Porównanie zabezpieczeń przed botami w istniejących aplikacjach internetowych

(Comparing bot protection in existing web applications)

Mateusz Kisiel

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

29.01.2024

Streszczenie

Z powodu rosnącej cyfryzacji, coraz więcej zadań wykonujemy z użyciem komputera i Internetu, a coraz mniej w formie fizycznej i papierowej. Każdy z nas wyklikuje wiele powtarzalnych akcji, które zabierają nam coraz więcej czasu. Warto byłoby spróbować ten czas zaoszczędzić. Dodatkowo, cyfryzacja otwiera także furtkę na możliwości wykorzystania systemów w sposób nie przewidziany przez twórców, które mogą znacznie uprościć nam życie.

Celem tej pracy jest stworzenie maksymalnie uniwersalnego i modułowego systemu, opartego o mikroserwisy, który pozwoli nam w łatwy sposób konfigurować różnego rodzaju boty oraz nimi zarządzać. Możemy definiować kiedy jakie boty mają się uruchamiać, a także relacje między nimi. Dane zwrócone z jednego bota mogą zostać automatycznie przekazane do kolejnego, umożliwiając płynne kolejkowanie akcji.

With the rise of digitization, an increasing number of tasks are being executed using computers and the Internet rather than through physical or paper-based means. We quite often find ourselves performing numerous repetitive actions that consume a growing amount of our time. Identifying ways to save this time can make a pretty big difference. Furthermore, digitization opens up opportunities to use systems in unanticipated ways that can greatly simplify our lives.

The aim of this thesis is to develop a highly universal and modular system based on microservices. This system enables the easy configuration and management of various types of bots. It allows for the scheduling of bots and the establishment of relationships between them. Data generated by one bot can be automatically transferred to another, facilitating a seamless operational flow.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Motywacja

Podczas studiów i szkoły średniej napisałem wiele różnych botów do aplikacji internetowych. Za każdym razem pojawiał się ten sam problem: „Gdzie je uruchamiać i jak nimi zarządzać?". Z początku miałem je uruchomione na moim komputerze. Rozwiązanie to posiadało szereg wad: Komputer był uruchomiony całą dobę. Musiałem pilnować czy boty działają. Dodatkowo zużywały one zasoby komputera, z którego korzystałem, spowalniając go. Następnie postanowiłem uruchomić niektóre z nich na serwerze VPS. To rozwiązanie było trochę lepsze, ale nadal nie było idealne. Każdego nowego VPS'a trzeba skonfigurować, żeby móc uruchomić na nim boty, co zabiera czas. Natomiast późniejsze debugowanie botów i zarządzanie nimi, jest dosyć niewygodne. Później, po wyczerpaniu wersji próbnej, musiałbym płacić za VPS lub konfigurować go od nowa, co mnie na tyle zniechęciło, że w pewnym momencie zrezygnowałem.

Projekt opisany w tej pracy rozwiązuje wszystkie powyższe problemy, poza kosztami utrzymania serwera VPS lub domowego (czego niestety nie obejdziemy). W ramach pracy nie implementuję wielu botów, skupiam się głównie na systemie, który jest maksymalnie otwarty na rozszerzenia i nowe boty. Wystarczy przesłać plik .dll z nowym botem na jeden z istniejących serwerów uruchamiających boty.

1.2. Informacje o aplikacji

Aplikację można zobaczyć, wchodząc na adres: https://bot.matik.live
(Aby móc korzystać z aplikacji należy najpierw utworzyć sobie konto)

Kod całego projektu można znaleźć na Github'ie pod adresem:
https://github.com/matik001/SuperBotManager

Repozytorium do automatycznej konfiguracji VPS'a znajduje się pod adresem:
https://github.com/matik001/VPSManager

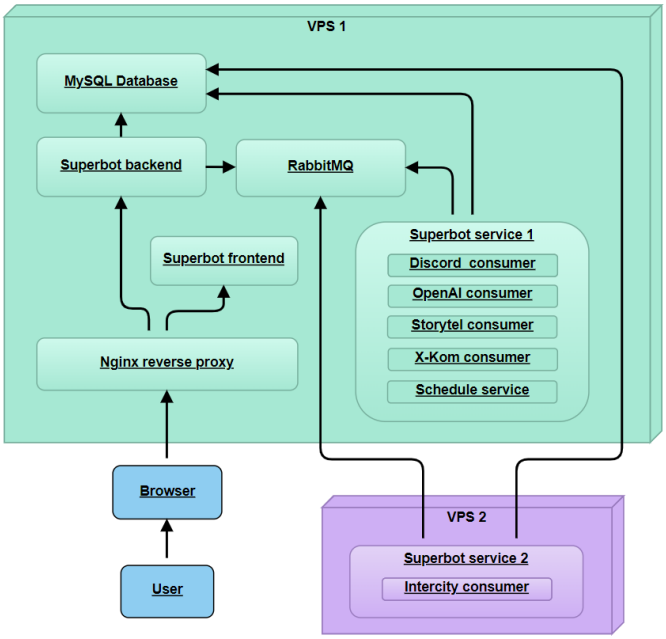
1.3. Architektura aplikacji

Projektując aplikację, chciałem, aby była ona jak najbardziej modułowa i możliwa do rozszerzania o nowe boty. W tym celu postanowiłem zastosować architekturę mikroserwisów, gdzie każdy mikroserwis odpowiadałby za wykonywanie akcji konkretnego bota.

Idealnie w tym przypadku nada się RabbitMQ [1], który uporządkuje nam komunikację pomiędzy mikroserwisami. Każdy z nich będzie miał stworzoną osobną kolejkę z listą swoich akcji oczekujących na uruchomienie. Dzięki takiemu podejściu nawet jeżeli jakiś mikroserwis zawiedzie, akcje dalej mogą się prawidłowo zakolejkować, i gdy mikroserwis wznowi działanie, prawidłowo uruchomi zaległe akcje. RabbitMQ zwiększa modułowość i niezależność mikroserwisów (nie muszą one wiedzieć o istnieniu innych mikrousuług, wykonują tylko swoją pracę w ramach swojej kolejki), dzięki czemu łatwiej jest wyłączać lub dodawać nowe mikroserwisy.

Podejście to daje też dużo większe możliwości w skalowaniu aplikacji. W przypadku monolitu konieczne jest skalowanie wertykalne (zwiększanie zasobów pojedynczego komputera). Z początku jest to dosyć wygodne podejście, jednak później, gdy obciążenie serwera zaczyna rosnąć, mogą pojawić się problemy z wydajnością, które są bardzo kosztowne do rozwiązania. Rozwiązaniem tego problemu są mikroserwisy, które umożliwiają nam skalowanie horyzontalne. Możemy dołożyć dodatkowe komputery. Jest to tańsze i znacznie zwiększa potencjał wydajnościowy naszej aplikacji. Wąskim gardłem teraz będzie serwer bazy danych i serwer RabbitMQ, jednak granica przesunięta zostanie dużo dalej niż początkowo. Możemy także dostosować ilość mikroserwisów do naszych potrzeb. Jeżeli widzimy, że jakiś mikroserwis nie nadąża z wykonywaniem akcji (ilość elementów w kolejce RabbitMQ stale rośnie), możemy dołożyć drugi taki sam serwis, aby zwiększyć prędkość przetwarzania akcji danego typu.

Początkowo miałem zamiar, aby każdy konsument akcji RabbitMQ uruchomiony był jako osobny serwis, w osobnym kontenerze Docker'owym [12]. Otrzymalibyśmy w ten sposób wszystkie zalety, o których pisałem wcześniej. Docelowo w idealnym świecie każdy mógłby być nawet uruchomiony na osobnej maszynie.



Rysunek 1.1: Architektura aplikacji

Rozwiązanie to miałoby jednak dużą wadę. Gdyby było 100 różnych typów akcji, wymuszałoby to utworzenie 100 rożnych kontenerów, co znacznie utrudniłoby zarządzanie nimi. Dodatkowo rozwiązanie to byłoby bardzo zasobożerne. Nie miałem też zamiaru używać więcej niż jednego VPS'a.

Postanowiłem połączyć zalety monolitu oraz mikroserwisów i dać użytkownikowi swobodę, w jaki sposób chce podzielić konsumentów na różne serwisy. Projekt SuperbotConsumerService pozwala na uruchomienie dowolnej liczby wybranych konsumentów w obrębie jednego serwisu. Na starcie programu wyszukiwane są wszystkie pliki .dll w katalogu programu zawierające odpowiednią nazwę i ładowane jako Assembly. Następnie dla każdego konsumenta z Assembly tworzony jest osobny wątek będący konsumentem danej akcji (więcej o tym w rozdziale SuperbotConsumerService).

Dzięki temu podejściu mamy pewność, że zachowamy wszystkie zalety mikroserwisów(bez problemu dalej możemy uruchomić każdy konsument na osobnej maszynie), a przy tym mamy także możliwości uruchomić wszystkich konsumentów w jednym serwisie na jednym kontenerze. W szczególności, jeżeli mamy tylko jedną maszynę nie ma sensu rozdzielać każdego konsumenta na osobny kontener, ponieważ nie niesie to za sobą żadnych zalet.

Obecnie system został wdrożony w następujący sposób jak na obrazku 1.1. Strzałka oznacza, że jeden miroserwis łączy się do drugiego. Dlaczego bot do Intercity jest w osobnej maszynie, tłumaczę w rozdziale o Intercity.

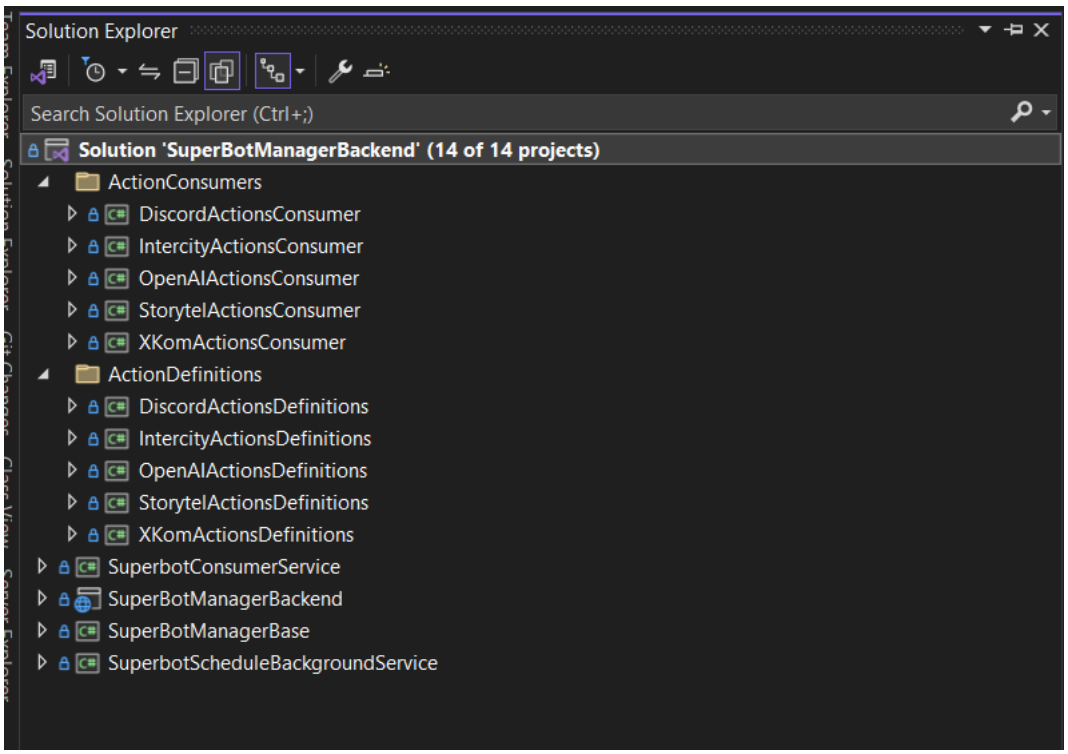
1.4. Podział na projekty

Aplikacja składa się z następujących głównych projektów:

1. SuperBotManagerFrontend – Projekt jest napisany w technologii React [2] z użyciem Typescript'a. (Więcej w rozdziale SuperBotManagerFrontend)
2. SuperBotManagerBackend – Projekt jest napisany w technologii w C# w technologii ASP.NET Core. (Więcej w rozdziale SuperBotManagerBackend)
3. SuperbotConsumerService – znajduje się katalogu SuperbotConsumerService. Jest to projekt, wspomniany wcześniej. Ma za zadanie załadować pliki .dll z serwisami, które mają działać w tle jako osobny mikroserwis). Jest napisany w C#.
4. SuperbotScheduleBackgroundService – Ma za zadanie uruchamiać zaplanowane akcje według harmonogramu stworzonego przez użytkownika. Jest to projekt typu ClassLibrary. Jest uruchamiany przez SuperbotConsumerService, jeżeli dll'ka znajduje się odpowiednim katalogu. Jest napisany w C#.
5. SuperBotManagerBase – Jest to najważniejszy projekt po stronie backend'u. Zawiera on wszystkie wspólne klasy i metody, które mają być możliwe do użycia w obrębie innych projektów Jest napisany w C#.

