikacji i powinna zostać rozwiązana w konfiguracji.

5.4.4 Utworzenie skryptów tworzących kontener

Utworzone projekty znajdują się w katalogach oraz są gotowe do uruchomienia. Aplikacje są niezależne oraz nie wiedzą o swoim istnieniu. W celu utworzenia jednego systemu zbudowanego ze wszystkich wygenerowanych aplikacji wymaga warstwy zarządzania.

Pierwszym krokiem umożliwienia zarządzania jest ujednolicenie aplikacji. Najpopularniejszym aktualnie standardem opakowanych aplikacji są kontenery. Zastosowano w tym przypadku technologię Docker. Skrypty tworzenia obrazu tej technologii są generowane na podstawie informacji zawartych w wygenerowanym projekcie. Tak jak w przypadku poprzednich kroków, tworzenie skryptu jest operacją specyficzną dla języka, w którym projekt został wygenerowany.

Na rzecz projektów generowanych w platformie .NET przygotowano prosty wzorzec przedstawiony na listingu 25. Dostępne w pliku znaczniki wzorca to:

- `projectFileName` – podmieniany na nazwę mikroserwisu,
- `projectFileExtension` – podmieniany na wartość „csproj”,

- `entryFileName` – podmieniany na nazwę mikroserwisu,
- `entryFileExtension` – podmieniany na wartość „dll”,

Przykładowy wygenerowany skrypt tworzenia obrazu Docker został przedstawiony na listingu 14.

Listing 25. Wzorcowy plik `project_packaging.template` obsługujący generowanie `Dockerfile` dla projektów .NET

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["{{ projectFileName }}.{{projectFileExtension}}", "."]
RUN dotnet restore "./{{ projectFileName }}.{{projectFileExtension}}"
COPY . .
WORKDIR "/src/."
RUN dotnet build \
"{{ projectFileName }}.{{ projectFileExtension }}" \
-c Release \
-o /app/build

FROM build AS publish
RUN dotnet publish \
"{{ projectFileName }}.{{ projectFileExtension }}" \
-c Release \
-o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "{{ entryFileName }}.{{ entryFileExtension }}"]
```

5.4.5 Przydzielenie portów i utworzenie konfiguracji Docker compose

Warstwą zarządzania kontenerami zajmuje się Docker compose. Utworzona konfiguracja pozwala na określenie, które kontenery powinny być uruchamiane, ich konfigurację oraz kolejność uruchamiania. Każdy z kontenerów ma dwa udostępniane porty, aby umożliwić komunikację porty muszą być przekierowane na zewnątrz. Porty przekierowane zaczynają się od wartości 8000. Każdy kolejny mikroserwis dostaje następne wartości portów.

5.5. Wynik końcowy

Przykładowy plik konfiguracyjny zawiera informacje potrzebne do utworzenia trzech mikroserwisów wymaganych do spełnienia założonego wymagania. W przypadku pomyślnego

generowania, prototyp wyświetla informację o dostępnych mikroserwisach oraz ich portach. Przykładowy pozytywny wynik został przedstawiony na rysunku 10.

```
INFO:root:ProductService: Preparing dockerfile
INFO:root:ShopCartService: Preparing dockerfile
INFO:root:OrderService: Preparing dockerfile
INFO:root:Service 'ProductService' will be availiable under port: 8001.
INFO:root:Service 'ProductService' swagger http://localhost:8001/swagger.
INFO:root:Service 'ShopCartService' will be availiable under port: 8002.
INFO:root:Service 'ShopCartService' swagger http://localhost:8002/swagger.
INFO:root:Service 'OrderService' will be availiable under port: 8003.
INFO:root:Service 'OrderService' swagger http://localhost:8003/swagger.
INFO:root:Created docker compose file
INFO:root:Disabled automatic service building. Run 'docker-compose up --build' in output directory.'
INFO:root:Finished successfully.
```

Rysunek 10. Pozytywne zakończenie procesu generowania mikroserwisów

Aplikacje są dostępne jako kontenery połączone ze sobą wewnętrzną siecią. Na przedstawionym rysunku 10 użytkownik jest informowany o wyłączonym trybie automatycznego budowania i uruchamiania kontenerów. Uruchomienie automatyczne blokuje zakończenie pracy generatora do wyłączenia mikroserwisów.

5.5.1 Uruchomienie projektów

Wyłączony tryb automatycznego uruchamiania mikroserwisów wymaga od użytkownika uruchomienia manualnie komendy narzędzia Docker compose. Plik *docker-compose.yml* znajduje się w katalogu wynikowym, domyślna nazwa katalogu to `output`. W celu wykorzystania przygotowanej konfiguracji należy uruchomić polecenie `docker-compose up --build`. Opcja komendy `--build` jest opcjonalna, informuje Docker compose o zbudowaniu obrazów od początku. Obrazy po pierwszym uruchomieniu są zapisywane w wewnętrznej kolekcji obrazów. Przyspiesza to tworzenie kontenerów.

5.5.2 Pierwsze połączenie

Wynik uruchomienia kontenerów został przedstawiony na rysunku 11. Informacje wypisywane przez poszczególne kontenery są oznaczane etykietą o innym kolorze. W przypadku rysunku 11 zaprezentowano logi wypisane na strumień *stdout* mikroserwisu `ShopCartService` oraz `ProductService`. Wyświetlane są również informacje o występujących błędach.

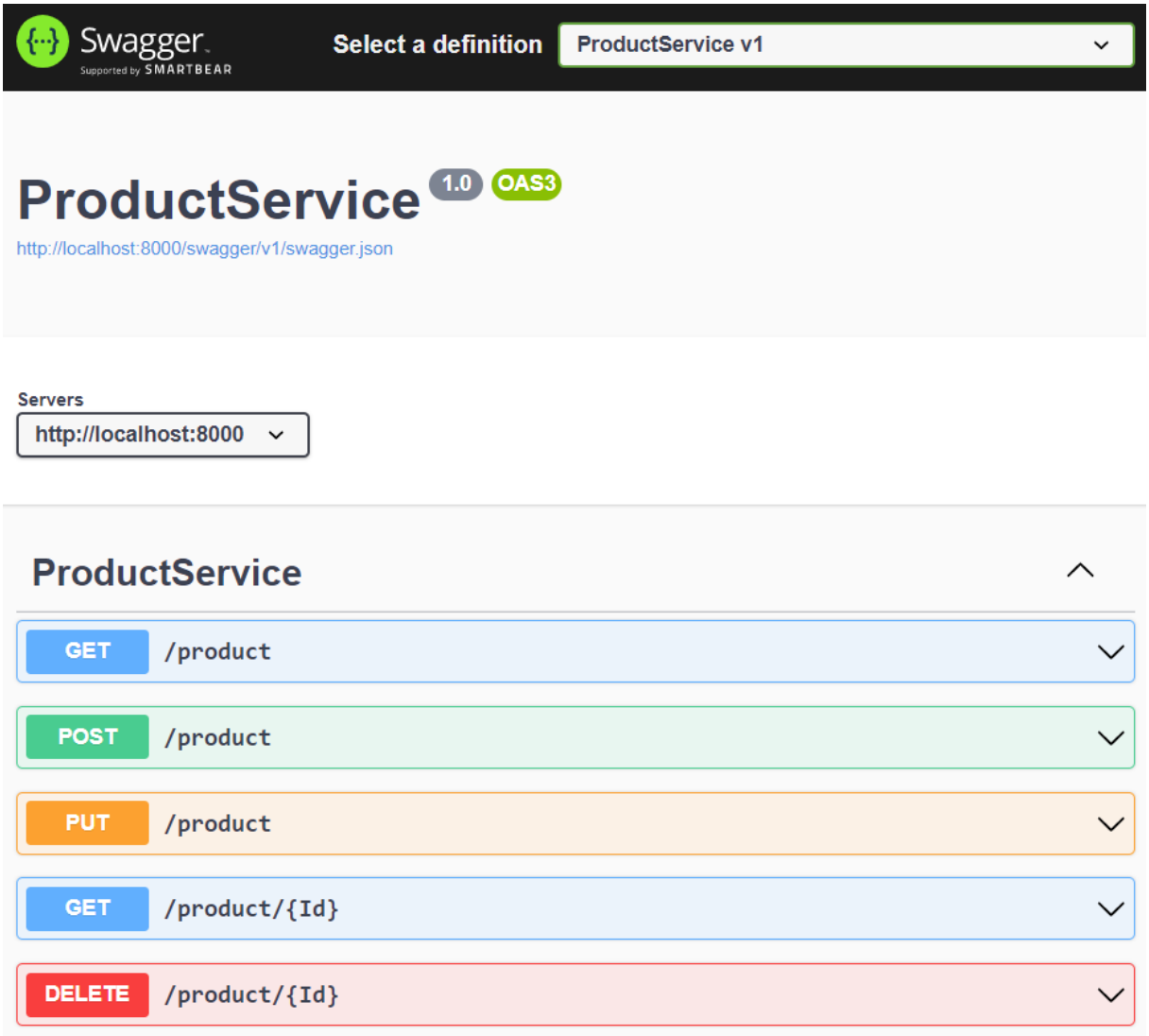
Aby zweryfikować działanie utworzonych aplikacji wystarczy otworzyć przeglądarkę i przejść pod URL: `http://localhost:8000`. W przypadku pojawienia się napisu *Hello World!* mikroserwis jest gotowy do działania.

