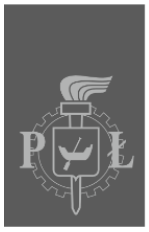


Politechnika Łódzka
Wydział Elektrotechniki Elektroniki Informatyki i Automatyki

Piotr Klimczak

239533

PRACA DYPLOMOWA

magisterska

na kierunku Informatyka

Rozproszony system uczenia maszynowego z wykorzystaniem kart graficznych

Instytut Informatyki Stosowanej

Promotor: dr inż. Robert Susik

ŁÓDŹ 2023

Streszczenie

Celem pracy jest stworzenie rozproszonego systemu wspomagającego uczenie maszynowe z wykorzystaniem kart graficznych. Zadanie polega na uruchomieniu treningu modelu na urządzeniu, którego karta nie jest w pełni obciążona. Zakres pracy obejmuje wybór technologii oraz zbudowanie aplikacji serwerowej i klienckiej. Wynikiem jest stworzony system, który spełnia założone cele oraz nie obciąża całkowitego czasu procesu uczenia.

Słowa kluczowe: AI, ML, GPU, Python, ASP.NET

Abstract

The aim of the work is to create a distributed system supporting machine learning with the use of graphics cards. The task is to run a model training on a device whose card is not fully loaded. The scope of work includes the selection of technology and building a server and client application. The result is a created system that meets the assumed goals and does not burden the total time of the learning process.

Keywords: AI, ML, GPU, Python, ASP.NET

Spis treści

1. Wstęp

Uczenie maszynowe to dziedzina, która łączy m.in statystykę i informatykę. Jest to część badań naukowych zajmujących się sztuczną inteligencją (ang. artificial intelligence, AI). Oczekiwanym celem może być stworzenie automatycznego systemu, który stale doskonali się bez ingerencji człowieka. Wykorzystuje m.in. zebrane dane i informacje, które są przetwarzane w celu znalezienia wzorców, których ludzie nie mogą dostrzec. W ten sposób maszyna może uczyć się nowych rzeczy tylko poprzez analizę istniejących informacji i wyciąganie z nich wniosków.

Rynek związany z uczeniem maszynowym jest obecnie bardzo rozwijającym się i dynamicznym sektorem technologicznym. Jak podaje BCC Research, wartość rynku ma urosnąć z 17,1 (2021 r.) do 90,1 miliarda dolarów (w 2026 r.), co daje skumulowany roczny wskaźnik wzrostu o wartości 39,4% za podany wyżej okres czasu (w badaniu brały udział takie firmy jak Alphabet (Google), Amazon, IBM, Microsoft, Oracle, SAP) [1]. Wynikiem zainteresowania tą tematyką jest coraz szersze stosowanie jej w wielu branżach i dziedzinach, takich jak:

- przemysł: do automatyzacji procesów produkcyjnych i optymalizacji ich kosztów operacyjnych [2]
- ochrona zdrowia: do wspierania diagnostyki, przewidywania różnych chorób (np. rak; choroby serca, nerek) [3]
- handel: do personalizacji ofert i rekomendacji produktów [4]
- finanse: wnioski na podstawie historii firmy i pomoc w decyzjach biznesowych [5]

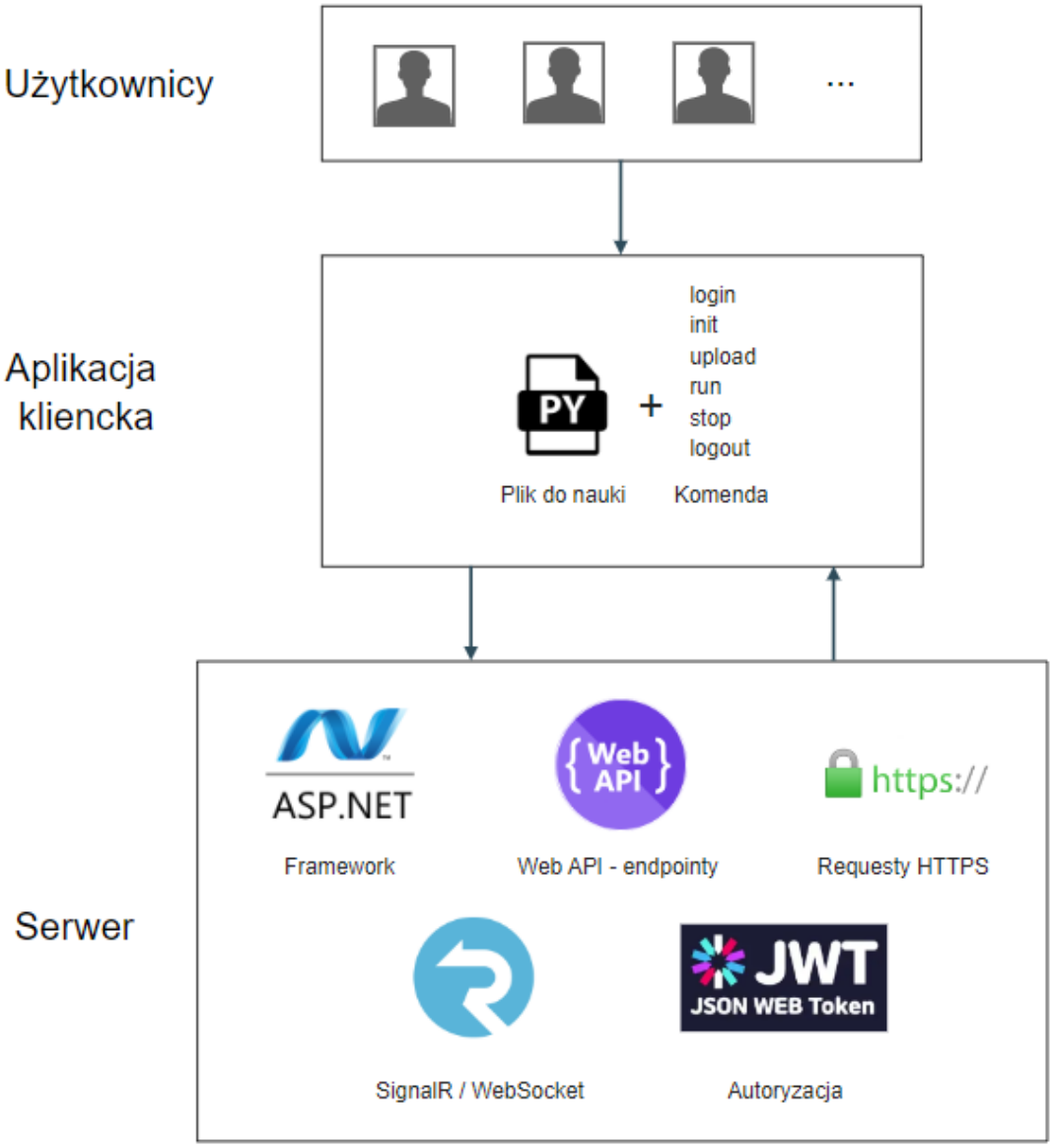
W wyniku tego rynek uczenia maszynowego jest obecnie jednym z najbardziej rozwijających się sektorów z rosnącym zapotrzebowaniem na specjalistów w tej dziedzinie i coraz większym wkładem inwestycji.

Podczas wybuchu pandemii COVID-19 w 2020 roku, coraz więcej firm zaczęło pracować zdalnie. Trend ten pozostał i wiele firm oferuje nadal taką możliwość. Firmy w takim przypadku przeprowadzają wdrożenie pracownika zdalnie, a sprzęt jest przysyłany do domu. Najczęściej, z powodu kosztów, postanawiają także kupować laptopy, które są dużo mobilniejsze i wygodniejsze dla pracowników, niż komputery stacjonarne. Mają one jednak, w odniesieniu do uczenia maszynowego wadę – najczęściej wersje biznesowe laptopów nie posiadają w ogóle karty graficznej (tylko zintegrowane o małej mocy), a to właśnie na niej są przeprowadzane najczęściej treningi modeli. Dodatkowo, jeżeli komputer przenośny posiada kartę, nie jest ona tak samo wydajna, jak wersja do komputerów stacjonarnych, co jest spowodowane rozmiarami, temperaturami, pobieraną mocą. Ponadto, ceny kart graficznych od czasów pandemii zdrożały z powodu ograniczenia produkcji i dostępności elementów elektronicznych [21].

Uczenie głębokie opiera się często na bardzo dużej liczbie danych wejściowych, z reguły podzielonych na zbiór treningowy, testowy i walidacyjny. Czym większa ilość danych, tym wymaga to coraz większej mocy obliczeniowej. Bardzo dużo także zależy problemu, który badamy oraz algorytmów. Jedną z możliwości jest kupno mocnej karty graficznej i udostępniać jej zasoby, niż kilkanaście słabszych (dodatkowo powoduje to skumulowany większy pobór mocy z sieci). Dodatkowo, jeżeli wielu użytkowników chce skorzystać z GPU, to potrzebny jest system, który będzie zarządzał. Alternatywną opcją jest korzystanie z usług chmurowych, lecz te są bardzo drogie – przykładowy koszt korzystania z maszyny z kartą na Google Cloud to ponad 2000$ miesięcznie [22].

Z powyższych powodów postanowiłem napisać pracę na temat „Rozproszony system uczenia maszynowego z wykorzystaniem kart graficznych”, która ma w celu stworzenie oprogramowania umożliwiającego klientom na zdalne uruchomienie uczenia maszynowego na komputerze, na którym karta graficzna nie jest obciążona.

2. Wprowadzenie



Rys. 1 – Schemat pracy magisterskiej. [Źródła logotypów w dziale 10.]

Praca została podzielona na 2 części: aplikacje kliencką i serwerową.

Schemat stworzonego systemu (rys. 1), pokazuje elementy składowe, m.in. aplikacje, z której korzystają użytkownicy, gdzie wskazując plik i komendę (z parametrami) wysyłają dane na serwer, który następnie zarządza procesem.

Aplikacja przeznaczona użytkownikom jest napisana w języku Python. Jest ona uruchamiana jako plik wykonywalny (w konsoli) w danym folderze, w którym użytkownik posiada m.in. swoje pliki do nauki. Konfiguracja została ograniczona do minimum. Użytkownik podaje do komend tylko niezbędne parametry, np. nazwę pliku czy ilość pamięci karty graficznej, która jest niezbędna do ukończenia procesu trenowania modelu. Przesłanką do napisania aplikacji w taki sposób jest idea działania systemu kontroli wersji Git (także aplikacja konsolowa oraz zbliżone komendy) – jest to proste i jednocześnie efektywne narzędzie.

Drugim elementem systemu jest serwer napisany w ASP.NET. Przyjmuje on i wykonuje wszystkie żądania przychodzące od wyżej wymienionej aplikacji, które są wysyłane poprzez HTTPS (rodzaj protokołu, który szyfruje połączenie wykorzystując certyfikat). Serwer obsługuje także żądania WebSocket, które umożliwiają wymianę danych na żywo, co pozwala użytkownikowi na podgląd treningu w czasie rzeczywistym. Wybrano te technologie, ponieważ są to znane rozwiązania (długo dostępne na rynku) oraz istnieją do nich obszerne dokumentacje techniczne. Wykorzystano także standard JWT (JSON Web Token), do implementacji systemu autoryzacji użytkowników. Po zalogowaniu, do każdego żądania do serwera, trzeba dodać wygenerowany klucz. Zwiększa to bezpieczeństwo przed nieuprawnionym dostępem do zasobów.

3. Cel i założenia projektu

Celem pracy jest stworzenie systemu, który umożliwi uczenie na zasobach sprzętowych, które nie są w pełni obciążone, dlatego w pracy zrealizowano:

a. Przegląd dostępnych narzędzi
b. Część kliencką - aplikację jako interfejs wiersza poleceń (ang. CLI – Command Line Interface)
c. Część serwerową

Rozdział 4. i 5. przedstawia przegląd dostępnych technologii (tj. języków, bibliotek) w odniesieniu do aplikacji oraz serwera. W rozdziale 6. pokazano zasadę działania obu części. Następnie, w rozdziale 7. jest przeprowadzone badanie, które mierzy dodatkowy czas spowodowany użytkowaniem stworzonego systemu (inicjalizacja oraz przesyłanie danych).

Opracowany system nosi nazwę AIHUB, która powstała ze złożenia dwóch słów: AI odpowiada sztucznej inteligencji / uczeniu maszynowemu, a HUB jest punktem połączenia różnych przychodzących danych w jedno miejsce.

4. Przegląd dostępnych narzędzi – aplikacja kliencka

4.1 Python

Python jest dynamicznym, wysokopoziomowym językiem programowania, który został zaprojektowany z myślą o czytelności i łatwości użycia [6]. Jest on uważany za jeden z najbardziej elastycznych i uniwersalnych języków programowania, co oznacza, że może być stosowany do wielu różnych celów, w tym web development, analiza danych, automatyzacja.