Dodatkowo każdy rodzaj akcji powinien posiadać dwa dodatkowe projekty. Jeden zawierający funkcję, którą SuperbotConsumerService uruchomi, aby wykonać akcję i drugi z definicją akcji (zawiera informacje, jakie wejście przyjmuje dana akcja i jakie zwraca wyjście). Ten z definicją, w teorii nie jest potrzebny. Można wprowadzić do bazy danych te informacje ręcznie. Istnieje jednak wtedy ryzyko pomyłki oraz trzeba by o tym pamiętać przy tworzeniu nowej bazy. Jeżeli dodamy plik .dll z definicją do katalogu, w którym jest SuperBotManagerBackend, serwer automatycznie wykona seeding w bazie danych, tworząc odpowiednie typy akcji. Ten sposób jest bezpieczniejszy i wygodniejszy.

Na obrazku 1.2 widzimy listę projektów konsumentów i definicji dla kilku typów akcji. Warto powiedzieć, że obecnie SuperbotConsumerService ma dodane referencje do projektów z konsumentami, aby przy buildzie, kompilator automatycznie kopiował pliki .dll. konsumentów do katalogu w którym został zbudowany SuperbotConsumerService. Nie jest to jednak konieczne i nic nie stoi na przeszkodzie, by w zupełnie innej solucji zrobić niezależnie konsumenta i wgrać jego .dll na serwer bez dodawania referencji.



Rysunek 1.2: Projekty C#

Rozdział 2.

Funkcjonalności

2.1. Jak uruchomić akcję?

Miałem kilka koncepcji jak najlepiej zaprojektować elementy aplikacji by wygodnie się z niej korzystało. Ostatecznie zdecydowałem się na następujący podział klas reprezentujących akcje:

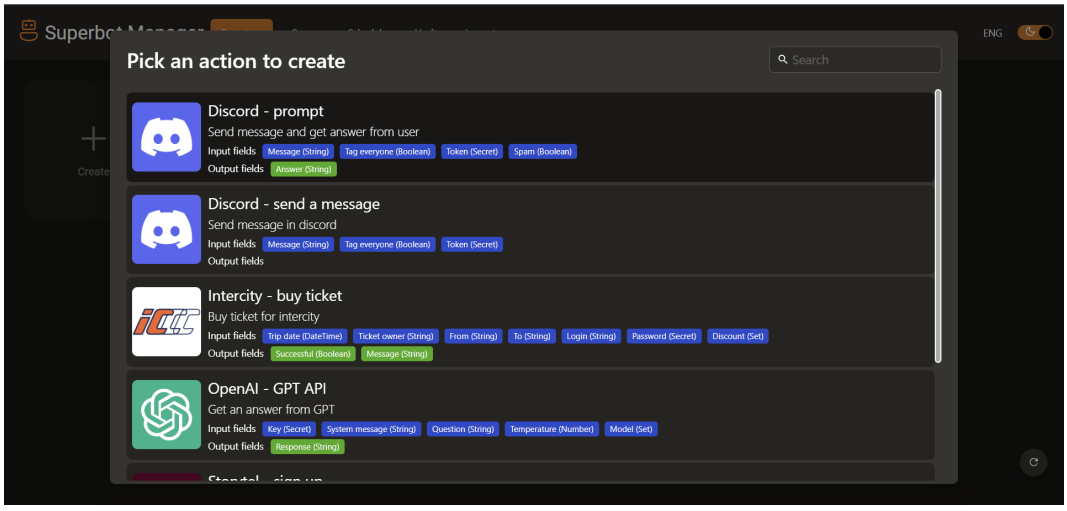
- **ActionDefinition** – Zawiera informację o tym, jak się akcja nazywa, jej opis, ikona, w jakiej kolejce będzie się zapisywać oraz o tym, jakie dane przyjmuje na wejściu, oraz co zwraca.

  Oto przykładowa definicja akcji:

```
1  public static ActionDefinition Prompt { get; } = new
       ActionDefinition()
2  {
3      ActionDefinitionQueueName = "discord-prompt",
4      ActionDefinitionName = "Discord -- prompt",
5      ActionDefinitionDescription = "Send message and get answer from
           user",
6      ActionDefinitionIcon =
           "https://images-eds-ssl.xboxlive.com/image?url=4rt9.lXDC4H_93laV1_eHHFT949fUipzk
7      ActionDataSchema = new ActionDefinitionSchema()
8      {
9          InputSchema = new List<FieldInfo>()
10         {
11             new FieldInfo("Message", FieldType.String, "What message
                   do you want to send?")
12             {
13                 Placeholder = "Enter a message"
14             },
15             new FieldInfo("Tag everyone", FieldType.Boolean, "Do you
                   want to tag @everyone?"),
```

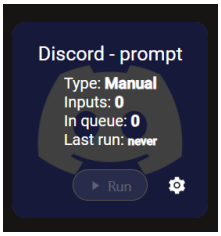
```
16              new FieldInfo("Token", FieldType.Secret, "How to get it:
                    https://discordnet.dev/guides/getting_started/first-bot.html")
17              {
18                  Placeholder = "Enter a token"
19              },
20              new FieldInfo("Spam", FieldType.Boolean, "Do you want to
                    spam it, every 5 seconds?"),
21          },
22          OutputSchema = new List<FieldInfo>()
23          {
24              new FieldInfo("Answer", FieldType.String, "User's reply
                    for the bot's message"),
25
26          },
27      },
28      CreatedDate = DateTime.SpecifyKind(new DateTime(2024, 1, 9),
            DateTimeKind.Utc),
29      ModifiedDate = DateTime.SpecifyKind(new DateTime(2024, 1, 9),
            DateTimeKind.Utc),
30      PreserveExecutedInputs = true
31  };
```

- **ActionExecutor** – Wyobraźmy sobie, że chcemy co jakiś czas uruchamiać akcję z pewnymi danymi. Pierwsze wybieramy sobie, które ActionDefinition nas interesuje w pickerze jak na obrazku 2.1

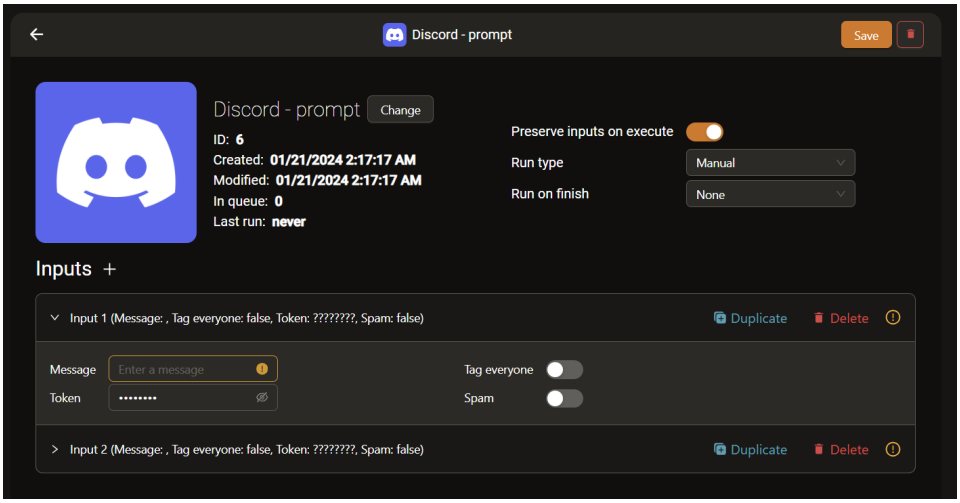


Rysunek 2.1: Wybór akcji do uruchomienia

Właśnie utworzył się nam obiekt ActionExecutor. Chcielibyśmy teraz zdefiniować z jakimi danymi uruchamiać naszą akcję w tym celu wchodzimy w jej ustawienia (biała zębatka)



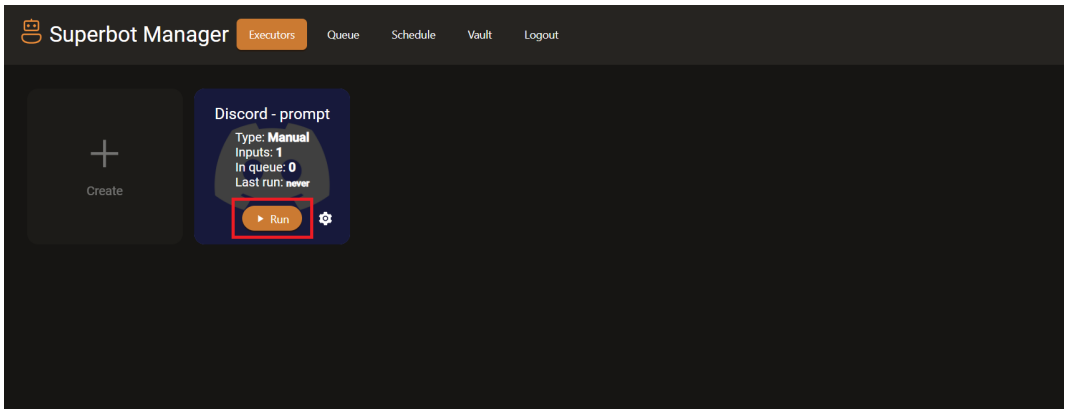
ActionExecutor pozwala nam zdefiniować jakie akcje mają się dodać do kolejki po jego uruchomieniu. Za jednym uruchomieniem możemy dodać wiele akcji tego samego typu.



Rysunek 2.2: ActionExecutor – ustawienia

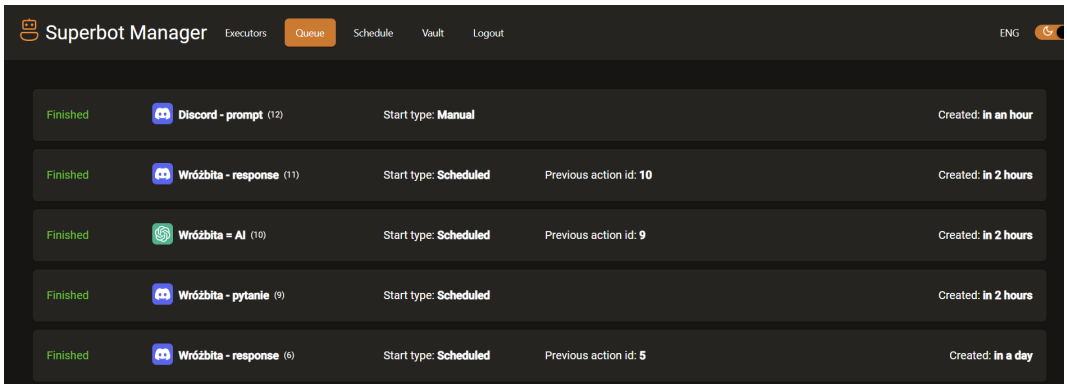
Jak widzimy, w celu edycji wejścia akcji, automatycznie wygenerował się formularz zgodny z naszym schematem ActionDefinition.

- **Action** – Gdy już mamy zdefiniowany executor chcielibyśmy móc go uruchomić.



Rysunek 2.3: Uruchomienie Executora

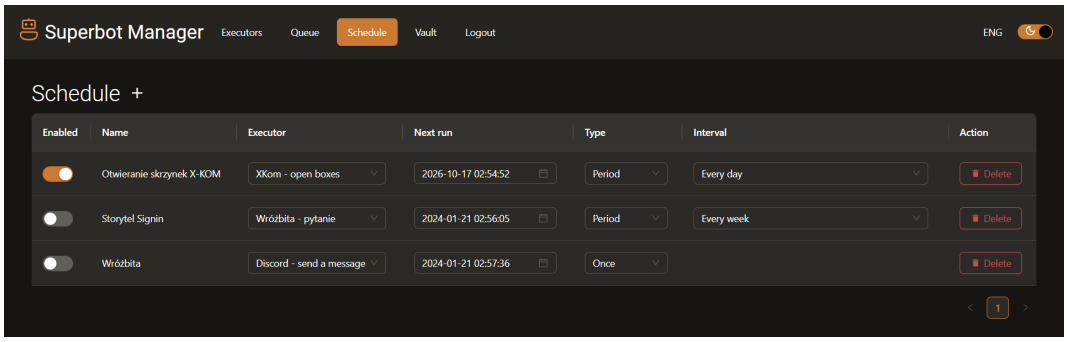
W momencie kliknięcia do kolejki doda się tyle akcji, ile zdefiniowanych miał Executor. W zakładce queue możemy zobaczyć, że akcja dodała się do kolejki i została już wykonana (jest już w statusie Finished)



Rysunek 2.4: Kolejka zadań

Istnieją także bardziej zaawansowane mechanizmy jak np. łańcuchowe wykonywanie akcji, ale poruszę je później.