```
[+] Running 3/0
 - Container output-productservice-1   Created                                    0.0s
 - Container output-shopcartservice-1  Created                                    0.0s
 - Container output-orderservice-1     Created                                    0.0s
Attaching to output-orderservice-1, output-productservice-1, output-shopcartservice-1
output-shopcartservice-1  | {"EventId":14,"LogLevel":"Information","Category":"Microsoft.Hostin
g.Lifetime","Message":"Now listening on: http://[::]:80","State":{"Message":"Now listening on:
http://[::]:80","address":"http://[::]:80","{OriginalFormat}":"Now listening on: {address}"}}
output-shopcartservice-1  | {"EventId":0,"LogLevel":"Information","Category":"Microsoft.Hosting
.Lifetime","Message":"Application started. Press Ctrl\u002BC to shut down.","State":{"Message":
"Application started. Press Ctrl\u002BC to shut down.","{OriginalFormat}":"Application started.
 Press Ctrl\u002BC to shut down."}}
output-shopcartservice-1  | {"EventId":0,"LogLevel":"Information","Category":"Microsoft.Hosting
.Lifetime","Message":"Hosting environment: Production","State":{"Message":"Hosting environment:
 Production","envName":"Production","{OriginalFormat}":"Hosting environment: {envName}"}}
output-shopcartservice-1  | {"EventId":0,"LogLevel":"Information","Category":"Microsoft.Hosting
.Lifetime","Message":"Content root path: /app/","State":{"Message":"Content root path: /app/","
contentRoot":"/app/","{OriginalFormat}":"Content root path: {contentRoot}"}}
output-productservice-1   | {"EventId":14,"LogLevel":"Information","Category":"Microsoft.Hostin
g.Lifetime","Message":"Now listening on: http://[::]:80","State":{"Message":"Now listening on:
http://[::]:80","address":"http://[::]:80","{OriginalFormat}":"Now listening on: {address}"}}
```

Rysunek 11. Uruchomione kontenery mikroserwisów

5.5.3 Dostęp do *Swaggera*

Generowane aplikacje nie posiadają żadnego interfejsu użytkownika. Tworzone jest API które może zostać wykorzystane w dodatkowej aplikacji interfejsu użytkownika. W celu ułatwienia testowania i budowy tego typu aplikacji przydatny może być Swagger. Dokumentacja API oraz aplikacja testowa do obsługi udostępnianego API jest dostępna pod adresem URL: `http://localhost:8000/swagger`. Swagger UI zaprezentowany na rysunku 12 zawiera wszystkie metody obsługi zapytań, przykładowe zapytania oraz odpowiedzi. Definicja specyfikacji OpenAPI 3.0 znajduje się pod adresem: `http://localhost:8000/swagger/v1/swagger.json`.



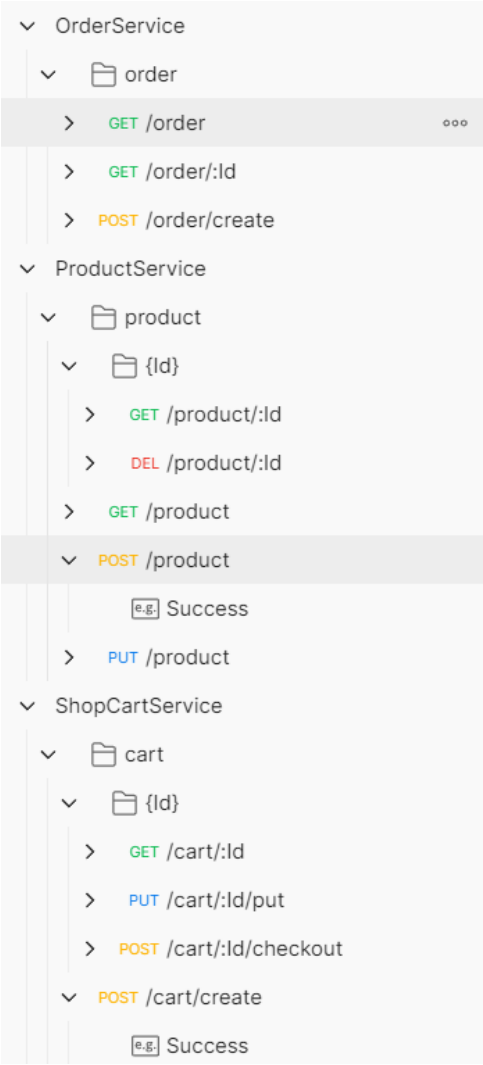
Rysunek 12. Interfejs użytkownika narzędzia *Swagger UI* dla *ProductService*

5.5.4 Wykorzystanie narzędzia Postman do weryfikacji działania

Specyfikacje *OpenAPI* każdego mikroserwisu udostępniane mogą zostać zaimportowane bezpośrednio do narzędzia *Postman*. Przykładowa struktura kolekcji i zapytań została zaprezentowana na rysunku 13. *Postman* może zostać wykorzystany do sprawdzenia poprawności działania mikroserwisów jak i komunikacji między nimi. Kroki przykładowego procesu zarządzania transakcją może być:

1. Utworzenie produktu A oraz produktu B przy użyciu metody z kolekcji `ProductService` *POST* `/product`
2. Utworzenie koszyka metodą `ShopCartService` *POST* `/cart/create`
3. Dodanie identyfikatorów produktów do koszyka przy użyciu metody `ShopCartService` *PUT* `/cart/:Id/put`
4. Utworzenie zamówienia na podstawie wypełnionego koszyka przy użyciu metody `ShopCartService` *POST* `/cart/:Id/checkout`

5. Wysłanie zapytania do metody mikroserwisu `OrderService` *GET* `/order` zwraca utworzone zamówienia



Rysunek 13. Kolekcje narzędzia *Postman* zaimportowane z utworzonych mikroserwisów – źródło: opracowanie własne

Testowanie oraz wykorzystywanie mikroserwisów wygenerowanych dla innej konfiguracji będzie analogiczny. Importowane zapytania posiadają przykładowe opisy zapytań pozytywnych. Przykładowe ciało zapytania HTTP może zawierać elementy opcjonalne. W przypadku zapytań dodawania obiektów przekazywanie identyfikatora nie jest wymagane, jest on tworzony automatycznie przez aplikacje.