Język ten jest wieloparadygmatowy, ponieważ jest w pełni zgodny z zasadami programowania obiektowego i funkcyjnego. Posiada rozbudowane biblioteki i narzędzia, takie jak NumPy, Pandas, Matplotlib i TensorFlow, które umożliwiają projektowanie i implementację zaawansowanych aplikacji w dziedzinie statystyki czy uczenia maszynowego.

Python jest również językiem o otwartym kodzie źródłowym, co oznacza, że jest on całkowicie darmowy i dostępny dla wszystkich [6], a także posiada silną i aktywną społeczność programistów, którzy stale rozwijają i ulepszają język.

Jednym z głównych zalet Pythona jest jego czytelność i łatwość użycia. Składnia Pythona jest zaprojektowana tak, aby była zrozumiała dla programistów na wszystkich poziomach doświadczenia, co pozwala na szybsze i bardziej efektywne tworzenie kodu.

W projekcie język ten został użyty do stworzenia aplikacji klienckiej, która jest uruchamiana w konsoli. Użyte biblioteki oraz utworzona architektura (modularyzacja klas i plików) pozwala na łatwą skalowalność aplikacji oraz jej utrzymanie.

4.2 Biblioteka „Click”

```
Usage: client.exe [OPTIONS] COMMAND [ARGS]...

  AIHUB CLI

Options:
  -h, --host TEXT  AIHUB server endpoint
  -q, --quiet      AIHUB will be quiet during operations (logs only in file)
  --help           Show this message and exit.

Commands:
  init    Initialize new project.
  login   Allows user to login with his username and password.
  logout  Logout current user.
  run     Runs given filename, depending if VRAM is avalaible.
  stop    Stop current running process.
  upload  AIHUB will upload contents of the current directory to server.
```

Rys. 2 – Wynik działania biblioteki „Click” w aplikacji klienckiej.

Click to biblioteka języka Python służąca do tworzenia aplikacji w linii poleceń (ang. CLI) [7]. Jest to narzędzie łatwe w użyciu i elastyczne, umożliwiające szybkie i efektywne tworzenie aplikacji konsolowych.

Za pomocą Click można łatwo definiować polecenia, jak również personalizować sposób wyświetlania dostępnych argumentów i informacji. Biblioteka udostępnia także narzędzia umożliwiające automatyzację procesów i skryptów, co umożliwia oszczędność czasu i zwiększenie efektywności pracy. Możliwe jest także dodawanie własnych komunikatów błędów.

W omawianej pracy biblioteka została użyta do wyświetlania informacji o dostępnych komendach i pomocy oraz obsłudze podanych parametrów w kodzie.

4.3 Biblioteka „Marshmallow”

```python
from marshmallow import Schema, fields
from aihub.domain.base import BaseModel

class LoginSchema(Schema):
    username = fields.Str()
    password = fields.Str()

class LoginData(BaseModel):
    schema = LoginSchema()

    def __init__(self, user, password):
        self.username = user
        self.password = password
```

Listing 1 – Przykład użycia biblioteki Marshmallow w kodzie aplikacji klienckiej.

Marshmallow to biblioteka do serializacji i deserializacji obiektów Python, pozwalająca na łatwe mapowanie obiektów na formaty takie jak JSON i odwrotnie [8].

Główne cechy tej biblioteki to:

- Możliwość definiowania schematów danych za pomocą prostych klas
- Wsparcie dla różnych typów pól, takich jak tekst, liczby, daty i godziny, zagnieżdżone obiekty i inne
- Funkcja walidacji danych przed serializacją lub deserializacją
- Automatyczna obsługa relacji między obiektami
- Niestandardowe mapowanie nazw pól

Marshmallow jest prostym i elastycznym narzędziem, które umożliwia łatwe przekształcanie danych w aplikacjach internetowych. Jest łatwe w użyciu i posiada bogatą dokumentację, co umożliwia szybkie i skuteczne wykorzystanie jego funkcjonalności w projektach.

Na listingu 1. został pokazany przykład użycia (model używany podczas logowania użytkownika) omawianej biblioteki w kodzie aplikacji klienckiej. Stworzoną i zasiloną danymi klasę *LoginData* można przekazać np. do biblioteki Requests i wysłać do serwera. Przyśpiesza to pracę i normalizuje użyte modele w kodzie.

4.4 Biblioteka „Requests”

Requests to biblioteka w języku Python, która umożliwia łatwe wysyłanie żądań HTTPS i obsługę odpowiedzi [9]. Dzięki niej można w prosty sposób wysyłać różne rodzaje, takie jak GET, POST, PUT, DELETE i inne, a także łatwo obsługiwać nagłówki HTTP, pliki i dane binarne.

Główną zaletą Requests jest to, że umożliwia wysyłanie żądań HTTP bez konieczności ręcznego tworzenia i parsowania nagłówków i danych. Biblioteka ta automatycznie obsługuje kod odpowiedzi HTTP, takie jak 200 (OK), 404 (Not Found) i inne, co pozwala na łatwą i szybką weryfikację poprawności.

Biblioteka Requests jest również wyposażona w funkcjonalność uwierzytelniania i certyfikatów SSL/TLS, co pozwala na bezpieczne wysyłanie żądań do serwera. To ważne, ponieważ wiele serwisów internetowych i API wymaga certyfikatów, aby chronić dane i zapewnić bezpieczeństwo użytkowników.

Requests jest również łatwym do zintegrowania z innymi bibliotekami, takimi jak biblioteki do przetwarzania JSON i XML, co pozwala na łatwe przetwarzanie odpowiedzi z serwera. Pozwala to na stworzenie bardziej zaawansowanych i skomplikowanych aplikacji, które wymagają wysyłania żądań HTTP.

W pracy skorzystano tej biblioteki. Używa ona wcześniej wspomniany certyfikat, ponieważ serwer przyjmuje żądania tylko w postaci zabezpieczonej, tj. HTTPS. Główną metodę tej biblioteki, czyli „request” wykorzystano w postaci podstawowego klienta w aplikacji i zależnie od przekazanych wcześniej informacji (najczęściej danych opakowanych w bibliotekę Marshmallow) są one wysyłane do serwera, a wynik jest odbierany. Dodana jest także obsługa potencjalnych błędów, takich jak brak połączenia z serwerem.

4.5 Biblioteka „Websockets”

```
1. uri = f"""{self.wss_url}/{self.hub_url}?id={connectionId}
2.            &projectName={project_name}&access_token={self.access_token}"""
3. async with websockets.connect(uri, ssl=ssl_context) as websocket:
```

Listing 2 – Przykład użycia Websockets – łączenie do serwera.

WebSockets to protokół komunikacji w czasie rzeczywistym, który umożliwia dwukierunkową komunikację między klientem a serwerem przez pojedyncze połączenie. Protokół ten jest oparty na standardzie Internet Engineering Task Force (IETF) i jest opracowywany jako część HTML5.

Biblioteka WebSockets jest implementacją protokołu o tej samej nazwie, która umożliwia programiście łatwe tworzenie aplikacji sieciowych [10]. Biblioteka ta może być wykorzystywana w różnych językach programowania, m.in. w Pythonie (przykład w listingu 2).

Głównym celem biblioteki jest umożliwienie aplikacjom komunikowanie się z serwerem i klientem w czasie rzeczywistym, bez konieczności wielokrotnego wysyłania żądań i odpowiedzi. Często jest też stosowana w aplikacjach, takich jak gry online, czaty, aplikacje do współpracy w czasie rzeczywistym, a także aplikacje do transmisji na żywo.

Użyta biblioteka pozwala na uzyskanie pełnej funkcjonalności protokołu WebSockets, w tym:

- Dwukierunkową komunikację: Dzięki WebSockets, klient i serwer mogą wysyłać i odbierać dane jednocześnie, co pozwala na szybką reakcję na zdarzenia.
- Niskie opóźnienia: Umożliwia komunikację w czasie rzeczywistym bez konieczności opóźnień, co jest szczególnie istotne w aplikacjach do transmisji na żywo.

- Oszczędność zasobów: Dzięki pojedynczemu połączeniu, WebSockets pozwala zaoszczędzić zasoby, ponieważ nie trzeba wielokrotnie nawiązywać i zrywać połączenia, jak to ma miejsce w przypadku protokołu HTTP.
- Bezpieczeństwo: Protokół WebSockets jest bezpieczny i może być używany w połączeniach SSL, co zapewnia ochronę przed atakami i nieautoryzowanym dostępem do danych.

W aplikacji klienckiej wykorzystano bibliotekę Websockets w celu uruchomienia treningu oraz poglądzie na żywo wyniku uczenia. Na serwerze odpowiedzialnym za ten protokół jest SignalR.

4.6 Biblioteka „Asyncio”

```
1. async def start_pinging():
2.     nonlocal _running
3.     while _running:
4.         await asyncio.sleep(10)
5.         await websocket.send(self.toSignalRMessage({"type": 6}))
```

Listing 3 – Przykład użycia asyncio wraz z websocket.

Biblioteka asyncio jest biblioteką do programowania asynchronicznego w języku Python [11]. Jest to narzędzie, które pozwala programistom na tworzenie aplikacji, które są w stanie efektywnie i elastycznie reagować na zdarzenia i wymianę danych z wieloma klientami jednocześnie.

Programowanie asynchroniczne jest przydatne w sytuacjach, w których aplikacja musi współpracować z wieloma zdarzeniami jednocześnie, takimi jak połączenia z siecią, wejście z klawiatury i plików itp. W tradycyjnym, synchronicznym podejściu, aplikacja musi czekać na każde zdarzenie, aby je obsłużyć, co powoduje, że aplikacja jest blokowana i nie jest w stanie reagować na inne zdarzenia. W przypadku programowania asynchronicznego, aplikacja jest w stanie reagować na wiele zdarzeń jednocześnie, co pozwala na lepszą współpracę.

Biblioteka jest wyposażona w wiele narzędzi i funkcji, które umożliwiają programistom łatwe tworzenie aplikacji asynchronicznych. W szczególności, biblioteka udostępnia takie narzędzia, jak pętle wydarzeń, zadania (ang. tasks) i kolejki zdarzeń, które pozwalają na zarządzanie wieloma zdarzeniami jednocześnie.

Asyncio szeroko stosowana w różnych dziedzinach, takich jak sieci, gry, interfejsy użytkownika i wiele innych. Warto zaznaczyć, że asyncio jest biblioteką, która jest szybka i efektywna. Dzięki temu, aplikacje zbudowane z wykorzystaniem asyncio i websocketów są w stanie współpracować z dużymi ilościami danych, bez zauważalnego spadku wydajności.

Jak pokazano na listingu 3, asyncio w celu asynchronicznej wymiany danych z serwerem. Pozwala na odpytywanie co 10 sekund serwera tzw. „ping”, czyli podtrzymanie komunikacji. Używając tej biblioteki, nie musimy blokować na 10 sekund całej aplikacji, tylko asynchronicznie wykonujemy co określony interwał operację. Pozwala to na dalszy odsłuch pozostałych informacji, np. uczenia modelu maszynowego na żywo.

4.7 Biblioteka „logging”

Listing 4 – Część logów z pliku *logs.log* pokazująca informację o uruchomieniu aplikacji i wykonaniu zapytania do serwera odnośnie logowania.

Biblioteka logging jest jedną z podstawowych bibliotek standardowych języka Python [12] i jest często używana do logowania informacji o stanie aplikacji. Jest to potężne narzędzie, które umożliwia łatwe i elastyczne logowanie w aplikacjach Python.

Główne cechy biblioteki to:

- Poziomy logowania: Udostępnia kilka poziomów logowania, takich jak DEBUG, INFO, WARNING, ERROR i CRITICAL, które pozwalają użytkownikowi filtrować i kontrolować informacje, które są zapisywane.
- Źródła logów: Mogą być zapisywane do pliku (listing 4), wyświetlane na konsoli, wysyłane do sieci lub przekazywane do innych systemów.
- Formatowanie logów: Biblioteka umożliwia formatowanie logów, w tym dodawanie dodatkowych informacji, takich jak data i godzina, nazwa pliku, numer wiersza itp.
- Konfiguracja: Logging umożliwia konfigurację poprzez plik konfiguracyjny lub bezpośrednio za pomocą kodu, co umożliwia łatwe dostosowywanie logowania w miarę potrzeb.
- Wiele handlerów: Oprogramowanie umożliwia ustawienie wielu handlerów, które obsługują różne źródła logów.

W pracy użyto tej biblioteki oraz skonfigurowano tak, że użytkownik ma wybór (rys. 2 – opcja *-q* lub *--quiet*), czy chce mieć pokazywane opcjonalnie logi w konsoli, czy tylko sam zapis do pliku. Logi takie pozwalają na szybsze odnalezienie potencjalnej ścieżki z problemem, jeżeli taki by wystąpił i na szybsze jego naprawienie.