2.2. Harmonogram zadań



Rysunek 2.5: Harmonogram zadań

Wiemy już jak uruchamiać akcje ręcznie. Co jeżeli chcielibyśmy uruchamiać jakąś akcję okresowo np. raz na dzień. Do tego służy moduł harmonogramu zadań. Znajduje się on w zakładce Schedule. Każde zadanie w harmonogramie składa się z następujących właściwości. Każdą z nich możemy dowolnie zmienić.

- Enabled – (zadanie możemy tymczasowo wyłączyć bez konieczności usuwania)
- Name – (nazwa zadania)
- Executor – (jaki executor ma zostać uruchomiony przez nasze zadanie)
- Next run – (kiedy ma nastąpić kolejne uruchomienie naszego executora)

- Type – (możemy wybrać między Period a Once. Jeżeli wybierzemy Once, po wykonaniu stan zadania zmieni się z Enabled true na false)
- Interval – (pozwala zdefiniować co ile czasu Executor ma się uruchamiać. Możemy wybrać jedną z predefiniowanych wartości, lub zdefiniować własny przedział czasu)

Za wykonywanie akcji odpowiada projekt SuperbotScheduleBackgroundService. Odpytuje on co określoną ilość sekund bazę danych (wartość IntervalMs w appsettings.json), aby pobrać listę harmonogramu zadań i sprawdza czy są jakieś akcje do uruchomienia. Jak tak to dodaje je do kolejki i zmienia status w harmonogramie.

2.3. Szyfrowanie danych wrażliwych

Prawie każda akcja, która ma styczność z zewnętrznymi serwisami, posiada jakieś dane wrażliwe. Czy to klucz API, jakiś token, czy hasło do konta. Nie chcielibyśmy, aby takie dane były zapisywane w bazie danych plaintext'em. Nie chcielibyśmy też, aby kiedykolwiek backend te dane odkodował i wysłał przeglądarce w sposób jawny. Co więcej, nie chcielibyśmy nawet, aby RabbitMQ posiadało w swojej kolejce odkodowane dane, gdyż domyślnie komunikuje się on po http, bez szyfrowania i ktoś mógłby je podejrzeć. Dodatkowo wchodząc w RabbitMQ Management, można podejrzeć przesyłane dane.

W ActionDefinition definiujemy jakiego typu jest nasze pole. Na obrazku 2.6 znajduje się lista dostępnych typów (nic nie szkodzi, aby dodać ich więcej). Już część z nich widzieliśmy w przykładowej definicji ActionDefinition 2.1.

```
public enum FieldType
{
    String,
    Number,
    Secret,
    DateTime,
    Date,
    Boolean,
    Json,
    Set,
    ExecutorPicker
}
```

Rysunek 2.6: Typy pól

Jak widzimy, jednym z typów pola jest typ Secret. Jeżeli w ActionDefinition typ pola ustawiony jest właśnie na niego, to we wszystkich miejscach, gdzie normalnie znalazłaby się wartość pola, zapisywany jest tylko GUID odpowiedniego rekordu z tabeli secret. Jest to uwzględniane w tabeli actionexecutor oraz tabeli action. Również przy dodawaniu akcji do kolejki jest ona pierwsze szyfrowana, przesyłana, a następnie konsument po odbiorze ją odszyfrowuje. Również przy łańcuchowym wykonywaniu akcji, do następnej akcji przesyłane są także wejścia poprzedniej z wyjątkiem pól typu secret.

Z początku miałem zamiar, aby backend przy wysyłaniu użytkownikowi ActionExecutor'ów, po prostu maskował secrety jakimś tekstem np. ??????. Wydawało mi się to rozwiązanie możliwe do zrealizowania, ponieważ jesteśmy w stanie rozpoznać czy użytkownik edytował pole (jeżeli jest dalej równe masce, to znaczy, że go nie edytował i nie chcemy zmieniać sekretu, a jeżeli jest różne, to znaczy, że powinniśmy zaktualizować secret).

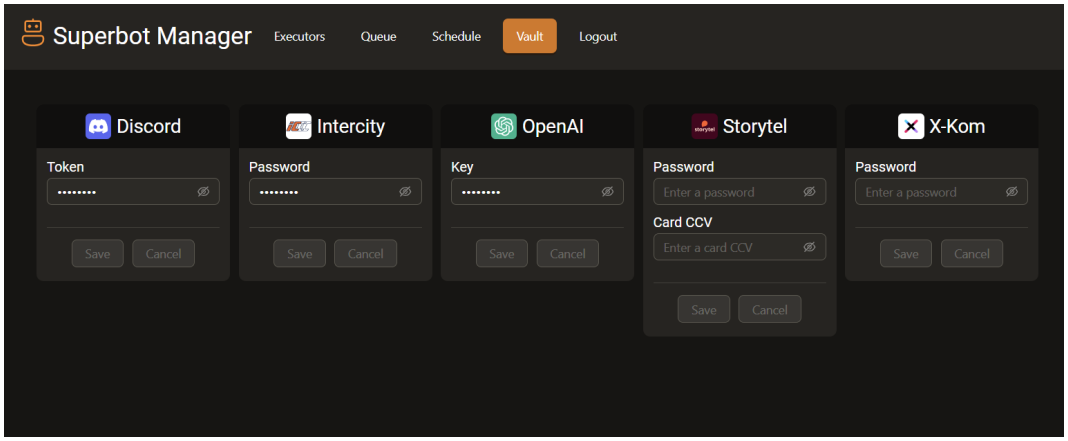
Musiałem, się jednak z tego sposobu wycofać, ponieważ rozwiązanie to okazało się zbyt mało elastyczne. Np. wyobraźmy sobie następującą sytuację:

1. Użytkownik edytuje ActionExecutor'a, gdzie są dwa Inputy, oba z różnymi wartościami w polu typu secret.
2. Użytkownik postanawia zduplikować jeden z inputów, klikając przycisk Duplicate.
3. Następnie zapisuje ActionExecutor'a i wychodzi.

Backend otrzymuje teraz ActionExecutor'a z nowym Inputem, gdzie w polu z secretem wpisane są znaki zapytania. Niestety nie ma prawa wiedzieć, z którego input'a powinna znaleźć się tam wartość. Z tego powodu konieczne było wysyłanie Id secretów zamiast maskowania.

Nie powoduje to jednak żadnych problemów z bezpieczeństwem, ponieważ nawet jeżeli ktoś uzyska dostęp do bazy danych i znając GUID secretu, będzie wiedział, gdzie leży hasło, którego szuka, nie będzie w stanie go odkodować. Wszystkie secrety szyfrowane są algorytmem AES z hasłem zapisanym w pliku konfiguracyjnym i zmiennym IV, więc bez dostępu do VPS'a nie będzie w stanie go odkodować.

Ten mechanizm zapewnia już stuprocentowe bezpieczeństwo, jednak można się tu jeszcze pokusić o poprawę komfortu użytkownika. Częstym przypadkiem może być, że użytkownik używa jednego klucza do danego serwisu, aby zrobić wiele ActionExecutor'ów. Musiałby wtedy przy definicji każdego ActionExecutor'a tę wartość uzupełniać. Można powiedzieć, że zrobi to raz na samym początku, a potem nie będzie musiał się tym martwić, jednak może być to niewygodne. W dodatku, na późniejszym etapie może wpaść na pomysł, by dołożyć nowy ActionExecutor z tym samym kluczem, lecz niestety nigdzie sobie poprzedniego klucza nie zapisał. Musiałby w takim przypadku wygenerować klucz ponownie i pozmieniać go w każdym ActionExecutorze.



Rysunek 2.7: Vault

Aby rozwiązać ten problem powstał mechanizm Vault. Pozwala on użytkownikowi wpisać klucz do serwisu, a system go zaszyfruje i będzie automatycznie temu użytkownikowi uzupełniał przy tworzeniu nowych Inputów w ActionExecutorze. Nie ogranicza to również w żaden sposób funkcjonalności, gdyż dalej możemy mieć wpisane dowolne klucze w executorze, tylko mamy tę zaletę, że domyślnie się on uzupełni i nie będziemy go musieli wpisywać za każdym razem. Zmiana klucza w Vaulcie nie zmienia klucza w ActionExecutor'ach ani akcjach.

2.4. Łańcuchowe wykonywanie akcji

Z początku miałem prostą ideę, aby były dwa osobne rodzaje obiektów – Akcje i Notyfikacje. Notyfikacje są bardzo ważnym elementem często potrzebnym do działania botów. Np. po uruchomieniu bota do kupna biletu w Intercity mamy tylko 15 min na opłacenie go. Czekanie aż zwolni się miejsce siedzące w pociągu możne trwać nawet kilka godzin, dlatego tak ważne jest, żebyśmy zauważyli to w porę, że bot wykonał zadanie.

Na etapie projektowania systemu zauważyłem jednak, że notyfikacje wcale nie różnią się tak bardzo od botów i można by spróbować dwa te obiekty ujednolicić do jednego wspólnego. Zamiast pozwalać uruchamiać na zakończenie bota tylko notyfikacji, można pozwalać uruchamiać dowolną akcję po dowolnej inne akcji, gdzie akcją może być zarówno bot, jak i notyfikacja. Dzięki takiemu podejściu system jest dużo bardziej uniwersalny i nie ogranicza się jedynie do botów, a może być wykorzystywany do dowolnych zadań, które można wygodnie ze sobą łączyć.

Skąd notyfikacja miałaby wiedzieć, co wyświetlić? Podobnie jak to jest w przypadku uruchamiania programów na komputerze, postanowiłem, że każda akcja mogłaby mieć jakieś wejście i wyjście. Wtedy można by łatwo przekazać wyjście jednej akcji do wejścia drugiej.

Przeanalizujmy przykładowy przepływ akcji, jaki można by zaprojektować przy użyciu tego podejścia. Załóżmy, że chcemy zrobić system, który połączy nam Discord'a z GPT od OpenAI. Chcemy, aby po uruchomieniu executora, system zapytał nas na Discordzie o ulubiony kolor, a następnie po otrzymaniu odpowiedzi od użytkownika, wypisał 10 zwierząt o podanym kolorze.

Możemy w prosty sposób zdefiniować powyższy przebieg w mniej niż minutę. Potrzebujemy dodać 3 ActionExecutor'y: Discord – Prompt, OpenAI – GPT oraz Discord – Send Message. W Message input'a pierwszej akcji piszemy prośbę o podanie koloru i w polu Run on finish wybieramy akcję drugą. Jak teraz w akcji drugiej można się odwołać do wyjścia akcji pierwszej? Na obrazku 2.1 widzimy, że dla każdego typu akcji mamy podane jakie przejmują wejścia, a także wyjścia. Wyjściem akcji Discord – Prompt jest pole Answer typu string. Chcąc użyć go w wejściu OpenAI – GPT możemy w polu Question wpisać „Podaj mi 10 zwierząt o kolorze {{{Answer}}}" dzięki takiej konwencji w miejsce {{{Answer}}} zostanie wstawiona wartość z outputa poprzedniej akcji. Analogicznie dla Discord – Send message możemy wpisać w polu Message „{{{Response}}}". W ten wszystko jest gotowe. Pozostaje tylko uruchomić Executor Discord – Prompt. Dla wygody korzystania warto sobie także pozmieniać nazwy executor'ów oraz zmienić na Automatic te których nie chcemy uruchamiać ręcznie (ukryje to przycisk Run, aby się nie mylić)

Co jeżeli chciałbym wyświetlić wynik nie tylko na Discordzie? Dodam sobie czwartą akcję i co wpiszę? Przecież trzecia akcja typu Discord – Send Message nic już nie zwraca? Aby rozwiązać ten problem, zastosowałem jeszcze jedno rozwiązanie. Akcje zwracają nie tylko swoje własne wyjście, ale także całe wejście, jakie dostały (z wyjątkiem secret'ów). Dzięki temu, dziesiąty executor z kolei może bez problemu wpisać do input'a wartość z np. pierwszego.