6. Porównanie kodu i generowanych aplikacji

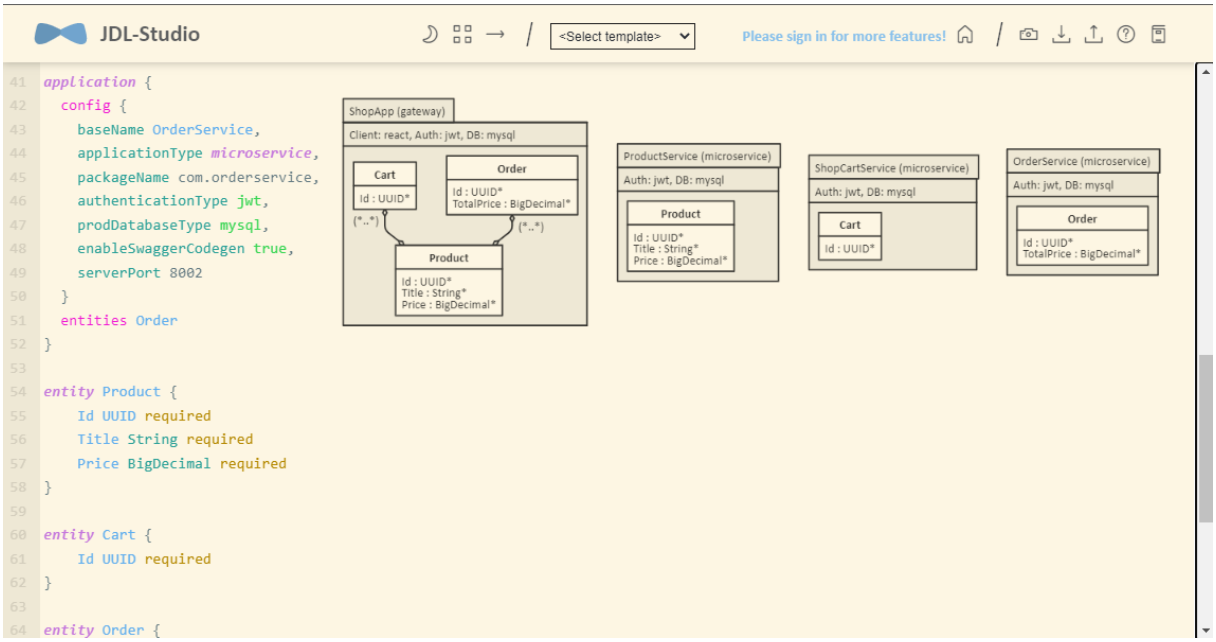
Dostępne narzędzia do generowania projektów pozwalają na zautomatyzowanie większości wymaganych procesów początkowych. Tworzone są pliki źródłowe, struktura katalogów, modele oraz warstwa modelu danych. Ważnym elementem w przypadku mikroserwisów jest umożliwienie komunikacji oraz zapewnienie stabilności działania całej infrastruktury. Wybrano najpopularniejszy i największy konkurencyjny generator w celu porównania jego wyniku końcowego wraz z rezultatem działania przygotowanego prototypu.

6.1. Generowanie mikroserwisów w JHipster

Tworzenie aplikacji monolitycznych w narzędziu JHipster jest prostym procesem. Wymagane jest zdefiniowanie ustawień projektu, encji oraz relacji między nimi. Poprawnie przygotowana konfiguracja może zostać zamieniona w gotowy projekt. Generator posiada liczne ograniczenia dotyczące tworzenia mikroserwisów. W tym przypadku konfiguracja nie zawsze zostanie poprawnie zinterpretowana a także utworzony projekt będzie niepoprawny. Szczegóły procesu tworzenia projektów mikroserwisowych przedstawiono w tym podrozdziale.

6.1.1 Konfiguracja

W celu przygotowania projektu mikroserwisowego w oprogramowaniu JHipster wymagane jest utworzenie pliku konfiguracji JDL. Plik taki można wygenerować przy pomocy dostępnych rozszerzeń do popularnych środowisk programistycznych. Przykładem takiego narzędzia jest edytor JDL-Studio, który dostępny jest na stronie `https://start.jhipster.tech/jdl-studio`. Narzędzie to pozwala na łatwe przygotowanie pliku JDL oraz jego wizualizacje. Przykład wizualizacji tworzonej konfiguracji został zaprezentowany na rysunku 14.



Rysunek 14. Interfejs użytkownika narzędzia JDL-Studio – źródło: opracowanie własne

`Application` to struktura, która jest odpowiedzialna za definiowanie aplikacji mikroserwisowej. Struktura ta wymaga konfiguracji dotyczącej typu, nazwy oraz pakietu aplikacji. Dodatkowe parametry pozwalają na konfigurację sposobu autoryzacji, rodzaju bazy danych a także portu wykorzystywanego w przypadku mikroserwisów. Następnym głównym elementem struktury `application` jest zdefiniowanie wykorzystywanych encji. Dostępne encje muszą zostać wymienione rozdzielając je przecinkiem za znacznikiem `entities`.

`Entity` jest strukturą do zdefiniowania informacji o modelu. Poszczególne linie konfiguracji pomiędzy klamrami określają jakie właściwości zawiera model. Każda właściwość zbudowana jest z nazwy, typu oraz dodatkowych atrybutów walidacyjnych.

JHipster wspiera także dziedziczenie modeli, komentarze oraz pozwala na zdefiniowanie następujących typów cech:

- `String`,
- `Integer`,
- `Long`,
- `BigDecimal`,
- `Float`,
- `Double`,
- `Enum`,
- `Boolean`,
- `LocalDate, ZonedDateTime`,
- `Instant`,
- `Duration`,
- `UUID`,
- `Blob`,
- `AnyBlob, ImageBlob, TextBlob`

W celu zdefiniowania zależności między modelami, należy wykorzystać strukturę `relationship`. Wspierane są wszystkie dostępne rodzaje relacji `OneToOne` (ang. *jeden-do-jednego*), `OneToMany` (ang. *jeden-do-wielu*), `ManyToOne` (ang. *wiele-do-jednego*), `ManyToMany` (ang. *wiele-do-wielu*).

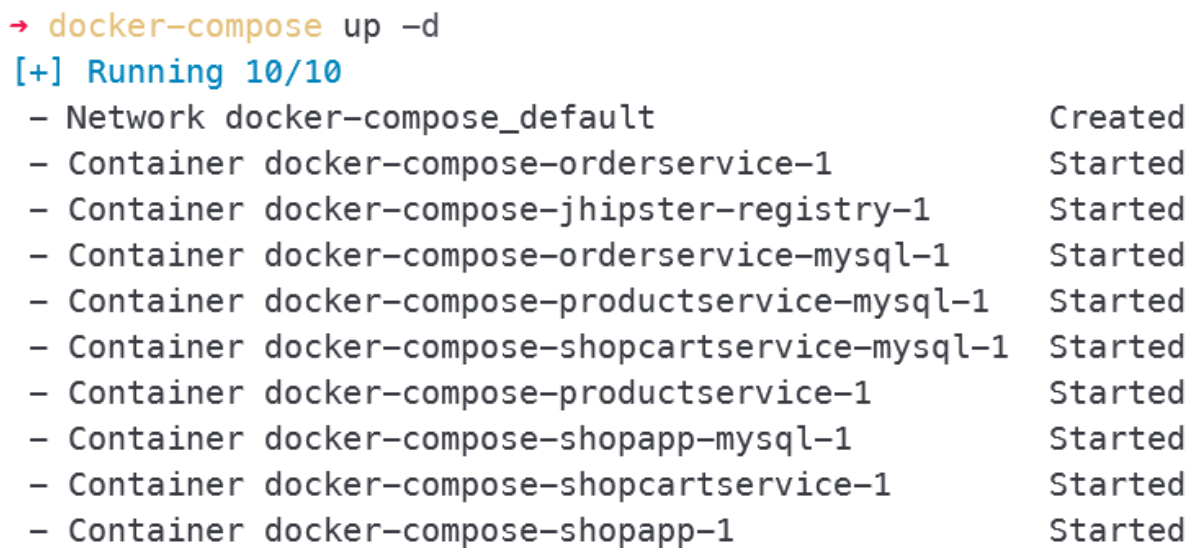
Ostatnim rodzajem wykorzystanej struktury jest `deployment`. Umożliwia ona skonfigurowanie informacji o wdrożeniu, strukturze wszystkich aplikacji oraz konfiguracji sposobu konteneryzacji.

6.1.2 Generowanie oraz uruchomienie infrastruktury

Utworzona konfiguracja zawiera informację jakie generator JHipster powinien przetworzyć. W celu rozpoczęcia procesu generowania należy wykonać polecenie `jhipster jdl .\plik_jdl.jdl`. Początek procesu generowania został przedstawiony na rysunku 15, gdzie wynikiem działania są wygenerowane katalogi z projektami.

Rysunek 15. Początek procesu generowania aplikacji przy użyciu JHipster – źródło: opracowanie własne

Projekty w aktualnym stanie mogą zostać uruchomione osobno przy pomocy polecenia `mvnw`. Poza projektami wymagane jest uruchomienie JHipster Registry, zarządcy wszystkimi mikroserwisami. Duża ilość kroków, które są wymagane do wykonania, skłania do wykorzystania osobnego rozwiązania jakim jest konteneryzacja. Generator pozwala na tworzenie konfiguracji Docker compose. Wymagane jest utworzenie obrazów każdej aplikacji poprzez uruchomienie polecenia: `./gradlew bootJar -Pprod jibDockerBuild`. W celu uruchomienia wszystkich kontenerów wraz z rejestrem oraz dodatkowymi zależnościami, należy przejść do katalogu `docker-compose` i uruchomić komendę `docker-compose up -d`. Pozytywny wynik utworzenia wszystkich mikroserwisów został przedstawiony na rysunku 16.