5. Przegląd dostępnych narzędzi – serwer

5.1 ASP.NET Web API

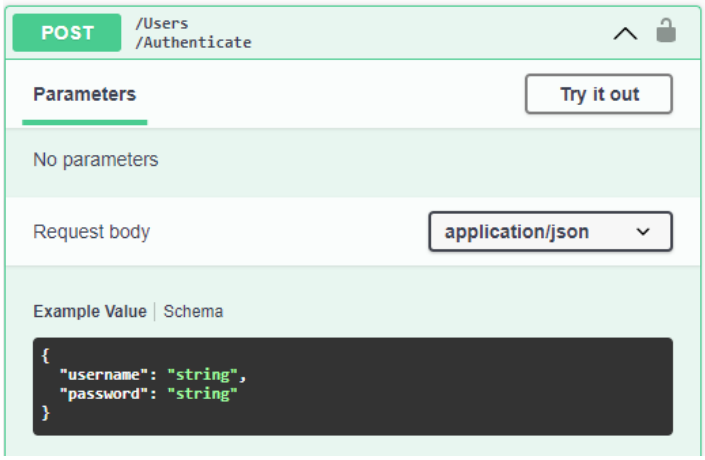
ASP.NET Core Web API to platforma do tworzenia aplikacji sieciowych, oparta na języku programowania C# i frameworku .NET [13]. Jest to bardzo elastyczne i wydajne narzędzie, które umożliwia programistom tworzenie aplikacji, które są proste, szybkie i bezpieczne. Zapewnia szereg narzędzi i funkcji umożliwiających tworzenie skalowalnych, wydajnych i łatwych w integracji API.

ASP.NET oferuje wiele narzędzi do zarządzania sesjami i autoryzacją, które pozwalają kontrolować dostęp do danych. Możliwa jest też integracja z bazami danych, która pozwala na łatwe przechowywanie i zarządzanie informacjami, a narzędzia do obsługi danych umożliwiają łatwe zbieranie i przetwarzanie informacji od użytkowników.

Web API jest także w pełni kompatybilne z innymi platformami i narzędziami, takimi jak LINQ, Entity Framework i inne narzędzia Microsoftu, co pozwala na łatwą integrację z istniejącymi systemami i aplikacjami.

W projekcie użyto właśnie ASP.NET jako bazę pod API. Z wyżej wymienionych powodów, łatwości dodawania nowych integracji (np. z usługami chmurowymi czy kolejnymi serwisami), i wsparciem z strony twórców (co roku są kolejne wersje ASP.NET wraz z frameworkiem .NET), czyli firmy Microsoft, stanowi to doskonały wybór do większości aplikacji, stosowanych także na szeroką skalę.

5.2 Narzędzie „Swagger”



Rys. 3 – Endpoint */Users/Authenticate* używanego do logowania, opisany w Swaggerze.

Swagger to narzędzie do tworzenia i dokumentowania interfejsów API. Jest to szczególnie ważne, ponieważ API stanowi jeden z głównych sposobów komunikacji między różnymi aplikacjami i systemami. Aby umożliwić innym programistom łatwą obsługę systemu, należy zapewnić im niezbędną dokumentację. Narzędzie to można go łatwo integrować z aplikacjami ASP.NET Core dzięki Swashbuckle [14].

Swashbuckle jest biblioteką, która automatycznie generuje dokumentację API opartą na adnotacjach w kodzie. Dzięki temu, gdy on się zmieni, dokumentacja zostanie automatycznie zaktualizowana. Swashbuckle również umożliwia łatwe testowanie API z poziomu interfejsu użytkownika Swagger. Pokazano to na rys. 3. Dodatkowo umożliwia dodawanie przykładów danych wejściowych i wyjściowych do dokumentacji API. Pozwala to programistom łatwiej zrozumieć, jak dane są przesyłane i jakie są oczekiwane wyniki.

W pracy magisterskiej wykorzystano Swagger, ponieważ jest bardzo przydatnym narzędziem do dokumentowania interfejsów. Pozwala to na zwiększenie przejrzystości i łatwości korzystania z API, co może przyczynić się do skrócenia czasu potrzebnego na rozwijanie aplikacji. W środowisku ASP.NET, biblioteka Swashbuckle umożliwia łatwe dodanie Swaggera do projektu i jego integrację z aplikacją.

5.3 Biblioteka „MediatR”

```
1. [Authorize]
2. [HttpPost("Init")]
3. [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(ExperimentsInitCommandResult))]
4. [ProducesResponseType(StatusCodes.Status400BadRequest, Type = typeof(ExperimentsInitCommandResult))]
5. public async Task<IActionResult> ExperimentsInit(ExperimentsInitCommand query)
6. {
7.     return await Mediator.Send(query);
8. }
```

Listing 5 – Przykład kodu kontrolera po użyciu „MediatR”.

MediatR to biblioteka, która umożliwia implementację wzorca projektowego znanego jako "mediator" [15]. Polega on na oddzieleniu logiki biznesowej aplikacji od komunikacji między jej poszczególnymi składnikami. Zamiast bezpośrednio wysyłać żądania do innych komponentów aplikacji, składniki wysyłają żądania do pośrednika, który jest odpowiedzialny za ich przekazywanie do odpowiedniego komponentu. Dzięki temu kod jest mniej zależny od siebie, a komunikacja jest bardziej elastyczna i łatwa do zmiany w przypadku potrzeby.

MediatR w ASP.NET umożliwia implementację wyżej wymienionego wzorca, co pokazano na listingu 5. Udostępniony jest także interfejs do wysyłania i odbierania żądań. Żądania są reprezentowane przez klasy, a każde z nich może mieć dedykowaną metodę, która jest odpowiedzialna za jego obsługę.

Korzystanie z MediatR pozwala na utrzymanie czystego i modularnego kodu, ponieważ każdy plik jest odpowiedzialny tylko za jedną konkretną operację. Dzięki temu kod jest łatwiejszy do testowania i rozwijania, a także łatwiejszy do zrozumienia dla innych programistów.

MediatR także umożliwia implementację asynchronicznej komunikacji między składnikami aplikacji, co jest szczególnie przydatne w przypadku aplikacji internetowych, które często wymagają wielu operacji równolegle.

5.4 SignalR

SignalR to biblioteka sieciowa dla platformy ASP.NET, która umożliwia tworzenie aplikacji w czasie rzeczywistym [16]. Pozwala na łatwe i szybkie tworzenie aplikacji, które wymagają bieżącej komunikacji z użytkownikami, takich jak aplikacje czatu, gry multiplayer, aplikacje do monitorowania i wiele innych.

SignalR umożliwia komunikację w obie strony, co oznacza, że aplikacja może wysyłać i odbierać dane jednocześnie. To powoduje, że użytkownicy mogą natychmiast otrzymywać informacje z aplikacji, bez konieczności odświeżania. Biblioteka korzysta z technologii WebSockets, aby umożliwić wymianę danych w czasie rzeczywistym.

SignalR umożliwia łatwe tworzenie grup użytkowników, co oznacza, że można wysyłać wiadomości tylko do określonej grupy użytkowników, a nie do wszystkich użytkowników naraz. To pozwala na bardziej zaawansowane scenariusze komunikacji, takie jak użyte w projekcie – można włączyć kilka instancji aplikacji klienckiej i na każdej z nich mieć podgląd treningu na żywo, jeżeli jesteśmy w zasięgu tego samego projektu. Dodatkowo biblioteka ta wspiera wielowątkowość, więc możliwe jest np. sterowanie procesem z kolejnej odpalonej aplikacji.

5.5 Wzorzec „Dependency Injection”

```
1. services.AddTransient<IProjectFolderService, ProjectFolderService>();
2. services.AddScoped<IUserService, UserService>();
3. services.AddSingleton<ISingletonSignalRService, SingletonSignalRService>();
```

Listing 6 – Przykład zarejestrowanych serwisów w kodzie serwera.

Dependency Injection (lub w skrócie: DI) jest jednym z najważniejszych wzorców projektowych w programowaniu obiektowym. Jego głównym celem jest oddzielenie kodu od jego zależności, aby był on bardziej elastyczny i łatwiejszy do testowania. W ASP.NET skorzystano z biblioteki *Microsoft.Extensions.DependencyInjection*, aby wdrożyć DI [17]. Wzorzec ten jest ostatnią literą w skrócie SOLID, czyli jednej z głównych zasad programowania. W skrócie, DI mówi o tym, że powiązania pomiędzy klasami powinny być luźne.

W DI, obiekty nie tworzą bezpośrednio swoich zależności, ale są one dostarczane im przez inne komponenty. W ten sposób, obiekt nie jest zależny od konkretnych implementacji swoich zależności, ale jest zależny od interfejsów. Dzięki temu, zależności mogą być zmieniane bez wprowadzania zmian do kodu samego obiektu, co jest szczególnie ważne w procesie testowania.

Aby wdrożyć DI w C#, należy najpierw zdefiniować interfejsy dla wszystkich zależności. Następnie, implementacje interfejsów są rejestrowane w kontenerze DI, który jest odpowiedzialny za tworzenie i udostępnianie instancji implementacji. W aplikacji, obiekty otrzymują swoje zależności przez konstruktor, właściwość lub metodę, które są wstrzykiwane przez kontener DI.

W tym wzorcu, jak pokazano na listingu 6, istnieją 3 typy możliwości rejestrowania zasobów, tj.:

- *Singleton* – pozostaje takim samym obiektem cały czas
- *Scoped* – obiekt jest taki sam w trakcie trwania przetwarzania żądania na serwerze, ale różnym pomiędzy różnymi zapytaniami

- *Transient* – obiekt za każdym razem jest inny, zawsze jest tworzona nowa instancja.

W kodzie serwera także wykorzystano omawiany wzorzec DI, przede wszystkim z powodu oddzielenia logiki w kodzie oraz swobodnego implementowania na żądanie pomiędzy różnymi plikami. Każdy zaawansowany kod w obecnych czasach korzysta z tego wzorca, także w innych językach programowania.

5.6 JWT Token

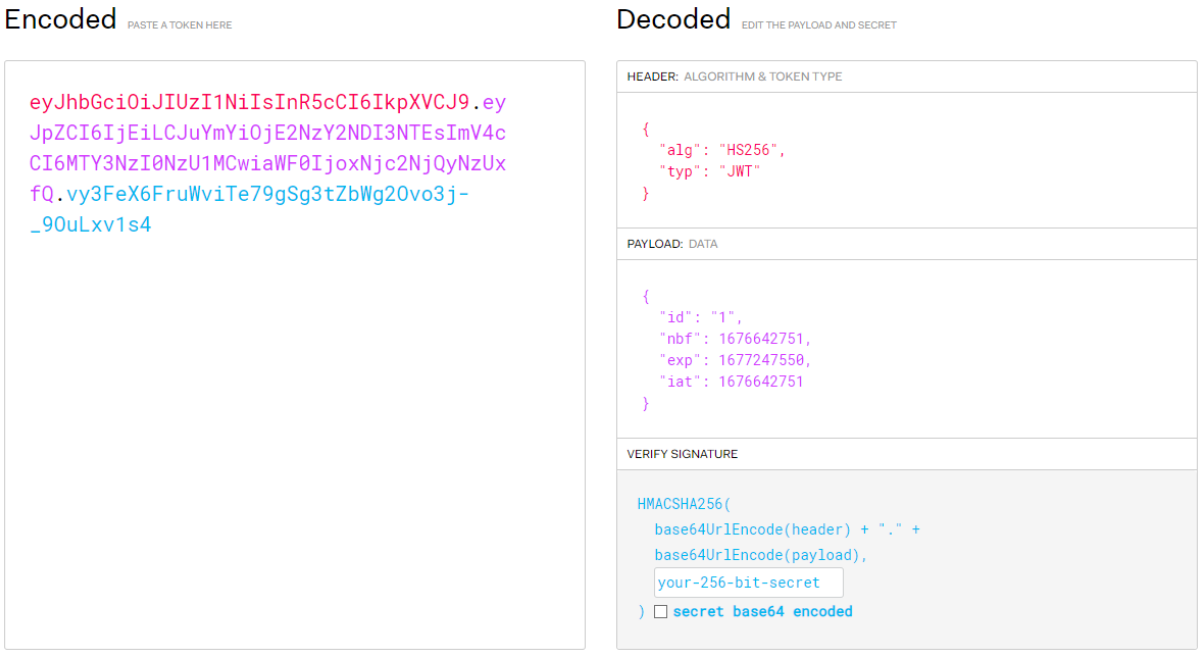
JWT (JSON Web Token) to standardowy format tokenu uwierzytelniającego oparty na formacie JSON. JWT składa się z trzech części: nagłówka, ciała i sygnatury, które są oddzielone kropkami. Każda z tych części jest zwykle kodowana w formacie Base64, co umożliwia łatwe przesyłanie tokenu między serwerami.