Rozdział 3.

Szczegóły techniczne

3.1. Baza danych

3.1.1. Obsługa bazy danych

Kod do obsługi bazy danych znajduje się w projekcie SuperBotManagerBase, w katalogu DB. Zastosowałem technologię Entity Framework Core w podejściu Code First. Zaimplementowałem wzorzec projektowy Repository oraz Unit of Work, dla większej modułowości kodu i w przyszłości dla wygodniejszego pisania testów jednostkowych mockujących bazę danych.

Zrobiłem interfejs IEntity, dodaje on do każdego obiektu z bazy CreatedDate oraz ModifiedDate. Napisałem także kod w AppDbContext.cs, który automatycznie aktualizuje te daty przy modyfikacjach rekordów z bazy. Uważam to za bardzo wygodną sztuczkę.

3.1.2. Migracje

Entity Framework zapewnia gotowy mechanizm migracji do aktualizowania struktury bazy danych. Aby zainicjować psutą bazę danych, należy wykonać następujące kroki:

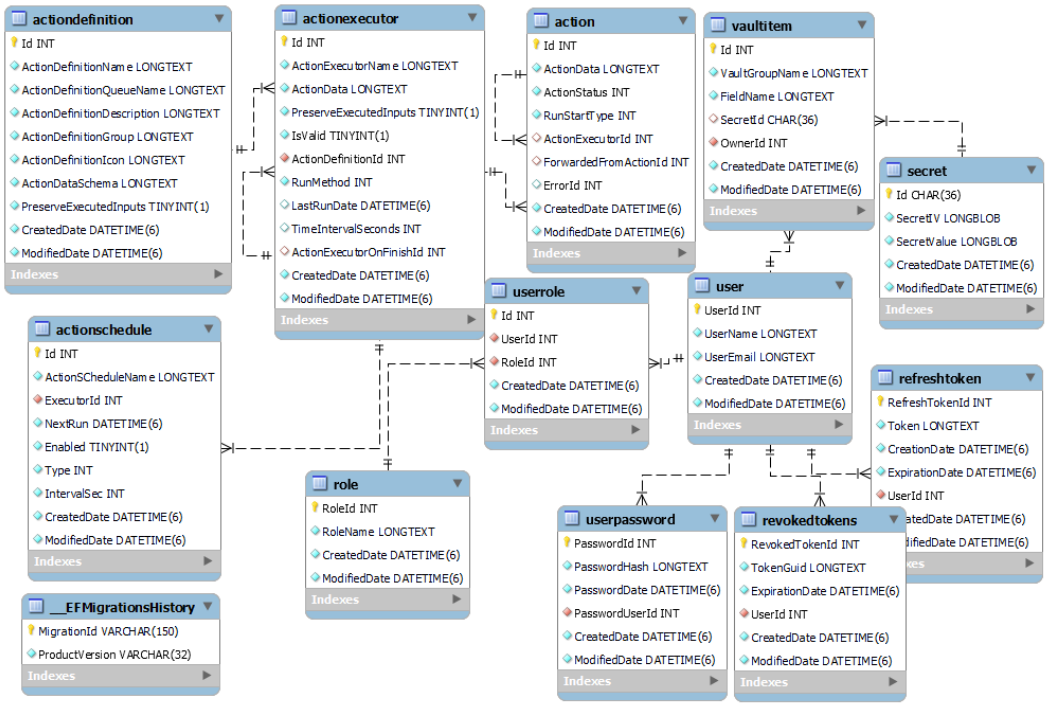
1. Ustawić prawidłowy ConnectionString do bazy w pliku appsettings.json w projekcie SuperBotManagerBackend.

2. Otworzyć Package Manager Console

3. Wybrać jako domyślny projekt SuperBotManagerBackend

4. Uruchomić Update-Database

3.1.3. Diagram tabel



Rysunek 3.1: Schemat bazy danych

3.1.4. Bug w PostgreSQL?

Przez większość czasu rozwoju projektu używałem bazy Postgres. Wszystko działało prawidłowo do momentu, aż nie dodałem Harmonogramu zadań. Od tego momentu zaczął pojawiać się dziwny błąd, z połączeniem do bazy danych. Część zapytań do bazy danych wykonywała się zwyczajnie, a na część baza zwracała błąd połączenia. Nie było na to żadnej reguły. W losowych momentach i modułach aplikacji raz połączenie nawiązywało się poprawnie, a raz nie.

Baza danych wyrzucała błędy połączeniom z innych kontenerów Docker'owych na VPS'ie. W przypadku gdy uruchamiałem backend lokalnie, łącząc się z tą samą bazą danych na VPS'ie, to błąd się nie pojawiał. Połączenia lokalne otrzymywały błąd, a zdalne działały. Z początku myślałem, że to ja zrobiłem jakiś błąd.

Co ciekawe, na tym serwerze bazy danych miałem kilka innych baz danych, do innych aplikacji, co do których miałem 100% pewności, że działają dobrze. Bardzo się zdziwiłem, że nagle zaczął pojawiać się w nich identyczny problem. Miałem wtedy pewność, że błąd nie był w moim kodzie, a jest to problemem z serwerem bazy danych. Usunąłem kontener serwera bazy danych i uruchomiłem jego najnowszą wersję obrazu od zera. Wydawało się, że problem ustąpił – przez kilka godzin wszystko działało prawidłowo. Jednak przez noc znowu się popsuło i następnego

dnia, był już identyczny problem jak wcześniej. W ostateczności postanowiłem przetestować, czy z MySQL będzie ten sam problem. Okazało się, że po zmianie bazy danych na MySQL wszystko działa idealnie i baza danych nie zwraca błędów.

Co mogło być przyczyną błędu? Zauważmy, że błąd zaczął występować po dodaniu ScheduleService. Jedyne co robił ten serwis, to odpytywał co 10 sekund bazę danych, pobierając wszystkie rekordy z tabeli actionschedule, których czas kolejnego uruchomienia był wcześniejszy niż obecny czas oraz które miały ustawione parametr Enabled na true. Wszystkich rekordów nigdy nie było więcej niż 3, a warunek spełniało prawie zawsze 0. Wydawałoby się, że nie może to powodować zepsucia serwera bazy danych, ponieważ nie jest to nic wyrafinowanego. Dodatkowo serwer bazy danych odrzucał nie wszystkie połączenia, a tylko te Docker'owe i też nie na każde, a losowo niektóre, niezależnie czy dotyczyło logowania, czy czegoś z dowolnej innej tabeli. Zauważyłem także, że serwer bazy danych zaczynał wyrzucać błędy do zupełnie innych, niezależnych baz. Jest to, więc bardzo dziwny i trudny to odtworzenia błąd. Może dlatego nie został jeszcze naprawiony.

Szukałem tego błędu w internecie i znalazłem taki wątek:
https://github.com/npgsql/npgsql/issues/3955
Wygląda na to, że błąd ten występuje już od kilku lat.

3.2. Frontend

3.2.1. Użyte technologie

Frontend pisałem przy użyciu Typescript'a oraz biblioteki React. Zastosowałem tam następujące biblioteki i narzędzia:

- pnpm [3] – Manager paczek. Jest to wydajniejsza alternatywa dla npm.
- vite – Alternatywa dla Create React App. Jest to lokalny, bardzo wydajny serwer React'a, z wieloma przydatnymi funkcjami.
- eslint – Narzędzie do sprawdzania kodu w poszukiwaniu potencjalnych błędów i złych praktyk
- prettier – Narzędzie do jednolitego, automatycznego formatowania kodu
- lint-staged – Narzędzie umożliwiające wykonanie dowolnych programów, podając im jako argumenty pliki które są w stanie staged w repozytorium git
- husky – narzędzie dodające hook precommit do repozytorium gita. Skonfigurowałem go, aby przy każdym commit'cie były automatycznie sprawdzane wszystkie pliki. Jeżeli jest jakiś błąd, to o tym poinformuje i nie pozwoli zrobić commit'a.

- zustand – Bardzo wygodny state manager. Lepsza alternatywa dla Reduxa.
- antd – Ant design. Jest to biblioteka z komponentami do React'a.
- @tanstack/react-query [4] – świetna biblioteka do korzystania z REST API. Ma wiele świetnych udogodnień np. cachowanie zapytań.
- axios – biblioteka do wygodnego wysyłania zapytań http
- immer, use-immer – są to biblioteki umożliwiające wygodniejszą aktualizację stanów, bez konieczności pilnowania by zmiana była immutable
- styled-components [6] – biblioteka do wygodnego stylowania komponentów
- polished – zawiera funkcje do modyfikacji kolorów css'owych
- react-icons – biblioteka z ikonami
- dayjs – biblioteka do wykonywania operacji na datach oraz ich formatowania
- framer-motion – biblioteka do tworzenia animacji
- i18next, react-i18next [5] – biblioteka do zarządzania tłumaczeniami
- react-router-dom [7]- biblioteka do zarządzania routingiem w aplikacji React'owej
- react-toastify – biblioteka do wyświetlania komunikatów (Użyta do wyświetlania ewentualnych komunikatów o błędzie)
- react-hook-form – biblioteka do walidacji formularzy (użyte przy rejestracji)
- usehooks-ts – biblioteka z wieloma przydatnymi hook'ami do React'a

3.2.2. React Query [4]

Bardzo użyteczną biblioteką w React, która od niedawna dopiero zyskuje coraz większą popularność, jest react-query. Dodaje ona do aplikacji frontend'owej możliwość cachowania odpowiedzi na zapytania HTTP. Jest to szczególnie użyteczne, gdy w backendzie stosowane jest REST API. Załóżmy, że kilka różnych komponentów React'owych potrzebuje informacje o jakimś obiekcie z backend'u. Normalnie użylibyśmy funkcji fetch lub Axios'a wewnątrz useEffect. Powodowałoby to, że niepotrzebnie każdy komponent wysłałby osobne zapytanie do serwera o to samo. Dodatkowo może zdarzyć się, że zmodyfikujemy obiekt backend'owy. W takim przypadku należałoby odświeżyć go w każdym useEffectcie, aby na nowo został wysłany request. React Query rozwiązuje oba te problemy i wiele więcej.

Wprowadza on hook'a useQuery, który zwraca m.in. otrzymane dane, informację czy w tym momencie trwa ich pobieranie oraz obiekt z możliwym błędem. Dzięki

temu łatwiej wyświetlić spinner z animacją, że trwa ładowanie danych. Jest tam także hook useMutation, który ma służyć do wysyłania requestów powodujących zmianę obiektu np. put lub delete. Po wysłaniu takiego requestu, zazwyczaj chcielibyśmy oznaczyć obecny stan cache'a jako już nieaktualny i poinformować o react-query o konieczności jego odświeżenia. Do wykonania tej operacji, wprowadzony został osobny mechanizm unieważniania zapytania. Wystarczy uruchomić funkcję invalidateQuery, podając jako argument klucz, którym identyfikujemy nasze zapytanie.

3.2.3. i18next [5]

Aplikację przygotowałem z tłumaczeniami polskimi i angielskimi. Znalazło się w niej po 95 tłumaczeń do obu języków. Aby zmienić język, można przełączyć się przyciskiem w prawym górnym rogu strony internetowej.

Pliki .json zawierające tłumaczenia znajdują się w katalogu: SuperBotManager-Frontened/locale.

Do skonfigurowania mechanizmu tłumaczeń użyłem bibliotek i18next oraz react-i18next. Zawierają one wiele przydatnych funkcji ułatwiających tworzenie tłumaczeń jak np. interpolacja stringów wewnątrz tekstu tłumaczenia. Druga z tych bibliotek wprowadza hook useTranslation, który powoduje rerender komponentu, jeżeli język zostanie zmieniony. Dzięki temu można zmienić tłumaczenia na całej stronie, bez konieczności jej przeładowywania.

3.2.4. styled-components [6]

Jedną z najlepszych bibliotek w React do stylowania interfejsów użytkownika jest styled-components. Nie pisze się w niej tak szybko jak w Tailwind, jednak powstały kod jest dużo bardziej czytelny i łatwiej go w przyszłości zmieniać. W bibliotece tej, można zdefiniować sobie własny motyw kolorystyczny, a następnie używać go w dowolnych komponentach. Motyw kolorystyczny możemy w każdym momencie zmienić, aby komponenty wykonały rerender i zmieniły swoje style css. Wprowadziłem w prawej górnej części strony internetowej przełącznik do zmiany między dark mode'm, a light mode'm.