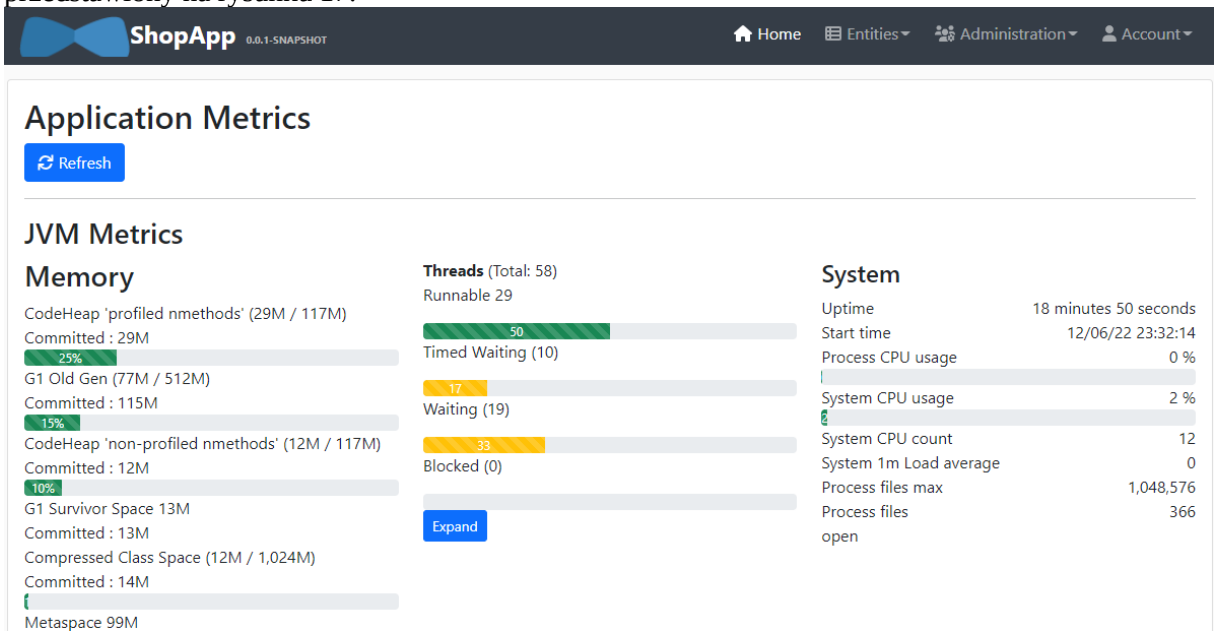
Rysunek 16. Uruchomienie kontenerów z mikroserwisami wygenerowanymi przy użyciu narzędzia JHipster - źródło: opracowanie własne

6.1.3 Panel zarządzania infrastrukturą oraz dostęp do mikroserwisów

Główny interfejs użytkownika dostępny jest pod adresem URL: `http://localhost:8080/`. W celu dostępu do zasobów wymagane jest zalogowanie, domyślny użytkownik posiada login: *admin*. Użytkownik ten umożliwia wyświetlanie wszystkich informacji na temat całej infrastruktury. Przykładowymi możliwościami panelu administracyjnego są:

- zarządzanie encjami oraz ich aktualnymi danymi,
- kontrola bramy wejściowej komunikacji z mikroserwisami,
- zarządzanie użytkownikami oraz ich profilami,
- statystyki i aktualny status aplikacji,
- konfiguracja platformy programistycznej Spring i bibliotek zależnych,
- konfiguracja logów

Widok statystyk panelu administracyjnego wygenerowanego projektu na rzecz tej pracy został przedstawiony na rysunku 17.



Rysunek 17. Panel zarządzania wygenerowany przez narzędzie JHipster - źródło: opracowanie własne

Dostęp do utworzonych mikroserwisów jest ograniczony dwoma elementami: uwierzytelnianiem JWT oraz bramą. W celu otrzymania identyfikatora JWT należy wysłać login i hasło pod adres URL: `http://localhost:8080/api/authenticate`. Otrzymany identyfikator należy dodać do nagłówka HTTP `Bearer` zawartego wewnątrz każdego zapytania wykonywanego do mikroserwisów. Przykładowe zapytanie w przypadku wygenerowanych aplikacji na rzecz tej pracy może zostać wysłane pod adres `http://localhost:8080/services/productservice/api/products`. Wynikiem zapytania jest wyświetlenie wszystkich dostępnych produktów w aplikacji *productservice*.

Wszystkie dostępne mikroserwisy są także dostępne w panel administracyjnym. Oferowany Swagger pozwala na zarządzanie mikroserwisem przy użyciu wygodnego interfejsu zapytań.

6.2. Analiza porównawcza z prototypem

Wymagania przedstawione na rysunku 9 zostały przygotowane, aby zaprezentować działanie generatorów oraz infrastruktury mikroserwisowej. Aplikacje wymagane do utworzenia mają oferować prostą funkcjonalność operacji wykonywanych w sklepie internetowym. Usługi oferują tworzenie produktów, dodawanie ich do koszyka oraz wykonywanie na podstawie koszyka zamówienia. W przypadku mikroserwisów aplikacje muszą być niezależne, uruchamiane osobno oraz powinny umożliwiać komunikację między sobą.

Projekty utworzone przy pomocy przygotowanego prototypu oraz narzędzia JHipster są bardzo odmienne. Posiadają odmienne podejście architektoniczne oraz zawierają różniące się od siebie zależności. Aplikacje utworzone przez prototyp oparte są o platformę .NET, gdzie konkurencyjne tworzone są na platformie Spring.

6.2.1 Wady i ograniczenia

Oba generatory posiadają znaczące różnice oraz posiadają swoje własne ograniczenia. Wymienione poniżej limitacje dotyczą założonych wymagań i są nie możliwe do rozwiązania bez ingerencji w implementacje tworzonych aplikacji.

Największym ograniczeniem obu generatorów jest ilość dostępnych sposobów tworzenia metod obsługujących zapytania. JHipster tworzy tylko podstawowe metody manipulacji na modelach. Oferowane są metody: odczytania jednego obiektu, wyczytaniu wszystkich obiektów z danej kolekcji, utworzenie nowego obiektu, modyfikacja obiektu, usunięcie obiektu. W przypadku prototypu dostępne są 3 dodatkowe rodzaje zapytań: wywołanie innej metody, wysłanie obiektu z aktualnego mikroserwisu do innego jako obiekt zależny oraz metoda odbierania zależnego obiektu. W obu przypadkach ograniczenie to wymaga dodatkowej modyfikacji projektów tworzonych przez generator.

Utworzona konfiguracja JHipster posiada czytelniejszą składnię konfiguracji oraz zawiera możliwość wykorzystania narzędzia manualnie. JDL język przygotowany na rzecz wspomnianego narzędzia posiada wiele generatorów, edytorów wizualnych oraz oferuje wiele dodatkowej konfiguracji. Prototyp posiada konfigurację, która musi zostać przygotowana ręcznie i zapisana w formacie ogólnego przeznaczenia JSON.

W przypadku narzędzia konkurencyjnego zastosowano odmienne podejście do tworzenia metod obsługujących zapytanie. Programista nie posiada kontroli nad nazwą metody, ścieżką URL oraz ilością *endpointów* dostępnych wewnątrz tworzonej aplikacji. Metody generowane są automatycznie na podstawie zdefiniowanych encji. Ułatwia to automatyzację tworzenia panelu administracyjnego dla wygenerowanych encji oraz mikroserwisów. Prototyp posiada oddzielone zarządzanie modelami oraz metodami obsługi zapytań. Konfiguracja umożliwia większą elastyczność i pozwala na zdefiniowanie wyżej wymienionych informacji.