Nagłówek (sekcja „Header” na rys. 4) JWT posiada informacje o tym, jakie algorytmy szyfrowania zostały wykorzystane do generowania tokenu, a także informacje o typie tokenu, którym jest JWT.

Ciało (część „Payload” na rys. 4) zawiera informacje o użytkowniku, który został uwierzytelniony, oraz inne metadane, takie jak czas wygaśnięcia tokenu. Może także zawierać niestandardowe informacje specyficzne dla aplikacji.

Sygnatura (oznaczona jako „Verify signature” na rys. 4) JWT jest generowana na podstawie nagłówka i ciała tokenu, a także tajnego klucza prywatnego, który jest znany tylko serwerowi, który go wygenerował. Sygnatura jest weryfikowana przez serwer, aby upewnić się, że nie został sfałszowany lub zmodyfikowany w transporcie.

JWT jest często wykorzystywany do uwierzytelniania i autoryzacji użytkowników w aplikacjach internetowych i mobilnych. Jest to popularna metoda, ponieważ pozwala na przenoszenie informacji o użytkowniku między różnymi serwerami, co umożliwia bezpieczne korzystanie z różnych usług. JWT jest także stosowany w mikrousługach, w których użytkownicy często muszą uwierzytelniać się na różnych serwerach w ramach jednego żądania.



Rys. 4 – Przykład zdekodowanego JWT tokenu na stronie jwt.io.

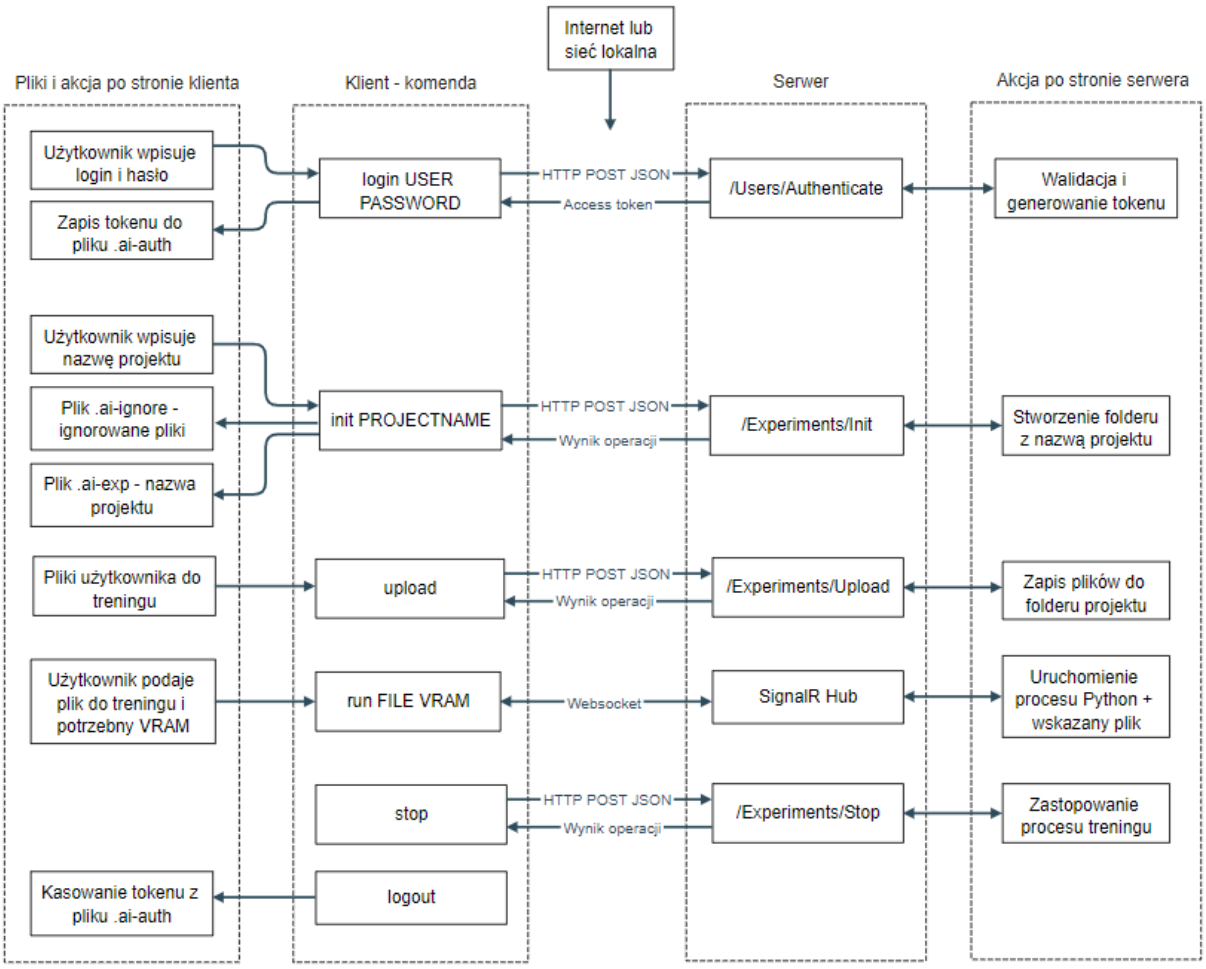
6. Oprogramowanie rozproszonego systemu do uczenia maszynowego

Rozdział ten opisuje oprogramowanie odpowiednio do:

- Części klienckiej
- Części serwerowej

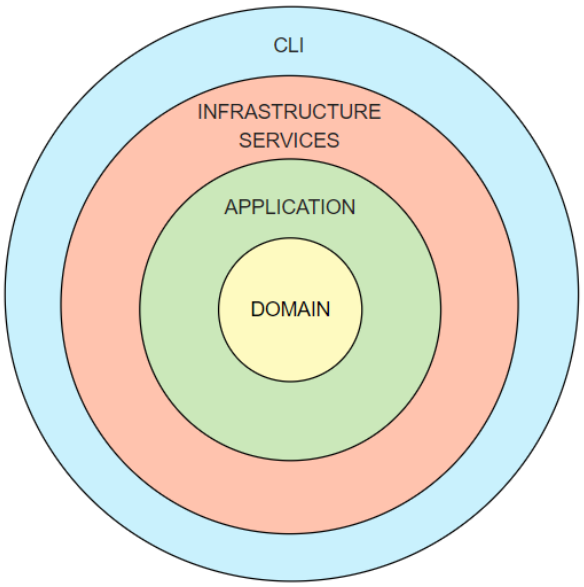
Do edycji wykorzystano edytor kodu Visual Studio Code oraz Visual Studio w wersji Community. Pierwszy z nich umożliwia pisanie w języku Python, co jest wykorzystane w części klienckiej, drugi z nich natywnie i lepiej wspiera język C# i framework .NET, użyty w stworzonym serwerze.

Poniżej (rys. 5) przedstawiono schemat działania od strony programistycznej:

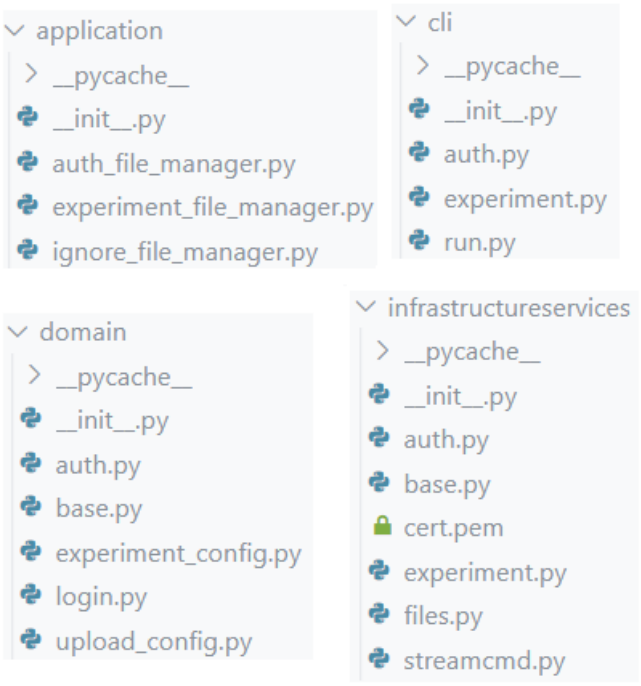


Rys. 5 – Zasada działania software – schemat blokowy.

6.1 Architektura aplikacji klienckiej



Rys. 6 – Graf przedstawiający zależność poszczególnych projektów w aplikacji.



Rys. 7 – Podział plików w folderach aplikacji klienckiej

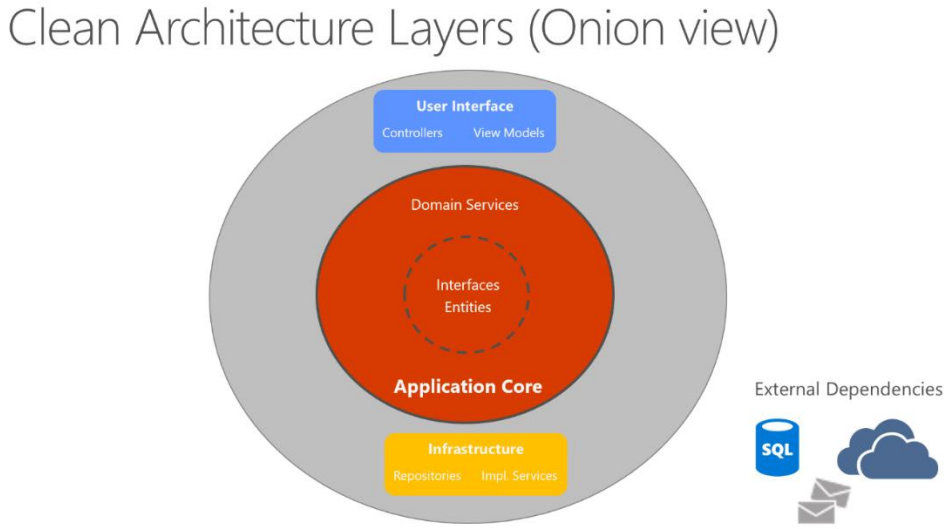
Architektura aplikacji klienckiej została podzielona na 4 zintegrowane ze sobą elementy:

- Domena (ang. Domain) – są to pliki, które zawierają opis od strony wysokopoziomowej logiki w zakresie danych, jakie są obsługiwane w aplikacji, np. zmienne przechowujące nazwę użytkownika i hasło podczas logowania. Wykorzystywana jest tutaj głównie biblioteka „Marshmallow".
- Aplikacja (ang. Application) – sekcja ta jest odpowiedzialna za obsługę wszystkich wydarzeń w ramach działania aplikacji, w tym przypadku są one odpowiedzialne za pliki *.ai-auth*, *.ai-exp* oraz *.ai-ignore*, które kolejno przechowują token po zalogowaniu, informacje o nazwie projektu i pliki, które aplikacja kliencka ignoruje podczas przesyłania na serwer. Kod zawiera tylko zależności w stosunku do warstwy domenowej i zarządza logiką biznesową w zakresie samej aplikacji, niskopoziomowo.
- Infrastruktura i serwisy (ang. Infrastructure and services) – pliki w tej części są odpowiedzialne za część roboczą aplikacji, tj. wysyłanie i odbieranie danych z serwera. W tym celu wykorzystuje wcześniej zdefiniowane zachowania aplikacji (z warstwy Aplikacji) oraz steruje nimi zgodnie z wejściem, tj. logiką zawartą w obsłudze CLI.
- CLI (ang. Command Line Interface, pol. Interfejs Linii Poleceń) – warstwa ta, jako najwyższa, importuje do swoich plików wszystkie pozostałe, tj. przypisuje rzeczywiste dane do warstwy domenowej, odczytuje dane zapisane w plikach z „Aplikacji" oraz wywołuje metody zawarte w infrastrukturze do wykonania odpowiedniego zapytania na serwerze. Wejściem jest komenda wywołana przez użytkownika (z parametrami, jeżeli takowe się znajdują). Główną biblioteką jest tutaj „Click".