3.3. RabbitMQ

[1] Przed tym projektem nie miałem, żadnego wcześniejszego doświadczenia z RabbitMQ. Szukając dobrych praktyk i wzorców, jak najlepiej z niego korzystać, natknąłem się na artykuł [20]. Kod tego autora do konfiguracji RabbitMQ wydał mi się bardzo profesjonalny i postanowiłem użyć go w mojej aplikacji.

Lekko go tylko dostosowałem, aby można było ten sam typ obiektu przesyłać do różnych kolejek, z dowolną nazwą kolejki.

Podejście, które zastosował autor, ma kilka zalet:

1. Jeżeli wystąpi błąd podczas wykonywania akcji, akcja powodująca błąd nie trafi z powrotem do tej samej kolejki, gdyż mogłoby to spowodować nieskończoną pętlę błędów. Serwis w nieskończoność pobierałby akcję, robił błąd i ją zwracał. W zamian za to, akcja trafia do osobnej kolejki z sufiksem -errors w nazwie.

2. Jeżeli wystąpi błąd podczas wykonywania akcji, wszystkie wykonane operacje z użyciem RabbitMQ przez ten serwis zostają cofnięte. Jest to możliwe dzięki zastosowaniu mechanizmu transakcji RabbitMQ. Obecnie tego mechanizmu nie potrzebuję, ale chciałbym dodać zwracanie wyniku akcji z powrotem do RabbitMQ, zamiast bezpośrednio do bazy, będzie to wtedy przydatne.

3. Wszystko jest oparte o Dependency Injection, dzięki czemu kod jest modułowy i łatwy do mock'owania na potrzeby testów.

3.4. SuperbotConsumerService

3.4.1. O projekcie

Jest to projekt, który ma za zadanie uruchamiać w tle serwisy. W momencie uruchomienia go, ładowane są wszystkie pliki z katalogu programu, kończące się na Consumer.dll oraz BackgroundService.dll. Następnie z użyciem mechanizmu refleksji znajdowane są wszystkie klasy, będące konsumentami RabbitMQ (oznaczone atrybutem ServiceActionConsumerAttribute) oraz klasy, które zawierają funkcję, która ma działać w tle (Oznaczone atrybutem SuperbotScheduleBackgroundService). Taka możliwość przyda nam się np. dla projektu SuperbotScheduleBackgroundService, który nie jest konsumentem RabbitMQ.

Dla każdego typu klasy tworzona jest nowa jej instancja i osobny obiekt IHost, co zapewnia większą separację między nimi. Projekt ten dodaje każdemu z IHostów w Dependency Injection dostęp do: bazy danych, RabbitMQ, modułu szyfrującego oraz modułu umożliwiającego uruchomienie Selenium Web Drivera [9].

Projekt ten posiada własny appsettings.json, niezależny od tego backend'owego. Jest on w trochę innej postaci. Np. w tym projekcie nie ma sensu konfigurować CORS'a, ale jest np. sens konfigurować Selenium.

3.4.2. Selenium Grid

Wiele botów do działania potrzebuje Selenium, aby móc sterować przeglądarką. Zastanówmy się, w jaki sposób najlepiej jest go skonfigurować w naszym przypadku. Docelowo chcemy korzystać z Selenium na VPS'ie. Do poprawnego działania, Selenium potrzebuje zainstalowanej przeglądarki w wersji takiej samej jak sterownik dodany w projekcie. Nie zapominajmy także, o tym, ze SuperbotConsumerService będzie pracować w kontenerze Docker'owym.

Pierwszym pomysłem może być zainstalowanie przeglądarki internetowej w kontenerze, w którym znajdować się będzie nasz SuperbotConsumerService. To rozwiązanie jest możliwe do realizacji, jednak byłoby bardzo niewygodne. Debugowanie błędów w bocie w takim kontenerze byłoby praktycznie niemożliwe. Dodatkowo nie skalowałoby się to dobrze.

Okazuje się, że Selenium posiada już gotowy mechanizm na rozwiązanie tego problemu. Pozwala on uruchomić osobny mikroserwis o nazwie Selenium Grid [10], który będzie zarządzał jedną lub wieloma przeglądarkami. Możemy uruchomić oddzielny kontener Selenium Grida z od razu zainstalowaną przeglądarką. Obraz Docker'owy znajduje się tutaj: https://hub.docker.com/r/selenium/standalone-chrome

Dzięki temu podejściu możemy mieć bota wykorzystującego Selenium na jednym komputerze, a kontener z przeglądarką kontrolowaną przez tego bota na zupełnie innym. Na dodatek ten obraz Docker'owy daje możliwość podejrzenia, co się dzieje w przeglądarce wewnątrz kontenera. Na porcie 7900 uruchamia się serwer strony internetowej na której możemy zobaczyć naszą przeglądarkę.

Aby powyższe technologie były konfigurowalne, zdefiniowałem w appsettings.json sekcję Selenium. Oto przykładowa konfiguracja:

```
"Selenium": {
    "Local": false,
    "Headless": true,
    "Hostname": "20.119.50.77",
    "Port": "4444",
    "Proxy": null
}
```

- Jeżeli **Local** jest true to Hostname i Port będą ignorowane, a Selenium uruchomi przeglądarkę w sposób tradycyjny, znajdując jej lokalizację na lokalnym komputerze. Ustawienie to może być przydatne na czas programowania. Na VPS wartość ta zazwyczaj będzie równa false.

- **Headless** pozwala nam uruchomić przeglądarkę, nie otwierając jej okna. Warto włączyć tę opcję, gdy już mamy pewność, że bot działa prawidłowo i nie trzeba go będzie debugować.

- **Hostname** określa nam adres serwera, na którym znajduje się Selenium Grid
- **Port** określa nam port serwera, na którym znajduje się Selenium Grid
- Mamy także możliwość ustawić **Proxy** dla przeglądarki

3.5. Consumers

Cały system jest dosyć wymagający, dlatego wolałem się skupić na jego rozwoju zamiast na dodawaniu przykładowych botów. Teraz, gdy już system w pełni działa, dodawanie do niego botów jest rzeczą prostą. W ramach pracy przygotowałem tylko kilka przykładowych możliwych akcji, aby zademonstrować jego działanie i możliwości.

3.5.1. Discord – Send Message + Discord – Prompt

Obie akcje reprezentują dosyć podobne działanie, dlatego postanowiłem je definiować i implementować w tych samych projektach. Jest to też dobry przykład, aby zademonstrować, że wiele różnych definicji, jak i wiele różnych konsumentów może znajdować się w jednym pliku dll. Jest to dobre podejście, jeżeli mamy pewność, że nie będziemy ich nigdy chcieli rozdzielać.

Obie te akcje nie są botami, a odpowiadają bardziej za powiadomienia dla użytkownika oraz otrzymywanie od niego informacji, poprzez wygodny komunikator. Aby móc z nich korzystać należy pierwsze wygenerować sobie token podłączenia do Discord'a. Można to zrobić według tego artykułu: https://www.writebots.com/discord-bot-token/. Mógłbym wprowadzić predefiniowane boty do Discord'a z prostym przyciskiem, aby je dodać serwerów Discordowych, ale uznałem, że nie warto robić i lepiej dać pełną swobodę użytkownikowi. Dzięki temu może sam sobie zdefiniować, jak Bot będzie wyglądał, jakie będzie miał uprawnienia itp.

Discord – Send Message służy do wysłania wiadomości. W momencie uruchomienia konsumenta znajduje on wszystkie serwery Discord'a, do których bot został dodany. Następnie wysyła na każdym serwerze na dowolny kanał zdefiniowaną wcześniej wiadomość. Dodałem także opcję w Executorze o nazwie Tag Everyone. Jeżeli jest ona włączona, to na początku wiadomości bot dopisze @everyone, dzięki czemu mamy większa szansę, że zobaczymy powiadomienie. Aby było to możliwe, musimy nadać botowi stosowne uprawnienie, w panelu API na stronie Discord'a.

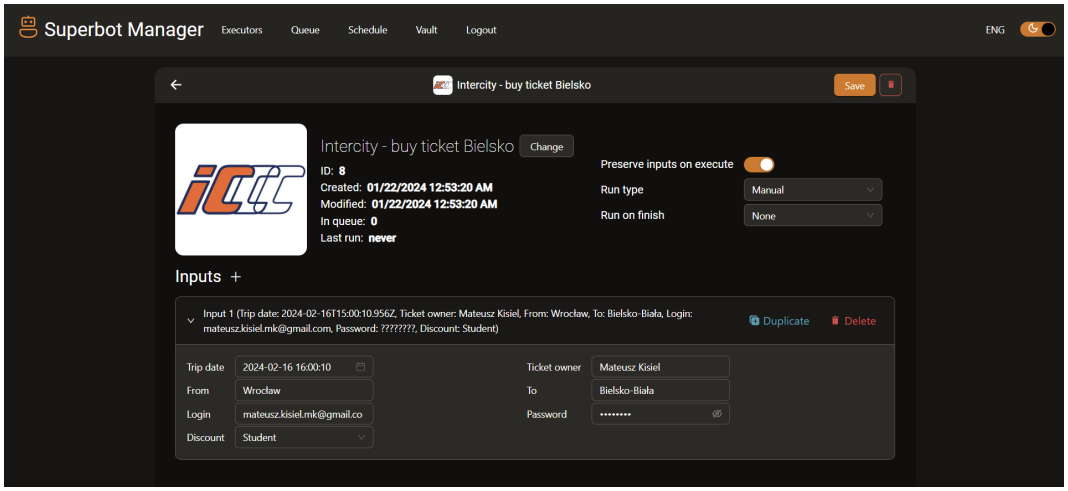
Discord – Prompt Analogicznie także wysyła wiadomość, a następnie oczekuje, aż użytkownik napisze odpowiedź do podanej wiadomości. Dodatkowo jest opcja spam, która przy ustawieniu na true, będzie wysyłać wiadomość początkową co kilka sekund, aby użytkownik na pewno zauważył pytanie. Ten typ akcji może przydać się w wielu zadaniach, gdzie wymagane jest potwierdzenie od użytkownika

np. przez aplikacje Google Authenticator. Wtedy bota można by podzielić na trzy części, etap do momentu potwierdzenia, Discord – Prompt, a następnie dalsza część.

Do komunikacji z API Discord'a wykorzystałem bibliotekę nuget Discord.Net. [21]

3.5.2. Intercity – buy ticket

Zapewne każdemu zdarzyła się kiedyś sytuacja, gdy chciał kupić bilet na pociąg, a okazało się, że już nie ma miejsc. Intercity bot jest rozwiązaniem na ten problem. Wystarczy zdefiniować, kiedy chcemy odbyć podróż, skąd, dokąd, kto ma być właścicielem biletu, login i hasło do konta oraz możliwa zniżka. Obrazek 3.2



Rysunek 3.2: Edycja Executora Intercity

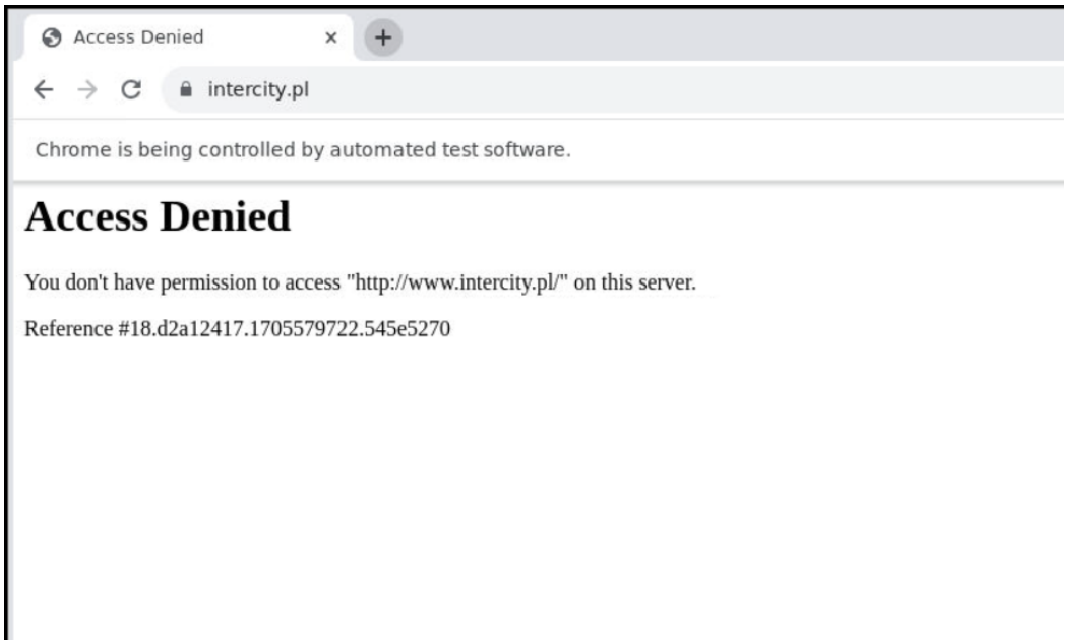
Bot otworzy stronę Intercity (z użyciem Selenium), wprowadzi dane o naszej podróży i o możliwych ulgach, a następnie wyszuka bilet. Jeżeli okaże się, że nie ma miejsca siedzącego, bot wyszuka ponownie, sprawdzając czy może ktoś zwrócił swój bilet i miejsce się zwolniło. Będzie tak długo powtarzał te kroki aż uda mu się znaleźć wolne miejsce. Następnie wprowadzi dane do logowania do Intercity i kliknie przycisk „zapłać później". Od tego momentu użytkownik ma 15 minut, aby opłacić swój bilet na stronie Intercity. Dlatego ważne jest, aby użytkownik zauważył powiadomienie, bo jeżeli nie zapłaci w określonym czasie, bilet przepadnie.