Jednym z ważniejszych ograniczeń JHipster w kontekście założonych wymagań jest brak możliwości wymiany informacji między mikroserwisami. Ograniczenie to zmusza do utworzenia encji w każdej z dostępnych tworzonych aplikacji i symulowanie zakładanego działania. Każda z dostępnych opcji konfigurowania relacji między encjami powoduje utworzenie dwustronnego powiązania wewnątrz klas modeli.

Ważnym elementem infrastruktury mikroserwisowej jest jej bezpieczeństwo. Prototyp w tym przypadku nie posiada żadnego rodzaju zabezpieczenia, wszystkie dane są udostępniane bez żadnego uwierzytelniania oraz autoryzacji. JHipster w tym przypadku przoduje pozwala na wybranie sposobu autoryzacji do danych aplikacji. Posiada podstawowe uwierzytelnianie JWT, przechowywanie zalogowanego użytkownika w sesji oraz rozbudowane OAuth2.

Prototyp wspiera aktualnie tylko jedną bazę danych SQLite, konkurencyjne narzędzie pozwala na utworzenie bazy SQL, NoSQL oraz innych. Poza systemem bazodanowym oferowana jest możliwość

przechowywania danych tymczasowych oraz doinstalowania systemów wyszukiwania np. Elasticsearch.

W przypadku obu tworzonych infrastruktur oferowany jest Swagger do komunikacji z mikroserwisami. JHipster dodatkowo tworzy bramę obsługi zapytań która zawiera także panel zarządzania. Oferowany interfejs użytkownika domyślnie nie posiada połączenia z mikroserwisami zależnymi, zmiany wykonane w panelu administracyjnym nie oznaczają wykonanie operacji na encji wewnątrz mikroserwisu.

6.2.2 Generowanie aplikacji

Konkurencyjne narzędzie wobec przygotowanego na rzecz tej pracy prototypu posiada bardzo duże możliwości konfiguracji. Dostępna dokumentacja pozwala na opanowanie podstaw działania oraz wymaganych informacji od użytkownika. Generator oferuje dwa sposoby działania, manualny oraz automatyczny na podstawie pliku konfiguracyjnego JDL. W przypadku wykorzystania sposobu pierwszego proces jest powolny oraz wymaga dużej uwagi ze strony użytkownika. Sposób drugi pozwala na automatyzację całego procesu, lecz wiąże się to potencjalnymi błędami które nie są bezpośrednio zgłaszane użytkownikowi. Błąd konfiguracji JDL w większości przypadków nie oznacza zatrzymania procesu generowania. Aplikacje mogą zostać wygenerowane częściowo o czym użytkownik dowie się dopiero przy pierwszym uruchomieniu.

Tworzenie aplikacji w JHipster wykonywane jest równolegle co powoduje przyspieszenie całego procesu. Efektem ubocznym jest także duży chaos w informacjach wyświetlanych użytkownikowi końcowemu. Dużym wyzwaniem jest znalezienie błędu, który powoduje niepoprawne tworzenie całej infrastruktury.

W celu zbadania szybkości generowania projektu wykorzystano polecenie dostępne w systemie Windows `Measure-Command`. Utworzenie wszystkich aplikacji do spełnienia założonych wymagań wymaga od narzędzia JHipster pracy przez ponad 3 minuty. Kolejne uruchomienie generatora dzięki zapisywanym metadanym wykonywane jest szybciej. Przygotowany prototyp na rzecz tej pracy wymaga ponad minuty, aby utworzyć podobnej struktury infrastrukturę. Test został wykonany na sprzęcie wyposażonym w procesor Intel i7-10750H o częstotliwości 2,6 GHz. Komputer testowy zawiera 32GB pamięci fizycznej.

Wspomniany wyżej czas wykonywania tworzenia aplikacji w narzędziu JHipster bierze pod uwagę tylko utworzenie wszystkich projektów oraz utworzenie pliku narzędzia `docker-compose.yml`. Dodatkowym procesem wymaganym, aby uruchomić cały system jest wygenerowanie obrazu kontenera. Użytkownik w zależności od ilości oczekiwanych mikroserwisów musi wejść w katalog projektu i uruchomić polecenie tworzące obraz. Prototyp pozwala na automatyczne tworzenie obrazów aplikacji przy użyciu narzędzia Docker compose.

6.2.3 Porównanie rozmiarów projektów

W przypadku porównywanych projektów porównywano same implementacje aplikacji zawierających obsługę zapytań. W przypadku projektów tworzonych w języku C# sprawdzano długość plików o rozszerzeniu *.cs*. Tę samą metodologię wykorzystano wobec wyniku działania JHipster. Pliki źródłowe tych projektów mają rozszerzenie *.java*. Generator JHipster tworzy także pliki źródłowe zawierające testy, zostaną one pominięte w porównaniu. Do wykonania porównania wykorzystano połączenie poleceń `find`, `xargs` oraz `wc`. Przykładowym poleceniem do liczenia ilości linii plików źródłowych języka C# jest: `find . -name '*.cs' | xargs wc -l`.

Tabela 1. Porównanie wielkości generowanych plików źródłowych wszystkich aplikacji - źródło: opracowanie własne

	JHipster	Prototyp
Ilość utworzonych plików źródłowych	237	28
Sumaryczna ilość linii	14429	1259
Ilość linii największego pliku	347 – plik: `UserService.java`	79 – plik: `ShopCartServiceController.cs`

W tabeli 1 przedstawiono zestawienie wielkości generowanych projektów przez testowane generatory. JHipster tworzy projekty ponad 10 razy większe od utworzonego prototypu na rzecz tej pracy. Jest to związane głównie z dostępnymi dodatkowymi funkcjami. Na ten wniosek wskazuje wielkość pliku UserService.java. Jest to plik odpowiedzialny za klasę zarządzającą użytkownikami, resetowaniem hasła itp. Prototyp nie oferuje uwierzytelniania oraz wielu innych dostępnych w JHipster możliwości, powodem tego jest znacznie mniejszy rozmiar projektu. Pomaga to programiście w zrozumieniu utworzonego projektu.

Porównanie plików źródłowych dotyczy głównie osób w przyszłości zarządzających implementacją. Kolejną kwestią jest wielkość projektu pod względem zajmowanego miejsca na dysku oraz wielkość tworzonego obrazu kontenera. Aplikacje utworzone przy pomocy JHipster posiadają wiele zależności języka Java oraz tworzonego interfejsu użytkownika. W przypadku całego tworzonego projektu najwięcej przestrzeni dyskowej zajmuje katalog `node_modules`. Na rzecz tego porównania katalog ten zostanie pominięty w wynikach. Mikroserwisy tworzone przy pomocy prototypu nie zawierają aplikacji interfejsu użytkownika oraz nie wymagają żadnych zależności z tego zakresu. Przedstawione wyniki w tabeli 2 ukazują ilość zajmowanej przestrzeni dyskowej projektów każdej aplikacji.

Tabela 2. Porównanie wykorzystywanej przestrzeni dyskowej wygenerowanych projektów – źródło: opracowanie własne

	JHipster	Prototyp
Projekt mikroserwisu zamówienia	163 MB	27,3 MB
Projekt mikroserwisu produkt	163 MB	27,1 MB
Projekt mikroserwisu koszyk	163 MB	27,3 MB
Aplikacja brama z panelem administracyjnym	382 MB	-
Suma wykorzystanej przestrzeni dyskowej	~872 MB	~82 MB

Narzędzia oferują także utworzenie obrazów kontenerów. Wyniki rozmiarów obrazów kontenera zaprezentowane zostały w tabeli 3. Różnica w wybudowanych obrazach jest mniejsza niż w przypadku projektów. Kontenery utworzone przy pomocy prototypu będą zajmować około 70% przestrzeni dyskowej w porównaniu do kontenerów utworzonych przy pomocy narzędzia JHipster. Prototyp tworzy aplikacje z bazą danych SQLite wewnątrz kontenera. W przypadku zastosowania zewnętrznej bazy danych mikroserwisy dodatkowo musiałby skorzystać z obrazu kontenera bazy danych. W tym przypadku całkowita różnica wykorzystywanej przestrzeni dyskowej pomiędzy konkurencyjnymi generatorami byłaby mniejsza.