Dodatkowymi plikami, nie zawartymi w żadnej warstwie są:

- *Client.py* – główny plik startowy z konfiguracją biblioteki „Click” odpowiedzialnej za komendy przeznaczone dla użytkowników
- *Constants.py* – zawiera wszystkie stałe używane w tym projekcie, np. lista plików która jako zawsze będzie dołączona to pliku *.ai-ignore*
- *Log.py* – konfiguracja biblioteki „logging” odpowiedzialnej za wyświetlanie informacji w konsoli oraz zapisu ich do pliku

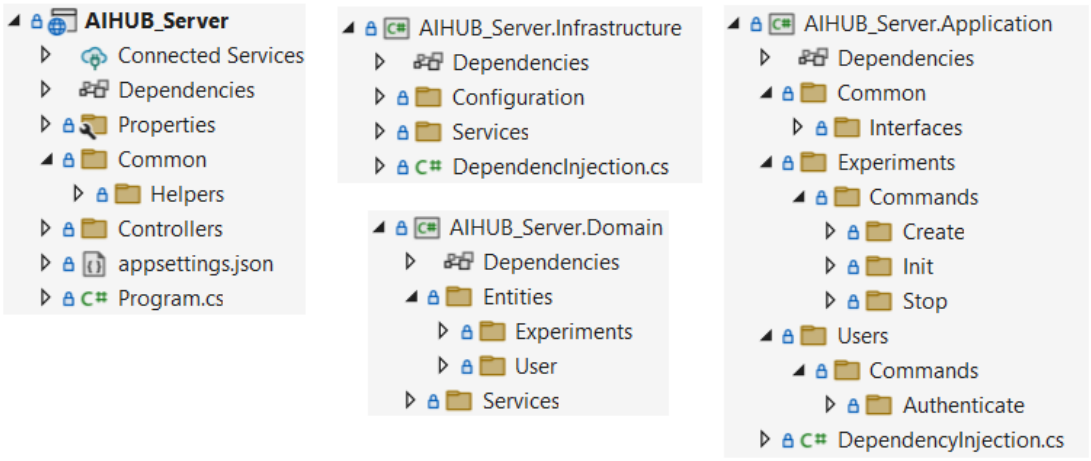
6.2 Architektura serwera



Rys. 8 – Schemat Clean Architecture z oficjalnej dokumentacji Microsoft [źródło: https://learn.microsoft.com/pl-pl/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures]

Clean Architecture to podejście do projektowania aplikacji oparte na separacji warstw i ich izolacji od siebie nawzajem [18]. Celem tej architektury jest zapewnienie elastyczności i łatwości w utrzymaniu aplikacji, poprzez odseparowanie warstw logicznych od warstw technologicznych. W Clean Architecture, warstwy są ułożone według kryterium "od wewnątrz na zewnątrz", co oznacza, że wewnętrzne są

bardziej abstrakcyjne i niezależne od konkretnych technologii, a zewnętrzne są bardziej konkretne i zależne od technologii.



Rys. 9 – Podział plików zgodny z Clean Architecture w plikach serwera (ASP.NET).

Pliki zostały podzielone na projekty (rys. 9), podobnie jak i w architekturze klienta, na:

- Główny projekt – posiada główny plik startowy serwera (*Program.cs*) oraz kontrolery (odpowiednik komend w kliencie). Dodatkowo znajdują się pliki pomocnicze, tzw. „Middleware” – kod, który wstrzyknięty podczas obsługi przychodzących żądań HTTPS obsługuje w tym przypadku token
- Infrastrukturę – zawiera kod, który obsługuje logikę poza terenem aplikacji serwera, np. tworzenie folderów i zapis plików czy odczyt danych użytkowników podczas logowania
- Domenę – jest to wysokopoziomowa logika biznesowa zawarta w kodzie, czyli wszystkie modele z wraz odpowiadającym im opisem w postaci zmiennych
- Aplikację – zawiera pliki obsługujące sam serwer i jego żądania, głównie została użyta tutaj biblioteka „MediatR”

6.2.1 Architektura wzorca MediatR

Serwer wykorzystuje bibliotekę MediatR, która pozwala na oddzielenie logiki biznesowej od aplikacji. Poniżej przedstawiono opis kodów obsługujących tę część architektury, na bazie kontrolera dot. logowania. Struktura ta jest powtarzalna też dla pozostałych kontrolerów.

W listingu 7 znajduje się klasa *UserController*, która dziedziczy po *ApiControllerBase* (znajdują się tam mniej istotne ustawienia kontrolera, np. bazowa część adresu pod którym będą dostępne zasoby). Linia 3. wskazuje na nazwę końcowej części endpointu. Linia 4. definiuje co będzie zwracane w przypadku wykonania zapytania z statusem 200 (OK). Następna linia wskazuje na to, co przyjmuje endpoint (klasę *UsersAuthenticateCommand*) oraz zwraca typ *IActionResult*.

Listing 8 pokazuje czym jest wejście do kontrolera, czyli dwa pola wymagane (atrybut *[Required]* w liniach 3. i 6.) tekstowe z nazwą użytkownika i hasłem, pobierane z żądania HTTPS (z jego ciała). Klasa ta dziedziczy interfejs z biblioteki MediatR, czyli *IRequest<out T>*, gdzie *T* to obiekt, który chcemy zwrócić w wyniku działania operacji.

Kolejny listing, 9, pokazuje kod klasy *UsersAuthenticateCommandResult*, czyli wynik wykonanej operacji. Mapuje on w swoim konstrukturze (linie od 9. do 16.) dane z przekazanego modelu domenowego *User* oraz tokenu na odpowiednie pola tekstowe (linie od 3. do 7.).

Ostatni listing w tej sekcji dotyczy klasy *UsersAuthenticateCommandHandler*, która dziedziczy interfejs biblioteki MediatR *IRequestHandler<in TRequest, TResponse>*, gdzie są to odpowiednio wejście i wyjście w kontrolerze. W konstruktorze tej klasy (linie 5. do 8.) został wstrzyknięty serwis (z warstwy infrastruktury) odpowiedzialny za logikę i sprawdzanie danych, które przyszły w żądaniu HTTPS; jest on wywoływany w linii 12. Kolejno (linie 14. i 17.) zwracają odpowiedni obiekt w zależności czy operację wykonano pomyślnie, który jest powiązany z wcześniej wspomnianym *IActionResult*.

```
1. public class UsersController : ApiControllerBase
2. {
3.     [HttpPost("Authenticate")]
4.     [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(UsersAuthenticateCommandResult))]
5.     public async Task<IActionResult> Authenticate(UsersAuthenticateCommand command)
6.     {
7.         return await Mediator.Send(command);
8.     }
9. }
```

Listing 7 – Kod kontrolera w języku C#.

```
1. public class UsersAuthenticateCommand : IRequest<IActionResult>
2. {
3.     [Required]
4.     public string Username { get; set; }
5.
6.     [Required]
7.     public string Password { get; set; }
8. }
```

Listing 8 – Kod klasy *UsersAuthenticateCommand* z zaimplementowanym MediatR.

```
1. public class UsersAuthenticateCommandResult
2. {
3.     public int Id { get; set; }
4.     public string FirstName { get; set; }
5.     public string LastName { get; set; }
6.     public string Username { get; set; }
7.     public string Token { get; set; }
8.
9.     public UsersAuthenticateCommandResult(User user, string token)
10.    {
11.        Id = user.Id;
12.        FirstName = user.FirstName;
13.        LastName = user.LastName;
14.        Username = user.Username;
15.        Token = token;
16.    }
17. }
```

Listing 9 – Kod klasy *UsersAuthenticateCommandResult*.

```
1. public class UsersAuthenticateCommandHandler : IRequestHandler<UsersAuthenticateCommand, IActionResult>
2. {
3.     private IUserService _userService;
4.
5.     public UsersAuthenticateCommandHandler(IUserService userService)
6.     {
7.         _userService = userService;
8.     }
9.
10.    public async Task<IActionResult> Handle(UsersAuthenticateCommand command, CancellationToken cancellationToken)
11.    {
12.        UsersAuthenticateCommandResult response = _userService.Authenticate(command);
13.
14.        if (response == null)
15.            return new BadRequestObjectResult(new { result = "Username or password is incorrect" });
16.
17.        return new OkObjectResult(response);
18.    }
19. }
```

Listing 10 – Kod klasy *UsersAuthenticateCommandHandler* z zaimplementowanym MediatR.

6.3 Pliki wspólne – aplikacja kliencka

Ten rozdział opisuje pliki w aplikacji klienckiej, które są używane wielokrotnie w całym projekcie w kodzie.

6.3.1 Warstwa domenowa

W tej warstwie znajduje się tylko jeden plik, który jest używany wszędzie jako baza pod pozostałe – *base.py*. Jest to szkielet, który definiuje dwie metody używane w bibliotece Marshmallow. W linii 6. listingu 11 istnieje metoda, która przetwarza wytworzoną domenę na postać JSON (lub inaczej – słownik). Analogicznie, w linii 9. znajduje się metoda, która przerabia postać słownikową na obiekt domenowy biblioteki.

```
1.  class BaseModel(object):
2.      """
3.     Base for all model classes
4.     """
5.
6.      def to_dict(self):
7.          return self.schema.dump(self)
8.
9.      @classmethod
10.     def from_dict(self, dct):
11.         return self.schema.load(dct)
```

Listing 11 – Kod pliku base.py z klasą BaseModel w aplikacji klienckiej.

6.3.2 Warstwa aplikacji

W tej warstwie znajdują się 3 pliki:

- *auth_file_manager.py*
- *experiment_file_manager.py*
- *ignore_file_manager.py*

Są one odpowiedzialne kolejno za pliki: *.ai-auth*, *.ai-exp*, *.ai-ignore*, które powstają podczas działania aplikacji.

Kod odpowiedzialny za plik *.ai-auth* znajduje się w listingu 12. W liniach od 15. do 17. jest zamieszczona metoda, która zapisuje otrzymany token do pliku. Od 20. do 27. linii znajduje się funkcja *get_auth*, która z kolei odczytuje z pliku token i go zwraca. Metoda *exists_auth*, która umieszczona jest pomiędzy liniami 30. i 34. sprawdza, czy plik istnieje – jeżeli tak, to znaczy że użytkownik już musiał się wcześniej zalogować, jeżeli nie – aplikacja jest wyłączana wraz z odpowiednią adnotacją dla użytkownika (linia 33.).

```
1. import json
2. import os
3. import click
4. import sys
5. from aihub.domain.auth import AuthSettings
6.
7. class AuthFileManager(object):
8.     """
9.    Manages .ai-auth file in the current directory
10.    """
11.
12.     AUTH_FILE_PATH = os.path.join(os.getcwd() + "/.ai-auth")
13.
14.     @classmethod
15.     def set_auth(self, auth_settings):
16.         with open(self.AUTH_FILE_PATH, "w") as file:
17.             file.write(json.dumps(auth_settings.to_dict()))
18.
19.     @classmethod
20.     def get_auth(self):
21.         if not os.path.isfile(self.AUTH_FILE_PATH):
22.             raise Exception("Missing .ai-auth file")
23.
24.         with open(self.AUTH_FILE_PATH, "r") as file:
25.             auth_settings = json.loads(file.read())
26.
27.         return AuthSettings.from_dict(auth_settings)
28.
29.     @classmethod
30.     def exists_auth(self):
31.         is_init_already = os.path.isfile(self.AUTH_FILE_PATH)
32.         if is_init_already is False:
33.             click.echo(click.style('[INFO] Please login first.', fg='yellow'))
34.             sys.exit()
```

Listing 12 – Kod *auth_file_manager.py* odpowiedzialny za plik *.ai-auth*.

Kod zawarty w listingu 13 tworzy plik *.ai-exp* z nazwą projektu, odczytuje ją i sprawdza czy plik istnieje (kolejno metody: *set_config*, *get_config*, *exist_config*). Nazwa jest uwzględniana jako punkt odniesienia w kolejnych procesach (np. w serwerze do utworzenia ścieżki projektu).

```
1. import json
2. import os
3. import sys
4. import click
5. from aihub.domain.experiment_config import ExperimentConfig
6. from log import logger as aihub_logger
7.
8. class ExperimentFileManager(object):
9.     """
10.    Manages .ai-exp file in the current directory
11.    """
12.
13.     CONFIG_FILE_PATH = os.path.join(os.getcwd() + "/.ai-exp")
14.
15.     @classmethod
16.     def set_config(self, experiment_config):
17.         with open(self.CONFIG_FILE_PATH, "w") as config_file:
18.             config_file.write(json.dumps(experiment_config.to_dict()))
19.
20.     @classmethod
21.     def get_config(self):
22.         if not os.path.isfile(self.CONFIG_FILE_PATH):
23.             raise Exception("[ERROR] Missing .ai-exp file, run init first")
24.
25.         with open(self.CONFIG_FILE_PATH, "r") as config_file:
26.             experiment_config = json.loads(config_file.read())
27.
28.         return ExperimentConfig.from_dict(experiment_config)
29.
30.     @classmethod
31.     def exist_config(self):
32.         is_init_already = os.path.isfile(self.CONFIG_FILE_PATH)
33.         if is_init_already is True:
34.             click.echo(click.style(
35. '[INFO] There is already initialized AIHUB project in that folder! Init skipped.' +
36.  'You can delete manually .ai-exp file.', fg='yellow'))
37.             sys.exit()
```

Listing 13 – Kod *experiment_file_manager.py* odpowiedzialny za plik *.ai-exp*.