Z doświadczenie wiem, że bot ten sprawdza się bardzo dobrze i wiele razy znalazł mi miejsce siedzące. Z reguły najwięcej ludzi rezygnuje dzień przed i w dzień odjazdu. Bot był testowany do kupna biletu na trasie Wrocław – Bielsko-Biała. Nie wiadomo, czy na innych trasach będzie działał równie poprawnie. Można dodać w nim obsługę także przesiadek oraz wybierania ręcznego miejsca z obrazka. Obecnie te dwie opcje nie są w bocie wspierane.

Nie musimy podawać w bocie dokładnej godziny odjazdu i nazw stacji. Bot wpi-

sze tę nazwę stacji, którą mu podamy, a następnie wybierze pierwszą podpowiedź od strony Intercity. Analogicznie z czasem odjazdu – zostanie kupiony bilet na pociąg, który domyślnie zaproponuje Intercity dla podanego czasu. Jeżeli jeden pociąg jest o 15:00, drugi o 16:00 wystarczy, że wybierzemy dowolną godzinę miedzy 15:00, a 16:00, żeby bot kupił bilet na ten o 16:00.

W momencie jak wgrałem bota na VPS'a, okazało się, że z Intercity jest jeden problem, który nie występował, jak uruchamiałem go lokalnie. Okazało się, że Intercity blokuje adresy IP serwerów VPS Azura. Próbowałem na dwóch VPS'ach – jeden z USA drugi z UK. Oba były blokowane, obrazek 3.3.



Rysunek 3.3: Intercity blokada

Miałem kilka pomysłów jak obejść ten problem:

- Zastosować proxy
- Zmienić dostawcę VPS
- Uruchomić na innym komputerze Selenium Grid'a
- Uruchomić na innym komputerze Selenium Grid'a oraz konsumenta Intercity

Nie udało mi się znaleźć żadnego dobrego darmowego proxy, które by działało w dłuższym przedziale czasu. Dostawcy VPS'a nie chciałbym zmieniać, bo mam jeszcze do wykorzystania darmowe środki dla studentów na Azure. Zdecydowałem się na opcję czwartą, czyli przenieść konsumenta i Selenium Grida na mój laptop. Na laptopie nie mam publicznego IP, dlatego opcja trzecia nie wchodziła w grę, bo VPS nie mógłby się do mnie połączyć. W opcji czwartej tylko mój laptop się łączy do VPS'a, więc tego problemu nie ma. Można to zobaczyć na obrazku 1.1

3.5.3. OpenAI – GPT API

Postanowiłem także dodać akcję, która pozwoli wykorzystać OpenAI API. Daje ona ogromne możliwości, zostawiając użytkownikowi masę swobody. GPT3.5 jest bardzo tanie, jest to kwestia dosłownie groszy, działa prawie za darmo. Można to wykorzystać np. do generowania maili na jakiś temat i automatycznego wysyłania ich (trzeba by tylko dodać akcję typu wysłanie maila). Dodatkowo idealnie nadaje się to także przykładu tłumaczącego działanie łańcuchowego uruchamiania akcji.

Wgenerować klucz API można tutaj: https://platform.openai.com/api-keys

3.5.4. Storytel – Sign up

Nowe konta na Storytel posiadają 7 dni okresu próbnego, dzięki czemu można przez ten czas za darmo korzystać z ich aplikacji. Nie ma problemu, by zrobić wiele kont ręcznie, w dodatku jest to dosyć proste, dlatego myślałem, że warto byłoby sprawdzić, jak bot poradziłby sobie z tym zadaniem.

Z początku chciałem to zrobić za pomocą czystych HTTP Requestów. Okazało się, jednak że to podejście jest praktycznie niemożliwe, bo serwer zwraca kod JavaScript'owy, który trzeba wykonać, aby utworzyły się odpowiednie klucze i wygenerowały ciasteczka.

Następnie napisałem bota w Selenium, który robi dokładnie to samo co użytkownik – uzupełnia formularz i klika przycisk zarejestruj. Okazało się, jednak że Storytel nie pozwalał utworzyć konta. Gdy przyjrzałem się wysyłanym zapytaniom http. Zauważyłem, że serwer Cloudflare zwrócił błąd, uznając mnie za bota. Po doczytaniu dowiedziałem się, że serwis ten jest używany do zabezpieczeń przeciw botom i w moim przypadku wykrył, że przeglądarka jest kontrolowana przez Selenium. Nie jest ciężko wykryć, że przeglądarka jest kontrolowana przez Selenium, np. domyślnie dodaje on zmienną zaczynającą się od $cdc_ w kodzie JavaScript przeglądarki, a jest to jeden z wielu problemów.

Aby to obejść, można spróbować użyć biblioteki undetected-chromedriver. Działa ona w taki sposób, że przed uruchomieniem przeglądarki patchuje ją sprawiając, że jest niewykrywalna. Niestety obecnie metoda ta możliwa jest do uruchomienia tylko lokalnie, ponieważ należy podać ścieżkę do chromedriver'a.

Uruchomiłem bota z użyciem tej biblioteki. Niestety okazało się, że Cloudflare na stronie Storytel, dalej w jakiś sposób wykrywa, że przeglądarka jest sterowana i nie pozwala zrobić konta. Myślałem, że być może, wykrycie następuje poprzez zbyt szybkie uzupełnienie formularza, jednak gdy robię to ręcznie w tej przeglądarce, system także i tego nie przepuszcza.

Obecnie nie miałem czasu dalej kontynuować obchodzenia tego problemu. Istnieje jednak więcej metod na rozwiązanie tego problemu. Można próbować analizo-

wać JavaScript strony internetowej, aby podszyć się pod Cloudflare przekazując informację, że jesteśmy poprawnie sprawdzenie. W ostateczności, jeżeli żadna z metod nie zadziała, można w pełni zasymulować interakcje użytkownika otwierając prawdziwą przeglądarkę i poruszać myszką wykorzystując API systemu operacyjnego. Ta metoda jest niemożliwa do wykrycia, jednak jest także bardzo mało wydajna.

Więcej o możliwych metodach obejścia Cloudflare można poczytać tutaj: https://www.zenrows.com/blog/bypass-cloudflarehow-cloudflare-detects-bots

3.5.5. X-KOM – open boxes

Aplikacja X-KOM na telefon posiada możliwość raz na 24h wylosować trzy promocje, otwierając skrzynki. Tutaj bot także idealnie mógłby się nadać. Można by wykorzystać Harmonogram zadań, aby skrzynki były automatycznie codziennie otwierane, być może na wielu kontach.

Aby podejrzeć jakie zapytania robi telefon z Androidem do serwera X-KOM, musiałem obejść kilka problemów.

- Można pomyśleć, że wystarczy ustawić w telefonie serwer proxy, aby wszystkie requesty przechodziły przez Burp'a lub Fidder'a. Z tym podejściem pojawia się jednak pewien problem: Obecnie wszystkie aplikacje używają szyfrowania SSL, przez co pośredni serwer nie jest w stanie w ten zobaczyć tych requestów.
- Burp ma opcję, podobnie jak w ataku Man-in-the-middle utworzyć połączenie szyfrowane zarówno z serwerem, jak i z klientem. Dzięki czemu jest w stanie podejrzeć treść wszystkich zapytań. Pojawia się za to wtedy inny problem – aplikacja w telefonie zauważa, że nie zgadza się certyfikat SSL i blokuje takie połączenie. Zabezpieczenie to nazywa się SSL Pinning [19].
- Idąc dalej, można spróbować wyeksportować certyfikat z Burp'a i wgrać go na telefon. Niestety tu pojawia się kolejny problem – od Androida 7 (Nougat) wzwyż, nie da się wgrać ręcznie certyfikatu bez root'owania urządzenie. A znowu root'owanie powoduje zablokowanie część aplikacji np. aplikacji banków.
- Użyłem zatem emulatora Androida. Aby wgrać własny certyfikat do wybranej aplikacji, należałoby ją zdekompilować, podmienić plik z certyfikatem, a następnie skompilować ponownie. Nie jest to jednak zbyt wygodne i może rodzić problemy, dlatego poszukałem i udało mi się znaleźć prostszy sposób.
- Zastosowałem narzędzie Frida, które pozwala obejść SSL Pinning, umożliwiając wstrzyknięcie dowolnego kodu do uruchomionej aplikacji, w tym takiego, który usuwanie zabezpieczenie do sprawdzania certyfikatu SSL.

W ten sposób uzyskałem informacje, jakie dokładnie requesty wysyła aplikacja X-KOM na telefonie. Aby otworzyć skrzynkę, wystarczy wykonać 2 requesty:

1. POST https://mobileapi.X-KOM.pl/api/v1/xkom/Token – podając w headerze X-Api-Key równe pewnej wartości (podejrzewam, że jest przypisana do instancji aplikacji), oraz w body typu x-www-form-urlencoded wartość username i password. W odpowiedzi otrzymujemy token.

2. POST https://mobileapi.X-KOM.pl/api/v1/xkom/Box/{skrzynka}/Roll – podając dodatkowo w header Authenticate: Bearer {Token}, gdzie {Token} to wartość otrzymana w poprzednim requeście, a {skrzynka} to liczba 1, 2 lub 3 w zależności od tego, którą skrzynkę chcemy otworzyć.

Napisałem prostego bota, który otwiera te skrzynki dla podanego maila, hasła oraz X-API-KEY. Niestety zauważyłem, że dosyć często X-API-KEY jest blokowane, a sam nie jestem w stanie takiego automatycznie wygenerować. Nie wiem w jaki sposób otrzymuje go aplikacja na telefon. Bot ten nie może być zatem użyty na masową skalę, ale jest dobrym przykładem jak można robić podobne do innych aplikacji na telefon.

3.5.6. Dodawanie nowych botów

Aby dodać nowego bota, mamy kilka możliwości:

1. Rekomendowana – Analogicznie jak w przypadku już istniejących zrobić dwa projekty. Jeden z definicją akcji i osobny z jej konsumentem. Do obu dodać referencję z projektu SuperBotManagerBase. Następnie skompilować oba i .dll konsumenta dodać do katalogu z SuperbotConsumerService, a .dll definicji do katalogu backend'u. Następnie zrestartować backend (spowoduje to automatyczne dodanie definicji do bazy). Alternatywnie zamiast dodawać pliki .dll ręcznie można dodać referncję do ich projektów w SuperbotConsumerService i w SuperBotManagerBackend.

2. Alternatywna – Zrobić to samo, z tą różnicą, że zarówno definicję, jak i konsumenta trzymać w jednym projekcie. Pierwszy sposób jest ciut lepszy, ponieważ nie trzeba dodawać wtedy do backend'u niepotrzebnie zależności używanych do wykonywania akcji.

3. Gorsza – Zrobić tylko konsumenta akcji, a wpis z definicją akcji dodać ręcznie do bazy danych. Jest to gorsze rozwiązanie, ponieważ istnieje ryzyko pomyłki. W dodatku, jeżeli baza padnie, możemy sobie nie przypomnieć, jak ta definicja była zbudowana.

Warto wspomnieć, że pisząc konsumenta akcji, nie musimy się przejmować obsługą RabbitMQ, ponieważ cały szablon jej obsługi już zrobiłem. Wystarczy tylko, konsument dziedziczył po klasie ActionQueueConsumer i miał atrybut [ServiceActionConsumer("")] z nazwą kolejki w parametrze.