Tabela 3. Porównanie wykorzystywanej przestrzeni dyskowej obrazów kontenerów infrastruktury – źródło: opracowanie własne

	JHipster	Prototyp
Obraz mikroserwisu zamówienia	333,56 MB	235,63 MB
Obraz mikroserwisu produkt	333,56 MB	235,44 MB
Obraz mikroserwisu koszyk	333,56 MB	235,63 MB
Obraz aplikacji bramy	346,76 MB	-
Obraz rejestru JHipster	344,5 MB	-
Obraz bazy danych MySQL	520,74 MB	-
Suma wykorzystanej przestrzeni dyskowej	~2,2 GB	~707 MB

6.2.4 Implementacja dodatkowych funkcjonalności

Projekty tworzone przez narzędzie JHipster posiadają wiele bibliotek, które pomagają w zarządzaniu cała infrastrukturą. Dostępne są statystyki, systemy odkrywania usługi oraz replikowania. Są to elementy bardzo potrzebne w przypadku tej architektury. Duża ilość zależności powoduje także chaos w strukturze projektu przez co indywidualny rozwój aplikacji mikroserwisowych jest utrudniony.

Założone wymagania komunikacji między mikroserwisami nie zostały spełnione przez generator konkurencyjny. Dodatkowo ograniczenie braku możliwości utworzenia encji dostępnych w osobnych mikroserwisach oznacza wymagane dodatkowe oczyszczenie projektów z modeli które nie są potrzebne.

W celu umożliwienia komunikacji między mikroserwisami wszystkie zapytania muszą być wysyłane przez bramę, która oddeleguje zadanie do wybranej aplikacji. Oznacza to dodatkowe zmiany w aplikacji bramy, dostępny panel administracyjny musi zostać dostosowany do nowego sposobu zarządzania encjami.

6.2.5 Aktualizacja projektu generatorem

Dużą zaletą tworzonego projektu przy użyciu narzędzia JHipster są metadane które pozwalają na pominięcie generowania wszystkiego od początku. Generator przechowuje tam informacje o strukturze plików, dostępnych encjach oraz zależnościach. Ponowne uruchomienie narzędzia informuje użytkownika o zmianach jakie mają nastąpić i oferuje decyzję co zrobić z plikami, które mają zostać nadpisane.

Prototyp aktualnie posiada możliwość tylko nadpisywania plików wygenerowanych i dodawanie nowych. Uniemożliwia to programiście aktualizację projektu po zmianie konfiguracji generatora. Wszystkie zmiany wprowadzone w wygenerowane pliki zostaną nadpisane. Częściowym rozwiązaniem tego problemu jest wykorzystany modyfikator klasy `partial` który oznacza, że dodatkowe funkcjonalności mogą zostać zaimplementowane w osobnym pliku.

7. Problemy związane z automatycznym generowaniem kodu

Generowanie projektów jest skomplikowanym procesem. Nawet w przypadku dobrego zdefiniowania wymagań przy pomocy konfiguracji narzędzie musiało wprowadzać uproszczenia. Te uproszczenia zazwyczaj oznaczają dodatkową pracę użytkownika. Dodatkowo nie istnieje złoty środek między ilością dostępnej konfiguracji a łatwością użycia. W przypadku zbyt dużego skomplikowania obsługa generatora może okazać się nieopłacalna co oznacza, że użytkownik zdecyduje się na zaimplementowanie projektu samemu.

7.1. Różnorodność konstrukcji języków

Wszystkie platformy programistyczne posiadają pewne podstawowe koncepcje, które są niezmienne. Jest to część elementów które są łatwo przenaszalne i mogą zostać zaimplementowane wewnątrz generatora. Struktura generatora też może zostać przygotowana na takie konstrukcje wystarczająco abstrakcyjnie co umożliwi implementację tworzenia projektów w wybranym języku. W takim przypadku wstępnym ograniczeniem jest ilość takich wspólnych czynników.

Generator musi brać także pod uwagę każdy wyjątek pomiędzy dostępnymi platformami. Takim wyjątkiem może być nawet rzecz trywialna jakim jest sposób uruchomienia projektu. Przykładem są języki skryptowe które zazwyczaj nie wymagają procesu kompilacji w przeciwieństwie do języków kompilowanych. To oznacza dodatkową wyjątkową obsługę tworzenia projektu. Ilość dostępnych języków oznacza na zdecydowanie się na półśrodki implementacyjne tworzonych projektów.

Prototyp utworzony na rzecz tej pracy posiada klasy abstrakcyjne za którymi są ukryte implementacje dotyczące tworzenia projektu na platformie .NET. Implementacja generowania w języku podobnej klasy nie powinna być znacząco skomplikowana. Umożliwienie obsługi tworzenia projektów w językach nie obiektowych lub skryptowych może być większym wyzwaniem.

W przypadku większej ilości wspieranych języków i platform oznacza też więcej wymaganych elementów do uruchomienia generatora. Prototyp ukrywa za abstrakcją obsługę uruchamianych poleceń. Może to być pomocne w przyszłości, aby utworzyć implementację opartą o środowiska programistyczne dostarczane z kontenerów Docker. Aktualnie, aby utworzyć projekt wymagane jest środowisko programistyczne .NET. Środowisko programistyczne w kontenerze oznacza brak wymaganej ingerencji użytkownika w system, na którym uruchamiany jest generator. Przy takiej implementacji potrzebny jest Python, aby umożliwić uruchomienie narzędzia oraz Docker który umożliwi tworzenie projektów oraz ich uruchomienie.

Biblioteki oraz platformy programistyczne posiadają odmienne podejścia do tworzenia struktury katalogów oraz plików. Układ tworzony przez generator nie zawsze będzie w pełni taki jak programista preferuje. Dodatkowo umożliwienie tworzenia odmiennych struktur plików oznacza prawdopodobnie dużo większe skomplikowanie implementacji generatora. W przypadku prototypu zastosowano najnowszy układ przygotowany na rzecz .NET 6 minimalne API (ang. *miniamal API*). Platforma .NET oferuje swój własny bazowy *scaffolding* który tworzy strukturę znaną większości programistów tego środowiska.

7.2. Uproszczenia

JHipster jest narzędziem, które stara się uniknąć uproszczeń struktury wewnętrznej tworzonego projektu. Posiada wydzieloną warstwę serwisów oddzieloną od metod obsługujących zapytanie. Dostępne są także repozytoria do zarządzania modelami.

W przypadku prototypu rozwiązanie jest bardzo uproszczone, metody obsługujące zapytanie dokonują bezpośrednio operacji na kontekście bazy danych. To rozwiązanie jest optymalne w przypadku prototypu. W przyszłości implementacja powinna działać podobnie jak w przypadku JHipster. Kontekst bazy danych już aktualnie jest wzorcem repozytorium, dlatego nie powinien zostać dodatkowo ukrywany za kolejnym wzorcem [34].

Każdy z generatorów ma pewne uproszczenia, zazwyczaj są związane one z zbyt dużym skomplikowaniem implementacji integracji dodatkowych komponentów. Przykładem takiego uproszczenia w przypadku narzędzia JHipster jest brak możliwości komunikacji między mikroserwisami. Implementacja takiego sposobu działania jest skomplikowana i wymaga tworzenia projektów w częściach. Dodatkowym problemem, dlaczego taki rodzaj komunikacji nie został dozwolony jest zapewne panel administracyjny który musiałby być dodatkowo dostosowany. Limitacja ta wymusza także na tworzenie encji, które nie posiadają relacji z encjami w innych mikroserwisach.

Prototyp posiada możliwość wymiany komunikatów między mikroserwisami. Problemem w tym przypadku jest ilość różnych typów komunikacji. Aktualnie prototyp zawiera tylko możliwość wywołania metody w innym mikroserwisie, przesłanie danych oraz odebranie danych wraz przypisaniem do obiektu nadrzędnego. Jest to bardzo ograniczony zbiór możliwości, aby umożliwić bardziej zaawansowany sposób tworzenia komunikacji wymagane byłoby dodane bardziej skomplikowanego generatora metod obsługujących zapytanie.

Dodatkowym brakiem obu generatorów jest brak wykorzystania kolejek, subskrypcji oraz możliwości publikowania komunikatów na wspólną szynę komunikacji. Takie podejście ułatwia sposób komunikacji mikroserwisów między sobą oraz nie wymusza wskazywania bezpośrednio odbiorcy wysłanego komunikatu.