Kod umieszczony w listingu 14, który dotyczy pliku *.ai-ignore* (odpowiedzialny za listę ścieżek, które są ignorowane podczas przesyłania plików na serwer). Listing ten różni się od poprzednich, mianowicie znajduje się w nim metoda *init* (linie od 14. do 23.) która tworzy plik fizycznie wraz z standardową zawartością, która znajduje się w osobnym kodzie (*constants.py*). Od linii 35. do 69. znajduje się kod funkcji *get_lists*, który tworzy listę ignorowanych plików oraz ważnych, których nie należy ignorować (należy je poprzedzić znakiem „!”, mogą to być pliki wcześniej ignorowane).

```python
import os

from constants import DEFAULT_IGNORE_LIST
from log import logger as aihub_logger

class IgnoreFileManager(object):
    """
   Manages .ai-ignore file in current directory
   """

    IGNORE_FILE_PATH = os.path.join(os.getcwd() + "/.ai-ignore")

    @classmethod
    def init(self):
        if os.path.isfile(self.IGNORE_FILE_PATH):
            aihub_logger.debug(f"AIHUB ignore file is already at {self.IGNORE_FILE_PATH}")
            return

        aihub_logger.debug(f"Setting default ignore list in file at {self.IGNORE_FILE_PATH}")

        # This creates file with default ones
        with open(self.IGNORE_FILE_PATH, "w") as ignore_file:
            ignore_file.write(DEFAULT_IGNORE_LIST)

    def slash_prefix_trim(path):
        """
       Remove '/' if it is 1st character in path.
       PurePath wants path as absolute path (for match)
       """
        if path.startswith('/'):
            return path[1:]
        return path

    @classmethod
    def get_lists(self, file_path=None):
        if file_path != None:
            ignore_file_path = file_path
        else:
            ignore_file_path = self.IGNORE_FILE_PATH

        # Safety if somebody gave not a file
        if not os.path.isfile(ignore_file_path):
            return([], [])

        ignore_list = []
        whitelist = []

        with open(ignore_file_path, "r") as ignore_file:
            for line in ignore_file:
                line = line.strip() # Remove all spaces etc.

                if not line or line.startswith('#'):
                    continue

                if line.startswith('!'):
                    line = line[1:]
                    whitelist.append(self.slash_prefix_trim(line))
                    continue

                # To allow escaping file names that start with !, # or \
                # remove the escaping \
                if line.startswith('\\'):
                    line = line[1:]

                ignore_list.append(self.slash_prefix_trim(line))

        aihub_logger.debug(f"IGNORE_LIST: {ignore_list}")
        aihub_logger.debug(f"WHITELIST: {whitelist}")
        return (ignore_list, whitelist)


```

Listing 14 – Kod *ignore_file_manager.py* odpowiedzialny za plik *.ai-ignore*.

6.3.3 Warstwa infrastruktury

W warstwie infrastruktury znajdują się pliki odpowiedzialne tylko i wyłącznie za komunikację z serwerem. Pośród nich znajduje się plik *base.py*, który opakowuje w klasę *HttpClient* gotową do importu dla pozostałych funkcji, bibliotekę requests (listing 15, linie od 30. do 39.). Danymi wchodzącymi do tej klasy są dokładnie takie same parametry (linie od 8 do 16.), jak w metodzie *request* zaznaczonej wcześniej. Korzyścią takiego rozwiązania jest możliwość dodania logowania danych (np. linie 19., 20., 41., 42.) oraz generowanie informacji zwrotnej na podstawie otrzymanego wyjątku (linie od 44. do 48.).

```
1. class HttpClient(object):
2.     """
3.    Base client for all HTTP operations
4.    """
5.     def __init__(self):
6.         self.base_url = client.aihub_host
7.
8.     def request(self,
9.                 method,
10.                url,
11.                params=None,
12.                data=None,
13.                files=None,
14.                json=None,
15.                timeout=5):
16.         request_url = self.base_url + url
17.
18.         try:
19.             aihub_logger.debug("Request URL:")
20.             aihub_logger.debug(request_url)
21.
22.             access_token_header = ""
23.             if url != "/users/authenticate":
24.                 auth_settings = AuthFileManager.get_auth()
25.                 access_token = auth_settings['access_token']
26.                 access_token_header = {'Authorization': 'Bearer {}'.format(access_token)}
27.             else:
28.                 access_token_header = ""
29.
30.             response = requests.request(method,
31.                                         request_url,
32.                                         params=params,
33.                                         data=data,
34.                                         json=json,
35.                                         files=files,
36.                                         headers=access_token_header,
37.                                         timeout=timeout,
38.                                         verify='cert.pem',
39.                                         )
40.
41.             aihub_logger.debug("Response status code:")
42.             aihub_logger.debug(response.status_code)
43.
44.         except requests.exceptions.ConnectionError:
45.             aihub_logger.info("[ERROR] Cannot connect to the AIHUB server. Check your internet connection.")
46.             sys.exit()
47.
48.         except requests.exceptions.Timeout:
49.             aihub_logger.info("[ERROR] Connection to AIHUB server timed out. Please retry or check your internet connection.")
50.             sys.exit()
51.
52.         self.check_response_status(response)
53.         return response
```

Listing 15 – Kod pliku *base.py* – bazowy klient HTTP dla pozostałych.

Przykładem zastosowania klasy są kolejne klienty, np. *AuthClient* wymieniony w listingu 16 – służący do obsługi żądań dot. logowania użytkownika do aplikacji (linia 16.). Jak można zauważyć w linii 3., powstał on na bazie *HttpClient*. Podobnym klientem jest *ExperimentClient* (nie wymieniony z powodu ilości kodu), który posiada analogicznie metody dot. inicjalizacji projektu na serwerze, zastopowania nauki oraz przesyłania plików na serwer (z dodatkową logiką dot. pliku *.ai-ignore*).

```
1. from aihub.infrastructureservices.base import HttpClient
2.
3. class AuthClient(HttpClient):
4.     """
5.    Client to interact with Experiment api
6.    """
7.     def __init__(self):
8.         self.url = "/users/authenticate"
9.         super().__init__() # Call __init__() on HttpClient
10.
11.     def authenticate(self, login_data):
12.         """
13.        Send request with user, password, get 'token'
14.        """
15.
16.         response = self.request('POST', self.url, json=login_data.to_dict())
17.
18.         return response.json()['token']
```

Listing 16 – Kod klasy *AuthClient* z warstwy infrastuktury.

6.4 Pliki wspólne – serwer

Pliki wspólne w kodzie serwera dotyczą autoryzacji. Każdy z kontrolerów (poza tym, który umożliwia logowanie) posiada nad sobą atrybut *[Authorize]*, przykład pokazano w listingu 17.

```
1. [Authorize]
2. [HttpPost("Init")]
3. [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(ExperimentsInitCommandResult))]
4. [ProducesResponseType(StatusCodes.Status400BadRequest, Type = typeof(ExperimentsInitCommandResult))]
5. public async Task<IActionResult> ExperimentsInit(ExperimentsInitCommand query)
6. {
7.     return await Mediator.Send(query);
8. }
```

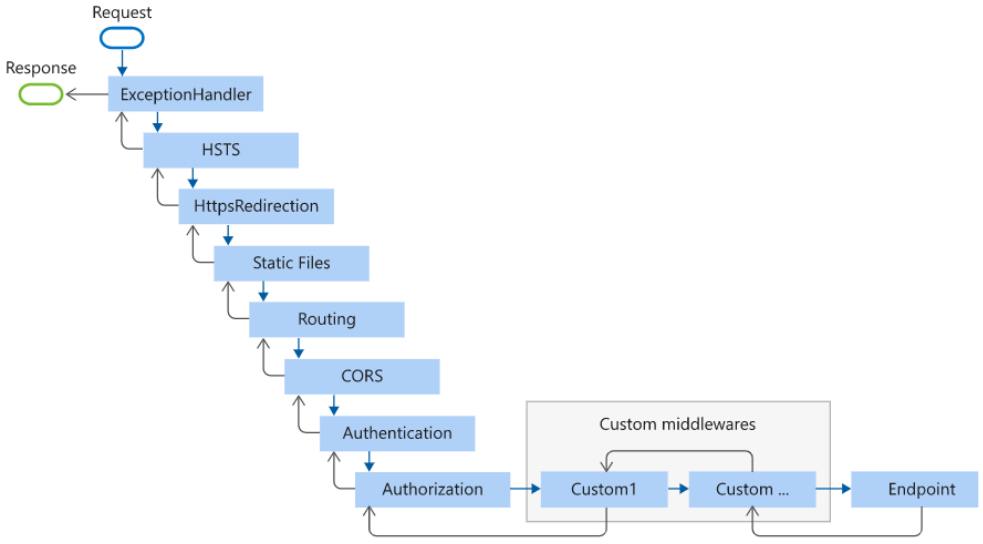
Listing 17 – Przykład kontrolera zabezpieczonego autoryzacją.

Kod wyżej omawianego atrybutu znajduje się w listingu 18. Autoryzacja odbywa się za pomocą metody *OnAuthorization* (linia 4.) która sprawdza, czy do bieżącego żądania dołączony jest uwierzytelniony użytkownik (linia 6.). W przypadku udanej autoryzacji nie jest podejmowana żadna akcja, a żądanie jest przekazywane do kontrolera; jeśli się nie powiedzie, zwracana jest odpowiedź o kodzie 401 - nieautoryzowany dostęp.

```
1. [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
2. public class AuthorizeAttribute : Attribute, IAuthorizationFilter
3. {
4.     public void OnAuthorization(AuthorizationFilterContext context)
5.     {
6.         var user = (User)context.HttpContext.Items["User"];
7.         if (user == null)
8.         {
9.             // Not logged in
10.            context.Result = new JsonResult(new { message = "Unauthorized" })
11.            {
12.              StatusCode = StatusCodes.Status401Unauthorized
13.            };
14.        }
15.    }
16. }
```

Listing 18 – Kod opisujący atrybut *[Authorize]* stosowany do zabezpieczenia kontrolerów.

W ASP.NET, oprogramowanie pośrednie ma swoją strukturę w postaci kolejności wywoływania, pokazaną na rys. 10.



Rys. 10 – Struktura działania oprogramowania pośredniczącego (ang. middleware) w ASP.NET [źródło: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-6.0]

W implementacji użytej na serwerze przechwytywany jest krok autoryzacji i wywoływany jest kod z listingu 18. Dodanie prawidłowo zalogowanego użytkownika do żądania HTTPS (linia 6. listingu 18) powstaje poprzez dodane niestandardowe oprogramowanie (na rys. 10 jako „Custom middlewares”), którego kod znajduje się w listingu 19. W liniach od 28. do 36. znajduje się metoda *Invoke*, która sprawdza, czy w żądaniu w nagłówkach jest także JWT Token. Jeżeli on istnieje, wywoływana jest metoda z serwisu *UserService* (listing 20), która po odkodowaniu tokenu (linie od 5. do 16.) przywiązuje użytkownika do wcześniej omawianego HttpContextu (linia 21.).

Analogicznie stworzono przechwytywanie tokenu dla SignalR, który przesyłany jest nie jako nagłówki w żądaniu, lecz jako parametr w adresie (listing 19. linia 15.). Zwracany jest także wynik w postaci flagi prawda/fałsz w linii 25. SignalR działa na Websockets, nie na żądaniach HTTPS, dlatego trzeba ręcznie rozłączyć użytkownika w przypadku nie przypisania tokenu do żadnego użytkownika (nie jest możliwe użycie listingu 18), na podstawie przekazanej flagi.

```
1. public class JwtMiddleware
2. {
3.     private readonly RequestDelegate _next;
4.
5.     public JwtMiddleware(RequestDelegate next)
6.     {
7.         _next = next;
8.     }
9.
10.    public static bool InvokeInSignalR(HttpContext context, IUserService userService)
11.    {
12.        bool response = false; // At default make it false.
13.        //If everythings is right, userService will return true
14.
15.        var token = context.Request.Query["access_token"];
16.
17.        var path = context.Request.Path;
18.        if (!string.IsNullOrEmpty(token) &&
19.            (path.StartsWithSegments("/streamcmd")))
20.        {
21.            // Moved to userService, otherwise this method wouldn't be static.
22.            response = userService.AttachUserToContext(context, userService, token);
23.        }
24.
25.        return response;
26.    }
27.
28.    public async Task Invoke(HttpContext context, IUserService userService)
29.    {
30.        var token = context.Request.Headers["Authorization"].FirstOrDefault()?.Split(" ").Last();
31.
32.        if (token != null)
33.            userService.AttachUserToContext(context, userService, token);
34.
35.        await _next(context);
36.    }
37. }
```