3.6. Backend

Projekt backend'u został napisany z użyciem ASP.NET Core. Importuje on projekt SuperBotManagerBase, który zawiera wszystkie pomocnicze klasy i modele potrzebne do łączenia z bazą danych.

W momencie uruchomienia projektu wyszukiwane są w jego katalogu wszystkie pliki kończące się na ActionsDefinitions.dll i importowane jako assembly. Następnie wyszukiwane są w nich klasy zawierające atrybut ActionsDefinitionProviderAttribute. Jeżeli klasa taka zostanie znaleziona, to program spróbuje zaktualizować definicję akcji w bazie danych, lub utworzy nową, jeżeli taka jeszcze nie istnieje.

Logowanie użytkowników jest zaimplementowane z wykorzystaniem tokenów JWT [18]. Gdy użytkownik poprawnie się zaloguje, wysyłane są mu:

- Access token – Domyślnie ważny 10 min. Czas ten jest konfigurowalny w pliku appsettings.json w sekcji JWT.
- Refresh token – token potrzebny do wygenerowania nowego Access tokena. Jednocześnie przy takiej prośbie generowany jest także nowy Refresh token, a poprzedni unieważniany.
- TokenExpiration – data wygaśnięcia access tokena (UTC)
- RefreshTokenExpiration – data wygaśnięcia refresh tokena (UTC)
- User – obiekt zawierający informacje o zalogowanym użytkowniku

Dodatkową technologią użytą w projekcie backend'u jest biblioteka Automapper [8]. Udostępnia ona prosty sposób konwersji obiektu z jednego typu na drugi. Wykorzystuję ją do konwertowania obiektów bazy danych na ich okrojone wersje DTO(Data Transfert Object), które mają trafić do przeglądarki użytkownika.

Do logowania błędów zastosowana została biblioteka Serilog. Można ją skonfigurować w pliku appsettings.json, bez konieczności modyfikacji kodu C# i ponownego kompilowania.

Rozdział 4.

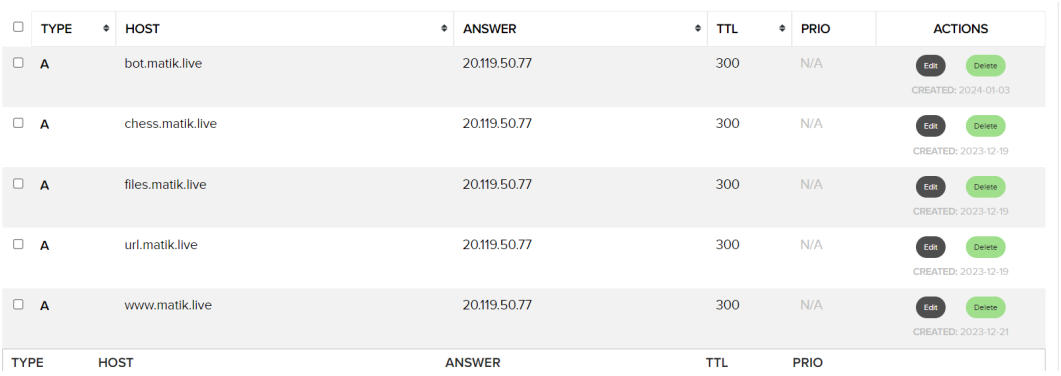
Deployment

4.1. VPS Manager

W pracy tej postanowiłem także rozwiązać problem wdrażania aplikacji na nowe serwery. W projekcie https://github.com/matik001/VPSManager znajdują się skrypty w Ansible [11] konfigurujące nowego VPS'a, a także tworzące odpowiednie kontenery Docker'owe, potrzebne do działania na nim aplikacji.

Aby skonfigurować nowy serwer VPS, należy:

- Wygenerować nową parę kluczy do ssh.
- Skopiować klucz publiczny do folderu ssh_keys
- Ustawić w inventory.yaml adres serwera oraz adres posiadanej domeny i swój adres email. Dodatkowo ustawić ansible_ssh_user na aktualną nazwę użytkownika wymaganą do połączenia do VPS'a. vps_username na nazwę nowego użytkownika
- W katalogu playbooks zmienić nazwę secrets.example.yaml na secrets.yaml i opcjonalnie ustawić własne hasła do baz danych
- Wykonać ansible-playbook playbooks/configure_vps.yml Zadanie zaktualizuje paczki, stworzy nowego użytkownika, zablokuje logowanie hasłem, wgra klucze ssh, z folderu ssh_keys i zainstaluje dockera.
- Następnie w inventory.yaml zamienić wartość ansible_ssh_user na nazwę nowo utworzonego użytkownika.
- Wykonać jeszcze raz profilaktycznie ansible-playbook playbooks/configure_vps.yml, aby wszystko na pewno było zainstalowane, dla nowego użytkownika.
- Wykonać ansible-playbook playbooks/update_containers.yml



Rysunek 4.1: Przykładowa konfiguracja dns

Ostatnie polecenie utworzy kilka kontenerów Docker'owych:

- Nginx [14] – Automatycznie utworzy się kontener z serwerem Nginx, który będzie działał jako reverse-proxy, oraz będzie hostować statyczne pliki projektów frontend'owych. Co więcej, udało mi się skonfigurować automatyczne podpisywanie certyfikatów SSL przez darmowy serwis Let's Encrypt, dzięki czemu jeżeli w DNS ustawione są rekordy A wskazujące na IP VPS'a (jak na obrazku 4.1), to certyfikaty zostaną automatycznie podpisane i ruch sieciowy będzie szyfrowany przez https.

  Dzięki zastosowaniu reverse proxy [13] przeglądarka pod tym samym adresem URL widzi zarówno backend jak i frontend. Dzięki temu, gdy backend nadaje przeglądarce ciasteczko, myśli ona, że dotyczy ono właśnie tego frontend'u. Również rozwiązuje to także wszystkie problemy z CORS'em. Daje także możliwość ustawienia headerów Content Security Policy, aby zwiększyć bezpieczeństwo aplikacji.

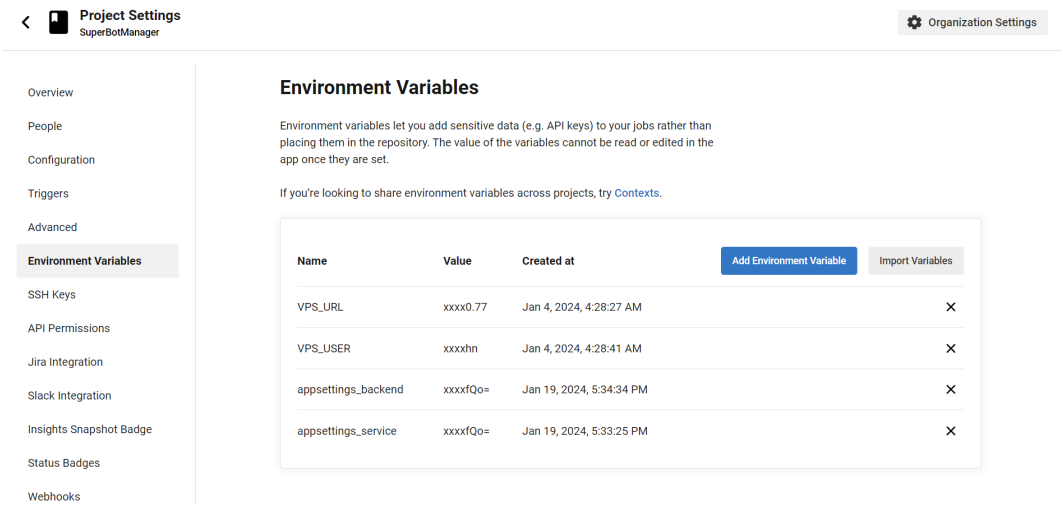
  Konfiguracja znajduje się w pliku playbooks/images/nginx/nginx.conf. Należy tam ustawić odpowiednie adresy domen.

- Mysql – konfiguracja w playbooks/images/mysql
- Postgres – konfiguracja w playbooks/images/postgres
- Selenium grid – konfiguracja w playbooks/images/selenium-grid
- RabbitMQ – konfiguracja w playbooks/images/rabbitmq

Wszystkie powyższe kontenery Docker'owe znajdą się w jednej sieci dockera, dzięki czemu mogą się ze sobą komunikować.

4.2. CircleCI [15]

W poprzednim podrozdziale skonfigurowaliśmy nasz VPS razem z wszystkimi serwisami potrzebnymi do uruchomienia naszej aplikacji. Teraz chcielibyśmy skonfi-



Rysunek 4.2: CircleCI zmienne środowiskowe

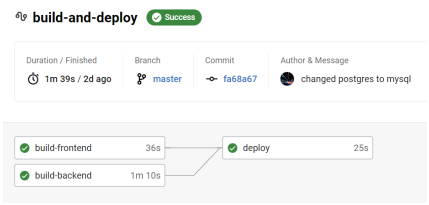
gurować rozwiązanie CI/CD, które zapewni nam wygodny sposób na robienie automatycznego deployment'u aplikacji po każdym commitcie. Przygotowałem gotowy pipeline w pliku .circleci/config.yml. Automatycznie buduje on trzy projekty: Frontend, Backend i ServiceConsumer. Następnie, jeżeli nie było żadnego błędu, łączy się z VPSem i wgrywa mu skompilowane pliki. Tworzy pliki appsettings.json dla projektów w C#. Na koniec uruchamia Backend i ServiceConsumer w nowych kontenerach.

Aby skonfigurować sobie CircleCI, należy założyć konto na ich stronie. Jest to rozwiązanie bezpłatne na nasze potrzeby. Podczas tworzenia projektów nie użyłem więcej niż 10% ich limitu, a resetuje się on co miesiąc.

Aby robił się poprawny deploy, musimy jeszcze dodać nasze appsettings.json, zawierające klucze i hasła do zmiennych środowiskowych CircleCI, jak widzimy na obrazku 4.2. Pliki te muszą być zakodowane w postaci bas64, aby nie zawierały niedozwolonych znaków. Pipeline je później odkoduje.

Ostatnią rzeczą jaką musimy zrobić jest dodanie w zakładce SSH keys, prywatnego klucza SSH do naszego VPS'a, aby CircleCI mógł wgrywać do niego projekty.

Możemy teraz uruchomić i podziwiać jak nasza aplikacja automatycznie wgrywa się na VPS'a po każdym commitcie do repozytorium. 4.3



Rysunek 4.3: CircleCI przykład

Rozdział 5.

Podobne rozwiązania

Projektując i implementując ten system, nie wzorowałem się na żadnym istniejącym rozwiązaniu. Wszystko wymyślałem na bieżąco, myśląc, że jestem dość oryginalny, w szczególności w pomyśle łańcuchowego wykonywania akcji.

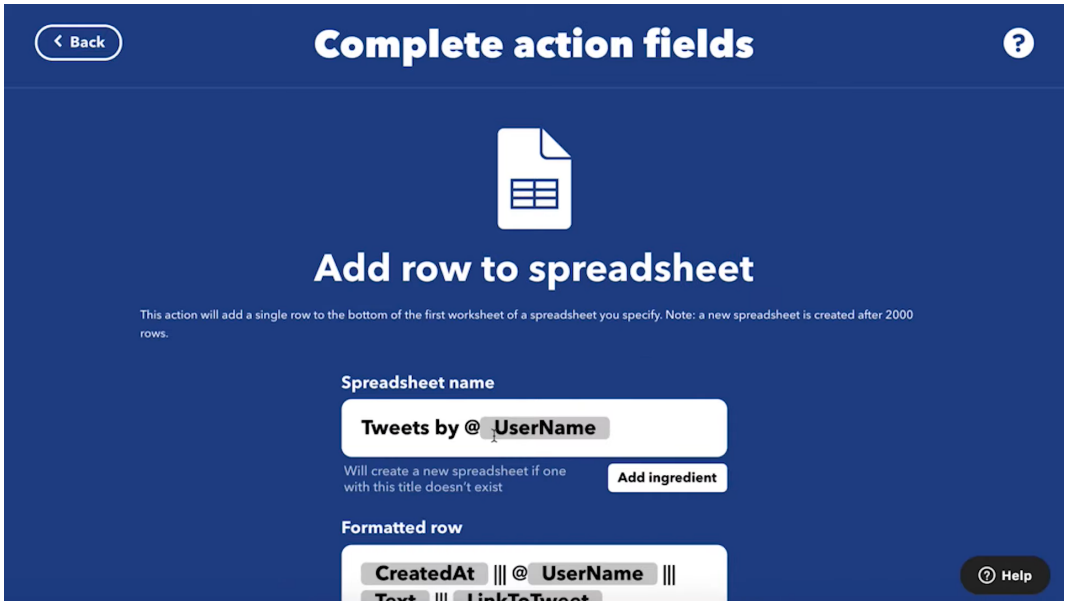
Co ciekawe okazało się, że istnieją już rozwiązania z praktycznie identycznym pomysłem jak mój.