7.3. Uwierzytelnianie i autoryzacja danych

W przypadku prototypu generator nie tworzy żadnego rodzaju uwierzytelniania. Jest to jedna z pierwszych rzeczy, które są implementowane na początku projektu. Dodanie uwierzytelniania do pojedynczej aplikacji nie jest problemem. Architektura mikroserwisów zazwyczaj nie posiada wprost określonego rozwiązania tego problemu. Jednym z popularniejszych rozwiązań jest utworzenie osobnego mikroserwisu zajmującego się zarządzaniem użytkownikami i ich dostępami. Taka aplikacja wymienia dane użytkownika, na identyfikatora który umożliwia komunikację z pozostałymi mikroserwisami. Prototyp generatora musi zostać rozwinięty, aby dodatkowo oprócz mikroserwisów opisanych w konfiguracji tworzyć dodatkowy mikroserwis autoryzacyjny. Skomplikuje to także implementację wszystkich pozostałych mikroserwisów, przed umożliwieniem dostępu do przechowywanych danych użytkownik musi najpierw zostać zidentyfikowany przez mikroserwis uwierzytelniający.

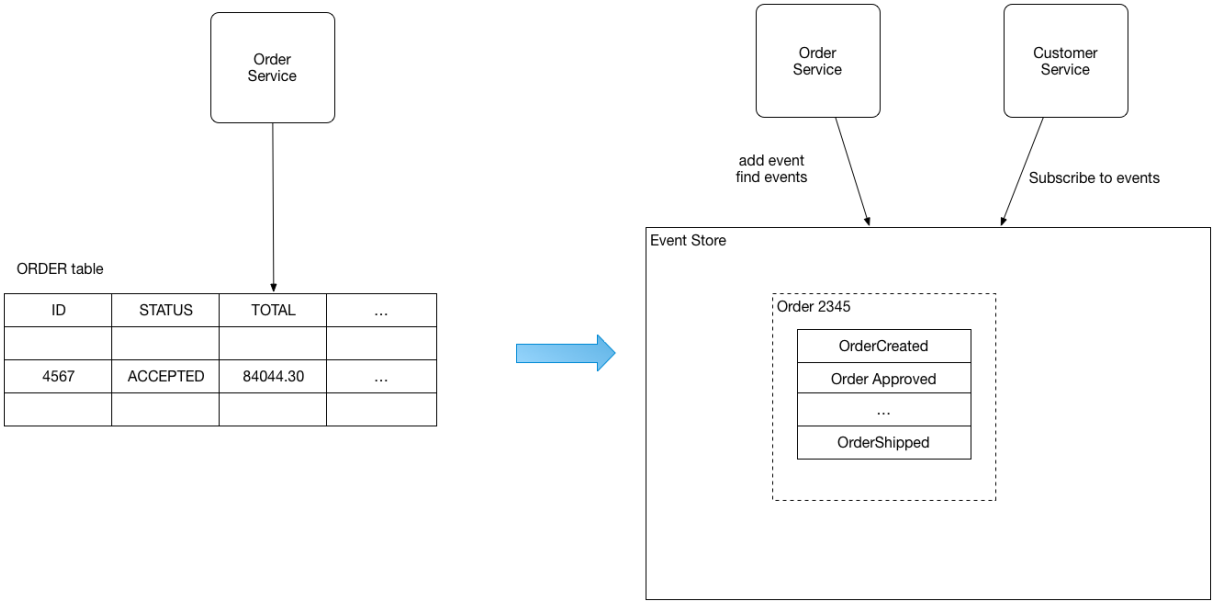
Opisane wyżej rozwiązanie jest częściowo zrealizowane w przypadku narzędzia JHipster. Generator obsługę uwierzytelniania obsługuje na poziomie bramy mikroserwisów. Mikroserwisy dostają komunikat tylko gdy aplikacja bramy na to pozwoli.

7.4. Spójność danych

Rozproszona infrastruktura aplikacji mikroserwisowych oznacza możliwość powielania aplikacji w celu obsłużenia większej ilości zapytań. W takiej sytuacji istnieje wiele potencjalnych sytuacji na brak spójności danych. Aktualizacja danych powinna być niezawodna oraz atomowa.

Oba generatory nie posiadają obsługi szyny komunikacyjnej. Taki sposób komunikacji jest popularny w przypadku podejścia tworzenia komunikacji opartej o wydarzenia. Wydarzenia oraz ich statusy mogą być zapisywane w bazie. Umożliwia to łatwe wprowadzanie zmian oraz ich wycofywanie.

W sytuacji istnienia dużej ilości aplikacji działających obok siebie istnieje szansa, że niektóre nie są w stanie przetworzyć danego zapytania. W takim przypadku rozpoczęte zapytanie może zostać odłożone na później. Stan zapisanego eventu w bazie zawiera informacje o tym jakie kolejne kroki są wymagane, aby kontynuować przerwany wcześniej proces. Dodatkową zaletą takiego wzorca jest posiadanie pełnych informacji przebiegu zmian dotyczących obiektów. Pozwala to na znalezienie potencjalnych błędów w danych oraz dokonanie korekty. Na rysunku 18 przedstawiono przykład jak powinny być przechowywane dane w przypadku wzorca *event sourcing*.



Rysunek 18. Diagram przedstawiający sposób przechowywania danych w przypadku wzorca *event sourcing* - źródło: [35].

Implementacja tego typu wzorca jest skomplikowana dla programisty. Wymaga zmiany podejścia myślenia oraz sposobu przechowywania danych. Utworzenie działającego generatora, który zawiera możliwość generowania komunikacji opartej na wydarzeniach komplikuje problem jeszcze bardziej. W przypadku takiego wzorca przechowywane dane są mniej zorganizowane niż w przypadku normalnego podejścia, gdzie obiekt oznacza dodatkowy wiersz w tabeli. Rodzaj poszczególnych stanów dla danego modelu jest też zmienny. Przykładowo dla zamówienia stan może przyjmować wartości: utworzony, zaakceptowany, wysłany, zrealizowany. Implementacja obsługi każdego stanu nie jest możliwa do wygenerowania automatycznie bez wiedzy intencji użytkownika.

W przypadku wzorca obsługi wydarzeń oznacza także znaczny wzrost ilości zapisywanych danych. Aktualna implementacja prototypu zawiera bazę danych SQLite która znajduje się wewnątrz aplikacji. Dodanie podstaw wymienionego wzorca oznacza także potrzebę przeniesienia przechowywanych danych do zewnętrznej usługi uruchomionej obok aplikacji. Przykładowym rozwiązaniem mógłby być serwer bazodanowy Microsoft SQL lub MySQL uruchomiony jako kontener wraz z aplikacją mikroserwisową.

8. Podsumowanie

Przygotowany prototyp oferuje tworzenie wybranej przez użytkownika ilości aplikacji mikroserwisowych. Podstawowym wymaganiem każdego generatora jest utworzenie projektu zawierającego modele oraz warstwę przechowywania danych. Pomocnym w tym przypadku było narzędzie wspomagające Entity Framework Core.

Aktualnie dostępne konkurencyjne generatory projektów oferują tylko komunikację wewnątrz aplikacji. Komunikacja między mikroserwisami jest niedostępna. W tym przypadku wymagana jest dodatkowa praca implementacyjna wykonywana manualnie. Brak tej funkcjonalności może okazać się powodem do zrezygnowania z wykorzystania konkurencyjnych rozwiązań na rzecz przygotowanego prototypu. Generator utworzony przez autora pracy pozwala na definiowanie rodzaju oraz ścieżek metod obsług komunikatów. Komunikaty mogą być wymieniane między mikroserwisami w celu utworzenia większej spójności całej infrastruktury.

Oferowana komunikacja nie jest w pełni zgodna z standardami architektury mikroserwisów. W celu zbudowania bezpiecznej oraz stabilnej komunikacji, wymiana komunikatów musiałaby odbywać się przy pomocy zarządcy komunikacji oraz kolejek. Aktualna komunikacja działa na zasadzie zapytań oraz odpowiedzi. Każda z aplikacji nadawczej wymaga poprawnego działania aplikacji odbiorczej. Obsługa w pełni asynchroniczna wymaga zmiany podejścia implementacji oraz jest skomplikowana w generowaniu.

Tworzone projekty przy pomocy prototypu posiadają niedużą strukturę oraz krótkie pliki źródłowe. Konkurencyjny generator JHipster tworzy projekty o prawie ponad 10 razy większej ilości plików oraz ponad 10 razy większej ilości linii kodu źródłowego. Mniejszy projekt jest łatwiejszy do utrzymania i zrozumienia. Generator powinien skupiać się na tworzeniu środowiska najbardziej przystępnego użytkownikowi końcowemu, w tym przypadku programiście, który będzie dodawać nowe funkcjonalności.