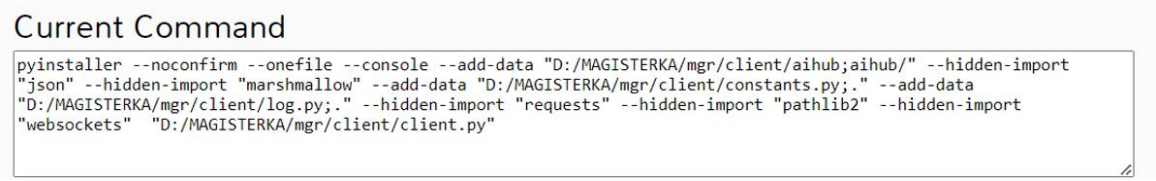
Listing 19 – Kod klasy *JwtMiddleware*.

```
1. public bool AttachUserToContext(HttpContext context, IUserService userService, string token)
2. {
3.     try
4.     {
5.         var tokenHandler = new JwtSecurityTokenHandler();
6.         var key = Encoding.ASCII.GetBytes(_authSettings.Secret);
7.         tokenHandler.ValidateToken(token, new TokenValidationParameters
8.         {
9.             ValidateIssuerSigningKey = true,
10.            IssuerSigningKey = new SymmetricSecurityKey(key),
11.            ValidateIssuer = false,
12.            ValidateAudience = false,
13.            // set clockskew to zero so tokens expire exactly at token expiration time (instead of 5 minutes later)
14.            ClockSkew = TimeSpan.Zero
15.        }, out SecurityToken validatedToken);
16.
17.        var jwtToken = (JwtSecurityToken)validatedToken;
18.        var userId = int.Parse(jwtToken.Claims.First(x => x.Type == "id").Value);
19.
20.        // attach user to context on successful jwt validation
21.        context.Items["User"] = userService.GetById(userId);
22.
23.        return true;
24.    }
25.    catch
26.    {
27.        // do nothing if jwt validation fails
28.        // user is not attached to context so request won't have access to secure routes
29.        return false; // for SignalR
30.    }
31. }
```

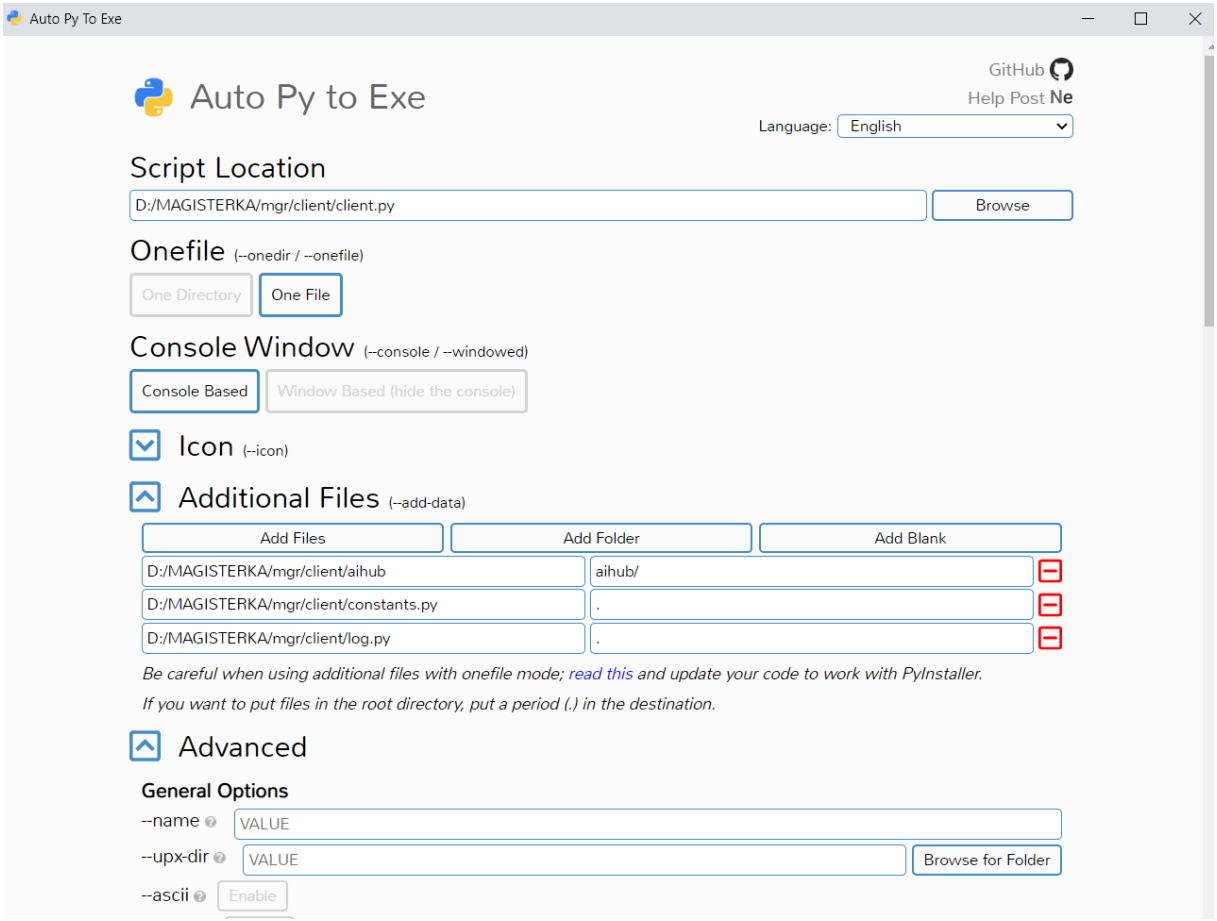
Listing 20 – Kod metody *AttachUserToContext* z serwisu *UserServices*.

6.5 Aplikacja kliencka w postaci pliku wykonywalnego

Programy napisane w języku Python uruchamiane są poprzez zainstalowany na urządzeniu interpreter. W celu możliwości włączenia aplikacji klienckiej na każdym komputerze z systemem Windows, postanowiono skompilować kod do pliku wykonywalnego (z rozszerzeniem *EXE*) – użytkownik nie potrzebuje wtedy posiadać zainstalowanego wcześniej wspomnianego środowiska. W tym celu wykorzystano narzędzie *PyInstaller*, które pozwala otrzymać wyżej wymienioną postać. Dodatkowo użyto także program *Auto Py To Exe* (rys. 12), który pozwala w łatwy sposób wygenerować komendę gotową do użycia (rys. 11), ponieważ jest ona skomplikowana i wymaga wielu parametrów (np. dodatkowe biblioteki, które należy użyć).



Rys. 11 – Wygenerowana komenda z programu *Auto Py To Exe*.



Rys. 12 – Program *Auto Py to Exe*.

6.6 Komendy

W tym rozdziale zostały opisane wszystkie komendy występujące w aplikacji (w relacji klient-serwer), które są widoczne w konsoli po uruchomieniu aplikacji bez żadnych parametrów, lub po wpisaniu *--help*. Opcje dostępne dla użytkownika to:

- *login* – logowanie użytkownika
- *init* – inicjalizowanie projektu w aktualnie uruchomionym folderze
- *upload* – przesyłanie na serwer plików, które nie są ignorowane (względem pliku *.ai-ignore*)
- *run* – uruchamianie wybranego pliku w języku Python do nauki
- *stop* – przerwanie aktualnie uruchomionego uczenia na serwerze
- *logout* – wylogowywanie użytkownika

6.6.1 Komenda „login”

Komenda *login* jest pierwszą, którą użytkownik powinien wpisać podczas obsługi aplikacji. W przeciwnym razie, przy użyciu dowolnej innej opcji, otrzyma komunikat by najpierw się zalogować. Nieuprawiony użytkownik nie może mieć dostępu do zasobów serwera; jednocześnie pozwala to na kontrolowanie kto ma dostęp. Przykład w jaki sposób wygląda komenda pokazano na rys. 13. Klient powinien wpisać ją w formacie: *login użytkownik hasło*, w przeciwnym przypadku otrzyma informację o niewłaściwej ilości podanych parametrów.

```
Usage: client.exe login [OPTIONS] USER PASSWORD

  Allows user to login with his username and password.

Options:
  --help  Show this message and exit.
```

Rys. 13 – Opis komendy *login* podczas próby użycia aplikacji.

Po wywołaniu komendy, biblioteka Click obsługuje ją i następnie uruchamiany jest kod z listingu 21. Na początku inicjalizuje on klienta do komunikacji z serwerem, kolejno tworzy model domenowy i wysyłane jest żądanie HTTPS (linie 8. – 10.). Kolejno tworzony jest plik *.ai-exp*, który przechowuje w sobie model utworzony w linii 12. – wraz z otrzymanym JWT tokenem od serwera. Kolejno jest on pobierany (z pliku) i sprawdza się, czy nie jest pusty – jeżeli istnieje, to znaczy że użytkownik został zalogowany poprawnie na serwerze.

Kod obsługujący logowanie na serwerze jest już wcześniej opisany, w dziale 6.2.1. (architektura wzorca MediatR). W kolejnych opisach komend będzie znajdować się tylko listing metody z logiką wywoływaną przez bibliotekę MediatR.

```python
@click.command()
@click.argument('user', nargs=1)
@click.argument('password', nargs=1)
def login(user, password):
    """
   Allows user to login with his username and password.
    """
     auth_client = AuthClient()
     login_data = LoginData(user, password)
     access_token = auth_client.authenticate(login_data)

     auth_settings = AuthSettings(access_token=access_token)
     AuthFileManager.set_auth(auth_settings) # Create .ai-exp

     auth_settings = AuthFileManager.get_auth()
     access_token = auth_settings['access_token']

     if access_token:
         aihub_logger.info("[INFO] User sucessfully logged in.")

     aihub_logger.debug(f"access_token: {access_token}")
```

Listing 21 – Kod z pliku *cli/auth.py* w aplikacji klienckiej odpowiedzialny za logowanie.

6.6.2 Komenda „init”

Komenda *init* pozwala na inicjalizację nowego projektu w aktualnym folderze, w jakim znajduje się uruchomiona aplikacja. Do poprawnego zadziałania tej opcji należy podać nazwę projektu (rys. 14). Po jej uruchomieniu aplikacja zapyta się ponownie, czy użytkownik jest pewien podanej nazwy i czy chce ją zmienić lub zostawić (rys. 15); w przypadku gdy projekt już został wcześniej zainicjalizowany, zostanie pokazana informacja, że plik *.ai-exp* już istnieje. (rys. 16).

```
Usage: client.exe init [OPTIONS] PROJECT_NAME

  Initialize new project. After this you can run other commands like "upload"
  or "run".

Options:
  --help  Show this message and exit.
```

Rys. 14 – Komenda *init* aplikacji uruchomiona z opcją *--help*.

```
D:\MAGISTERKA\mgr\client\dist>client.exe init nowy_projekt
Press ENTER to use project name "nowy_projekt" or enter a different name: projekt_mnist
[INFO] Initial folder created on the server.
```

Rys. 15 – Uruchomienie komendy *init* po raz pierwszy.

```
D:\MAGISTERKA\mgr\client\dist>client.exe init nowy_projekt
[INFO] There is already initialized AIHUB project in that folder! Init skipped. You can delete manually .ai-exp file.
```

Rys. 16 – komenda *init* w przypadku, gdy projekt już wcześniej był zainicjalizowany.

Kod wykonywany po uruchomieniu inicjalizacji znajduje się w listingu 22. Na początku jest sprawdzane, czy użytkownik jest zautoryzowany (posiada plik *.ai-auth*). Kolejno jest sprawdzane, czy w folderze istnieje już plik *.ai-exp* z zainicjalizowanym projektem, jeżeli tak – wyświetlana jest informacja jak na rys. 16, w przeciwnym przypadku użytkownik jest pytany jeszcze raz o nazwę projektu. Kolejno podana nazwa jest zapisywana w pliku *.ai-exp* z dodanym losowym ciągiem znaków, która następnie jest wysyłana na serwer.

```
1. @click.command()
2. @click.argument('project_name', nargs=1)
3. def init(project_name):
4.     """
5.    Initialize new project.
6.    After this you can run other commands like "upload" or "run".
7.    """
8.     AuthFileManager.exists_auth()
9.
10.    # Check if .ai-exp already exists, if yes - show error and sys.exit()
11.    ExperimentFileManager.exist_config()
12.
13.    # Get project name from user
14.    name = click.prompt('Press ENTER to use project name "%s" or enter a different name' % project_name, default=project_name,
    show_default=False)
15.    project_name = name.strip() or project_name
16.
17.    # Create .ai-exp JSON with proj name
18.    experiment_config = ExperimentConfig(name=project_name + "_" + str(uuid.uuid4()))
19.    ExperimentFileManager.set_config(experiment_config)
20.
21.    # Create .ai-ignore
22.    IgnoreFileManager.init()
23.
24.    # Sending request to create dictionary on server with folder
25.    try:
26.        init_response = ExperimentClient().init(experiment_config)
27.        aihub_logger.info(f"[INFO] {init_response}")
28.    except Exception as e:
29.        aihub_logger.error("[ERROR] %s", e)
```

Listing 22 – Kod w aplikacji klienckiej dot. komendy *init*.