5.1. IFTTT

Jednym z nim jest serwis https://ifttt.com/
Jest on podzielony na trzy główne obiekty: aplety, serwisy i akcje.
Serwis jest czymś w rodzaju zdarzenia (np. może być serwis: „gdy ktoś napisze do nas SMS'a"). Akcja jest zadaniem, które ma się wykonać, gdy zdarzenie zostanie wywołane. Tutaj także, akcje można uruchamiać łańcuchowo. Pierwszy serwis przekazuje swoje wyjście do pierwszej akcji, ona do kolejnej itd. Cały taki przebieg serwis + akcje jest nazywany w ich systemie Apletem.

Porównując to do mojego systemu, można zauważyć bardzo wiele podobieństw. Ich akcje są czymś w rodzaju moich ActionExecutor'ów, Tak samo definiują, co ma być na wejściu i ewentualnie z jakich danych wyjściowych poprzedniego zadania wziąć dane wejściowe, obrazek 5.1.

Dodatkowo można zauważyć, że u mnie obiekt serwis wywołujący akcje nie występuje. W moim systemie akcja może zostać uruchomiona ręcznie przez użytkownika, przez harmonogram zadań (w ich przypadku jest to serwis), albo przez inną akcję. Dlatego np. tym czym u mnie jest Discord – prompt u nich byłoby serwisem. Moje ActionExecutor'y są zatem uogólnieniem ich akcji i serwisów. Oba podejścia, zarówno moje jak i ich mają wady i zalety.



Rysunek 5.1: IFTTT definiowanie akcji

Zalety mojego podejścia oraz wady IFTTT:

- Ich system wymusza by najpierw były serwisy, a dopiero później akcje. Z tego powodu wielu sensownych przeplotów zadań nie można ułożyć. Mój system natomiast nie ma takiego ograniczenia. Np. Można zrobić Discord – Send Message, Discord – Prompt i ponownie Discord – Send Message. W ich rozwiązaniu Systemie Discord – Prompt musiałoby być na początku i nie można by wstawić nic przed nim.
- Uogólnienie akcji oraz serwisów pozwala pisać bardziej uniwersalny kod i zmniejsza szansę na potencjalne błędy/niespójności. Jest to jednak miecz obosieczny, ponieważ może chcielibyśmy mieć jakieś większe różnice?

Zalety IFTTT podejścia oraz wady mojego:

- Bardzo częstym przypadkiem w automatyzacji jest chęć zrobienia czegoś, gdy zajdzie jakieś zdarzenie. W przypadku ich systemu taki przepływ robi się bardzo intuicyjnie. W moim systemie akcje, które są odpowiednikiem ich serwisu, w kolejce są prezentowane jako ciągle In progress do momentu, aż coś by jej nie wywołało i wtedy by się zakończyła.
- Gdy mój odpowiednik ich serwisu doczeka się odpowiedniego zdarzenia, zakańcza on swoje działanie i uruchamia kolejną akcję. Gdy przebieg ten zostanie wykonany, moja akcja będąca odpowiednikiem serwisu jest teraz w stanie finished i nie jest uruchomiona. Przez to gdy drugi raz zdarzenie zostanie wywołane, system nie zareaguje. Problem ten można bardzo łatwo obejść, po prostu ustawiając ostatniemu ActionExecutorowi Run on finish jako z powrotem

pierwszą akcję (reprezentującą serwis). Dzięki temu powstanie cykl i zdarzenia będą dalej przechwytywane.

- W ich systemie grupa executor'ów będąca w jednym flow (w jednej spójnej składowej) nazywana jest apletem. Też myślałem na prowadzeniu takiego grupowania w moim systemie, ale ostatecznie się nie zdecydowałem. Teraz widzę, że jest to bardzo wygodne i warte zaimplementowania. Użytkownik nie ma potrzeby widzieć pośrednich executor'ów. W praktyce zawsze będzie chciał uruchomić tylko ten „na samej górze" będący pierwszą kostką domina.

Dodatkowo portal ten posiada osobną zakładkę stories, gdzie omawiają swoje akcje i prezentują idee, co można by dodatkowo zautomatyzować i jakie akcje użyć. Z pewnością będzie ona dla mnie dobrą inspiracją do dalszego rozwoju projektu.

5.2. Inne systemy

- **Zapier [22]** – Jest to równie popularne narzędzie co IFTTT, z podobną listą funkcjonalności. Główna różnica między nimi jest taka, że IFTTT jest bardziej nastawiony na zarządzanie Smart Home'm i pod ten cel ma utworzone większość predefiniowanych typów akcji. Oba systemy zostały utworzone już dosyć dawno. IFTTT w 2010 roku, a Zapier w 2011. Są już rozwijane kilkanaście lat.

  Zapier szczyci się tym, że ma około 3000 integracji do różnych aplikacji, podczas gdy IFTTT ma ich 600, ale za to jest tańsze.

  System ten nastawiony jest szczególnie na automatyzację procesów biznesowych np. automatyzację migracji danych między serwisami.

- **Microsoft Power Automate [23]** – Podobnie jak Zapier, serwis ten przeznaczony jest głównie do automatyzacji procesów biznesowych. Posiada on bardzo dobrą integrację z równymi aplikacjami internetowymi Microsoftu, a także z wieloma narzędziami AI.

Żaden z trzech wymienionych serwisów nie daje niestety możliwości wgrania własnego pełnoprawnego programu np. z botem, aby mógł pracować na ich serwerze. Posiadają one jedynie możliwość uruchomienia krótkiego kawałka kodu (Snippetów) w Python lub w JavaScript. Mogłyby się one nadawać np. do wysyłania zapytań http. Tylko Microsoft Power Automate daje możliwość uruchomienia własnej automatyzacji wykorzystującej Selenium.

Rozdział 6.

Podsumowanie

6.1. Metryki kodu

Narzędzie clock uruchomione dla głównego katalogu repozytorium projektu (bez node_modules oraz zainstalowanych nugetów zwraca następujące statystyki. Pomijam repozytorium VPS Manager.

```
> docker run --rm -v $PWD:/tmp aldanial/cloc /tmp
```

Language	files	blank	comment	code
C#	86	1136	145	6197
YAML	3	560	10	4191
TypeScript	64	247	69	3880
JSON	8	10	0	413
SVG	4	0	2	397
MSBuild script	10	38	0	205
Visual Studio Solution	1	1	1	117
Dockerfile	2	8	2	38
Markdown	1	8	0	19
HTML	1	0	0	13
CSS	2	1	0	7
SUM:	184	2009	229	15512

Widzimy, że większość kodu napisałem w C# (ponad 6 tys. linii). W Typescript natomiast napisałem prawie 4 tys. linii. W katalogu projektu frontend'u znajduje się także plik pnpm-lock.yaml, zawierający informacje o całym drzewie zależności w projekcie. Waży on 185KB i ma 4 tys. linii, dlatego narzędzie clock, pokazuje pliki .yaml jako drugi składnik projektu.

6.2. O temacie pracy

Tematem mojej pracy brzmi: "Porównanie zabezpieczeń przed botami w istniejących aplikacjach internetowych". Planując pracę kilka miesięcy temu miałem zamiar zrobić prosty system do uruchamiania botów i wiele przykładowych botów. W trakcie implementacji systemu trochę zmieniła mi się koncepcja i postanowiłem zbudować go najbardziej profesjonalnie jak potrafię i dołożyć maksymalnie wiele funkcji, aby projekt był dla mnie bardziej użyteczny na przyszłość.

Boty są zazwyczaj tymczasowe, po kilku miesiącach coś zmienia się w aplikacji i przestają działać. Natomiast system do ich uruchamiania, jeżeli jest napisany porządnie, może być użyteczny przez wiele lat. Rozpoczęcie od systemu pozwala także wyodrębnić części wspólne botów do niego, które w innym razie pisałoby się i zarządzało nimi osobno (co byłoby mniej wygodne).

6.3. Przyszły rozwój systemu

6.3.1. Co bym zmienił?

- Zabrałbym serwisom konsumentów możliwość łączenia się do bazy danych. Otrzymywaliby w jednej kolejce RabbitMQ akcje do wykonania, a w drugiej zwracaliby wynik, który następnie jakiś osobny mikroserwis, by przetwarzał i aktualizował bazę danych.

  Podejście to dałoby możliwość jeszcze prostszego wgrywania nowych konsumentów. Można by uruchamiać dowolne .dllki z konsumentami od użytkownika, bez ryzyka, że namieszają w bazie danych. Zwiększyłoby także bezpieczeństwo aplikacji.

- Zainspirowany podobnymi rozwiązaniami zastosowałbym jedną z bibliotek do tworzenia przepływów między komponentami, w celu wygodniejszego definiowania przebiegu łańcuchowego uruchamiania akcji. Np. wykorzystał bym bibliotekę react-flow.

6.3.2. Co bym dodał?

Z pewnością system będę wykorzystywał w praktyce oraz dalej go rozwijał. Teraz, gdy już wszystko działa, jak należy, będę mógł skupić na dodawaniu nowych akcji i botów, aby zwiększać użyteczność systemu. Chciałbym, także dodać kilka użytecznych funkcjonalności, które podpatrzyłem od innych systemów:

- Dodałbym dodatkowy typ akcji oprócz Manual i Automatic. Powodowałby on automatyczne, ponowne dodanie akcji do kolejki, gdy akcja zakończyła swoje działanie. W IFTTT zrobili na to osobny rodzaj obiektu o nazwie serwis.

- Dodałbym przycisk w kolejce, aby móc przerwać działanie akcji lub anulować akcję przed jej uruchomieniem.
- Dodałbym odpowiednik apletu podobnie jak jest w IFTTT. Zablokowałbym przy tym możliwość uruchamiania pośrednich executor'ów.

W najbliższym czasie mam zamiar dodać następujące typy akcji:

- Change LED color – z wykorzystaniem API Blebox wLightBox 3 zrobię akcję do zmiany koloru i jasności ledów. Dodam także w Harmonogramie zadań, aby na wieczór ich barwa była cieplejsza i były ciemniej, a rano barwa ustawiała się na białą i były jasno.
- Send HTTP Request – pozwoli to w uniwersalny sposób wysyłać proste zapytania HTTP i ich wynik przekazywać dalej
- Send Email – będzie to akcjam która pozwoli w prosty sposób wysłać email, z wiadomością automatycznie wygenerowaną przez poprzednią akcję.

Będę także rozwijał boty do różnych aplikacji.

Bibliografia

[1] RabbitMQ (https://rabbitmq.com/documentation.html)

[2] React (https://react.dev/reference/react)

[3] pnpm (https://pnpm.io/motivation)

[4] React Query (https://tanstack.com/query/latest/docs/react/overview)

[5] React i18Next (https://react.i18next.com/)

[6] Styled Components (https://styled-components.com/docs/basicsgetting-started)

[7] React Router (https://reactrouter.com/en/main/start/overview)

[8] Automapper (https://docs.automapper.org/en/latest/Getting-started.html)

[9] Selenium (https://www.selenium.dev/documentation/webdriver/getting$_s tarted/using_s elenium/$)

[10] Selenium Grid (https://www.selenium.dev/documentation/grid/)

[11] Ansible (https://www.ansible.com/resources/get-started)

[12] Docker (https://docs.docker.com/guides/get-started/)

[13] Reverse Proxy (https://en.wikipedia.org/wiki/Reverse$_p roxy$)

[14] Nginx (https://nginx.org/en/docs/beginners$_g uide.html$)

[15] CircleCI (https://circleci.com/docs/getting-started/)

[16] Entity Framework Core (https://learn.microsoft.com/en-us/ef/core/)

[17] C# Discord (https://discordnet.dev/guides/getting$_s tarted/first - bot.html$)

[18] JWT (https://jwt.io/introduction)

[19] SSL Pinning na Androidzie (https://medium.com/@anandgaur22/ssl-pinning-in-android-14851dc41703)

[20] Artkuł o RabbitMQ w ASP.NET Core: https://blog.devops.dev/robust-rabbitmq-implementation-for-asp-net-core-5672693544a2

[21] C# Discord (https://discordnet.dev/guides/getting$_started/first - bot.html$)

[22] Zapier (https://zapier.com/)

[23] Microsoft Power Automate (https://www.microsoft.com/pl-pl/power-platform/products/power-automate)