Nieduże tworzone projekty są wynikiem ograniczonych możliwości konfiguracji. Aplikacje końcowe posiadają prostą bazę plikową SQLite. W tym przypadku dane przechowywane są obok aplikacji, docelowo powinno to zostać przeniesione do osobnego serwera bazodanowego. Rozdzielenie przechowywanych danych oraz aplikacji oznacza także wymagane rozwinięcie konfiguracji. Należy umożliwić użytkownikowi decyzję, gdzie dane powinny być przechowywane.

Konkurencyjne narzędzia posiadają często edytory graficzne przygotowane na rzecz tworzenia konfiguracji. Pozwala to na zwizualizowanie struktury aplikacji oraz powiązań między encjami. Aktualna konfiguracja prototypu zapisana jest w formacie JSON, przygotowanie takiego edytora wizualnego nie powinna być skomplikowana.

Generowanie projektów jest skomplikowanym procesem i zazwyczaj oznacza dużo ograniczeń lub decyzji, które mogą teoretycznie nie być odpowiednie dla użytkownika. Autor pracy uważa, że nie istnieje złoty środek pomiędzy ilością dostępnych możliwości takiego narzędzia a łatwością jego używania. Generatory powinny być specyficzne do danej dziedziny działania i nie powinny obsługiwać każdego rodzaju przypadku. Konkurencyjny JHipster został utworzony jako generator aplikacji monolitycznych. Dodano w późniejszej wersji tworzenie mikroserwisów. Efektem tego jest dużo problemów z relacjami między modelami lub mikroserwisami.

Najważniejszym elementem jest utworzenie projektu, który jest przystępny dla osoby zajmującej się jego dostosowywaniem. Utworzenie zbyt skomplikowanego projektu może wiązać się z decyzją rezygnacji z generatora. Programista, który musi zrezygnować z dużej części generowanego projektu wybierze tworzenie projektu od początku manualnie. Taki podstawowy projekt może zostać zamieniony we wzorzec dla pozostałych projektów. Programista w ten sposób ułatwi sobie tworzenie następnych aplikacji oraz będzie to wykonane według jego wymagań.

9. Bibliografia

[1] „Round 20 results - TechEmpower Framework Benchmarks,” 08 02 2021. [Online]. Available: https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=composite.

[2] S. Nicoll, D. Syer i M. Bhave, „Spring Initializr Reference Guide,” [Online]. Available: https://docs.spring.io/initializr/docs/current/reference/html/. [Data uzyskania dostępu: 24 05 2022].

[3] „Custom templates for dotnet new - .NET CLI,” 18 12 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/tools/custom-templates.

[4] M. J. Price, „Working with Databases Using Entity Framework Core,” w *C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development - Fourth Edition*, Packt Publishing, 2019.

[5] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt i R. Karri, „Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions,” 2021.

[6] GitHub Inc., „GitHub Copilot · Your AI pair programmer,” [Online]. Available: https://copilot.github.com/. [Data uzyskania dostępu: 2022 05 24].

[7] Tabnine, „Plans & Pricing | Tabnine,” [Online]. Available: https://www.tabnine.com/pricing. [Data uzyskania dostępu: 2022 05 24].

[8] T. Vanek, „GitHub - tomi-vanek/microts: Microservice code generator: from OpenAPI (Swagger) REST API specification to TypeScript project with Docker.,” [Online]. Available: https://github.com/tomi-vanek/microts. [Data uzyskania dostępu: 06 06 2022].

[9] JHipster, „JHipster Domain Language,” [Online]. Available: https://www.jhipster.tech/jdl/intro. [Data uzyskania dostępu: 07 06 2022].

[10] JHipster, „API Gateway,” [Online]. Available: https://www.jhipster.tech/api-gateway/. [Data uzyskania dostępu: 07 06 2022].

[11] JHipster, „Creating microservices,” [Online]. Available: https://www.jhipster.tech/creating-microservices/. [Data uzyskania dostępu: 07 06 2022].

[12] Python Software Foundation, „Applications for Python,” [Online]. Available: https://www.python.org/about/apps/. [Data uzyskania dostępu: 2022 05 24].

[13] „Frozen instances - Data Classes,” [Online]. Available: https://docs.python.org/3/library/dataclasses.html#frozen-instances. [Data uzyskania dostępu: 2022 05 24].

[14] Python Packaging Authority, „Installing packages using pip and virtual environments,” [Online]. Available: https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/. [Data uzyskania dostępu: 2022 05 24].

[15] The Pallets Projects, „Jinja Documentation (3.1.x),” [Online]. Available: https://jinja.palletsprojects.com/en/3.1.x/api/#high-level-api. [Data uzyskania dostępu: 15 06 2022].

[16] A. Olsson i R. Voss, w *Git Version Control Cookbook*, Pact Publishing, 214, p. 1.

[17] Microsoft, „Visual Studio Code - Code Editing. Redefined,” [Online]. Available: https://code.visualstudio.com/. [Data uzyskania dostępu: 23 05 2022].

[18] Stack Overflow, „Integrated development environment - Stack Overflow Developer Survey 2021,” 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-new-collab-tools.

[19] Bhosale, T. Patil i P. Patil, „SQLite: Light Database System,” *International Journal of Computer Science and Mobile Computing*, tom 4, nr 4, pp. 882-885, 2015.

[20] Stack Overflow, „Databases - Stack Overflow Developer Survey 2021,” 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-databases.

[21] I. C. Education, „Containerization Explained,” 23 06 2021. [Online]. Available: https://www.ibm.com/cloud/learn/containerization.

[22] R. McKendrick i S. Gallagher, Mastering Docker - Second Edition, Packt Publishing, 2017.

[23] I. Postman, „Postman API Platform | Tools,” [Online]. Available: https://www.postman.com/product/tools/. [Data uzyskania dostępu: 27 05 2022].

[24] „Get started with Swashbuckle and ASP.NET Core,” 14 04 2022. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle.

[25] C. Richardson, „Pattern: Messaging - Microservices.io,” [Online]. Available: https://microservices.io/patterns/communication-style/messaging.html. [Data uzyskania dostępu: 2022 05 25].

[26] „Low-code development platform - Wikipedia,” 17 05 2022. [Online]. Available: https://en.wikipedia.org/wiki/Low-code_development_platform.

[27] M. Woo, „The Rise of No/Low Code Software Development—No Experience Needed?,” *Engineering*, tom 6, nr 9, pp. 960-961, 2020.

[28] „Create a custom connector from scratch,” [Online]. Available: https://docs.microsoft.com/en-us/connectors/custom-connectors/define-blank. [Data uzyskania dostępu: 27 05 2022].

[29] „JSON,” [Online]. Available: https://www.json.org/json-en.html. [Data uzyskania dostępu: 2022 05 24].

[30] A. Wright, H. Andrews i B. Hutton, „JSON Schema Validation: A Vocabulary for Structural Validation of JSON,” 28 01 2020. [Online]. Available: http://json-schema.org/draft/2020-12/json-schema-validation.html#rfc.section.6. [Data uzyskania dostępu: 2022 05 25].

[31] „Use the Azure CLI to create Service Bus topics and subscriptions - Azure Service Bus,” [Online]. Available: https://docs.microsoft.com/pl-pl/azure/service-bus-messaging/service-bus-tutorial-topics-subscriptions-cli. [Data uzyskania dostępu: 2022 05 25].

[32] D. i. i. A. Core, 06 05 2022. [Online]. Available: https://docs.microsoft.com/en-US/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0. [Data uzyskania dostępu: 26 05 2022].

[33] „Circular dependency - Wikipedia,” 05 12 2021. [Online]. Available: https://en.wikipedia.org/wiki/Circular_dependency. [Data uzyskania dostępu: 2022 05 27].

[34] J. P. Smith. [Online]. Available: https://www.thereformedprogrammer.net/is-the-repository-pattern-useful-with-entity-framework-core/. [Data uzyskania dostępu: 14 06 2022].

[35] C. Richardson. [Online]. Available: https://microservices.io/patterns/data/event-sourcing.html. [Data uzyskania dostępu: 14 06 2022].

10. Wykaz rysunków

11. Wykaz listingów

12. Wykaz tabel