Działanie na serwerze po otrzymaniu żądana HTTPS od klienta zostało pokazane na listingu 23 – jest to kod z metody *Handle* z biblioteki MediatR. Tworzony jest folder na serwerze (poprzez linię 3., gdzie za pomocą Dependency Injection został dodany serwis dot. tej operacji) – w nim są przechowywane później pliki (po uruchomieniu komendy *upload*). Wynik jest zwracany w postaci flagi, wg. której można ustawić komunikat dla użytkownika i przypisać mu odpowiedni kod HTTPS.

```
1. public async Task<IActionResult> Handle(ExperimentsInitCommand command, CancellationToken cancellationToken)
2. {
3.     bool createdFolder = _projectFolderService.CreateProjectFolder(command);
4.
5.     var result = new ExperimentsInitCommandResult();
6.
7.     if (createdFolder == true)
8.     {
9.         result.Result = "Initial folder created on the server.";
10.    }
11.    if (createdFolder == false)
12.    {
13.        result.Result = "Folder with given name actually exists on the server.";
14.        return new BadRequestObjectResult(result);
15.    }
16.
17.    return new OkObjectResult(result);
18. }
```

Listing 23 – Logika serwera w zakresie inicjalizowania projektu.

6.6.3 Komenda „upload”

Komenda *upload* ma za zadanie przesłać wszystkie pliki w aktualnie uruchomionym projekcie, które jednocześnie nie są zawarte w pliku *.ai-ignore*. Podczas uruchamiania nie jest potrzebny żaden dodatkowy parametr, jak pokazano na rys. 17.

```
Usage: client.exe upload [OPTIONS]

  AIHUB will upload contents of the current directory to server.

Options:
  --help  Show this message and exit.
```

Rys. 17 – Komenda *upload* aplikacji uruchomiona z opcją *--help*.

Kod aplikacji klienckiej odpowiadający za tę sekcję przedstawiono na listingu 24. Na początku jest sprawdzana autoryzacja, kolejno odczytywana jest nazwa projektu, a na końcu jest klient (linia 12.), który ma za zadanie przekazać pliki na serwer (do wcześniej utworzonego folderu na nim). W tym kliencie jest zawarty listing 25. Wywołuje on w linii 6. funkcję, która zwraca listę plików i ich całkowity rozmiar (użyto standardowej biblioteki *os*). Jeżeli rozmiar jest większy dozwolonego na serwerze to użytkownik jest odpowiednio o tym informowany. Na końcu listingu 25, w liniach od 34. do 38. zawarte jest wykonanie żądania

HTTPS. Pliki, które są przesyłane na serwer nie w ciele (ang. body), lecz w postaci form (nagłówek multipart/form-data).

Logika, która jest wykonywania po odebraniu żądania HTTPS jest zawarta w listingu 26. Wywoływany jest serwis, który zapisuje pliki i zwraca flagę prawda/fałsz, która pozwala zdefiniować odpowiedni komunikat dla użytkownika oraz kod HTTP odpowiadający sukcesowi lub niepomyślnemu wykonaniu zapytania.

```
@click.command()
def upload():
    """
    AIHUB will upload contents of the current directory to server.
    """
    AuthFileManager.exists_auth()

    experiment_config = ExperimentFileManager.get_config()
    upload_config = UploadConfig(name=experiment_config['name'])

    try:
        upload_response = ExperimentClient().upload(upload_config)
        aihub_logger.info(f"[INFO] {upload_response}")
    except Exception as e:
        aihub_logger.error("[ERROR] %s", e)
```

Listing 24 – Kod w aplikacji klienckiej dot. komendy *upload*.

```
def upload(self, upload_config):
        """
       Upload files to a server
       """
        try:
            upload_files, total_file_size = get_files_in_current_directory(file_type='files')
        except OSError:
            sys.exit("Directory contains too many files to upload. If you have data files in the current directory, "
                     "please pack them and send to github and download in python scipt and remove them from here.\n")

        if total_file_size > self.MAX_UPLOAD_SIZE:
            sys.exit(("Code size too large to sync, please keep it under %s.\n"
            "If you have data files in the current directory, "
            "please pack them and send to github and download in python scipt and remove them from here.\n") %
    (human_friendly_size(self.MAX_UPLOAD_SIZE)))

        aihub_logger.info("Uploading files. Total upload size: %s",
                          human_friendly_size(total_file_size))
        aihub_logger.info("Uploading: %s files",
                          len(upload_files))

        files_only = [x[1] for x in upload_files]
        file_name_only = [x[0] for x in files_only]
        aihub_logger.info(f"List of files being uploaded: {file_name_only}")

        # Creating payload
        payload = upload_config.to_dict() # instance type, experiment_name etc; more in run.py or Module class in models
        aihub_logger.debug("payload: ")
        aihub_logger.debug(payload)

        aihub_logger.debug("upload_files: ")
        aihub_logger.debug(upload_files)

        aihub_logger.info("Uploading...")
        response = self.request("POST",
                                self.url_upload,
                                files=upload_files,
                                data=payload,
                                timeout=3600)

        return response.json()['result']
```

Listing 25 – Kod klienta *ExperimentClient*.

```
1. public async Task<IActionResult> Handle(ExperimentsUploadCommand command, CancellationToken cancellationToken)
2. {
3.     bool savedFiles = _projectFolderService.SaveFilesToFolder(command);
4.
5.     var result = new ExperimentsUploadCommandResult();
6.
7.     if (savedFiles)
8.     {
9.         result.Result = "Files correctly saved on the server.";
10.    }
11.    else
12.    {
13.        result.Result = "One of file is empty. Please fix your files.";
14.        return new BadRequestObjectResult(result);
15.    }
16.
17.    return new OkObjectResult(result);
18. }
```

Listing 26 – Logika serwera dot. przesyłania plików.

6.6.4 Komenda „run”

Komenda *run* odpowiada za uruchomienie wybranego pliku wraz ze zdefiniowaną ilością pamięci, jaką chcemy wykorzystać na serwerze (oznaczona jako parametry na rys. 18). Jest to najważniejsza sekcja w całym powstałym systemie. Wykorzystuje ona Websockets, a na serwerze bibliotekę SignalR w celu obsługi w czasie rzeczywistym. Dodatkowo istnieje możliwość uruchomienia treningu ponownie, za pomocą opcji *--run_again*.

```
Usage: client.exe run [OPTIONS] FILE HOW_MUCH_VRAM

  Runs given filename, depending if VRAM is avalaible. You can also set --run-
  again flag.

Options:
  --run_again
  --help       Show this message and exit.
```

Rys. 18 – Komenda *run* aplikacji uruchomiona z opcją *--help*.

Obsługa w kodzie po stronie klienta przebiega podobnie, jak i pozostałe komendy (listing 27). Odczytywany jest JWT token w celu autoryzacji oraz nazwa projektu. Kolejno, w linii 16. wywoływany jest *StreamCmdClient*, a w następnej linii jego metoda *run*, która wraz z podanymi przez użytkownika parametrami trafia na serwer do biblioteki SignalR.

```
1. @click.command()
2. @click.argument('file', nargs=1)
3. @click.argument('how_much_vram', nargs=1)
4. @click.option('--run_again', is_flag=True, default=False)
5. def run(file, how_much_vram, run_again):
6.     """
7.     Runs given filename, depending if VRAM is avalaible. You can also set --run-again flag.
8.     """
9.     AuthFileManager.exists_auth()
10.
11.     experiment_config = ExperimentFileManager.get_config()
12.     project_name = experiment_config['name']
13.
14.     try:
15.         auth_settings = AuthFileManager.get_auth()
16.         access_token = auth_settings['access_token']
17.
18.         signalr_client = StreamCmdClient(access_token)
19.         signalr_client.run(project_name, file, how_much_vram, run_again)
20.     except Exception as e:
21.         aihub_logger.error("Error: %s", e)
```

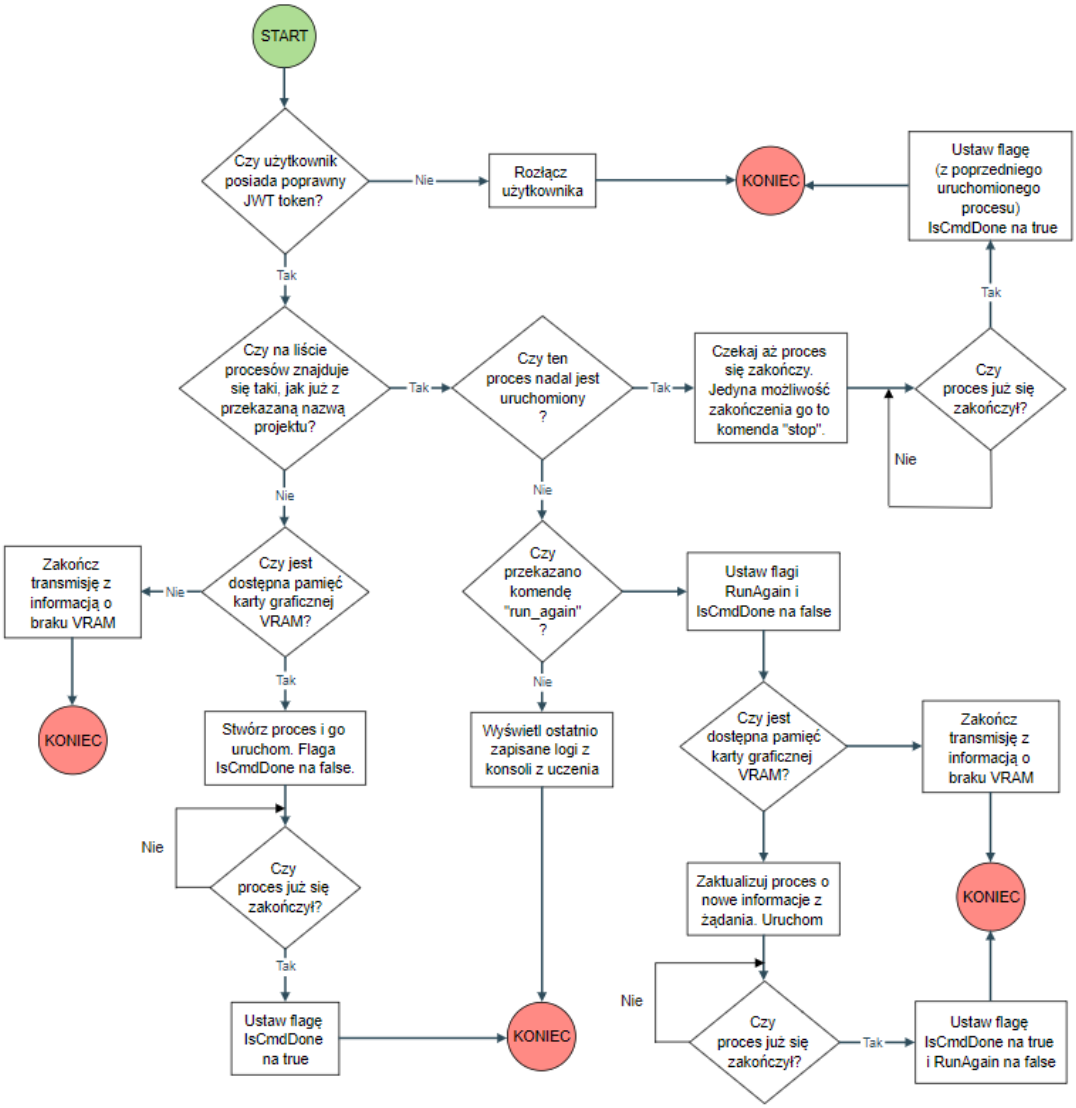
Listing 27 – Kod w aplikacji klienckiej dot. komendy *run*.

Listing 28 pokazuje kod *StreamCmdClient*, który wysyła WebSockets do serwera. Standard biblioteki SignalR definiuje pewną kolejność działań, które trzeba wykonać, pierwszym z nich jest ustalenie - tzw. negocjacja. Każdy *Hub* (czyli odpowiednik endpointu w serwerze) posiada wbudowany adres */negotiate*, pod który należy się najpierw udać w celu zdobycia *connectionId* (linia 94.), który następnie jest przekazywany wraz z każdą kolejną wiadomością do serwera (linia 96. i 34.). Kolejnym krokiem jest tzw. *handshake*, czyli porozumienie, w jakim standardzie będzie odbywać się transmisja danych (linie od 38. do 40. i wywołanie w 74.). Następnie są tworzone dwa asynchroniczne zadania (linie 76. i 77.), tj. nasłuch i ping, czyli systematyczne wysyłanie do serwera danych w celu podtrzymania ciągłości połączenia (w przypadku jej braku serwer uzna, że użytkownik się rozłączył). Ich metody znajdują się w liniach od 42. do 46. i od 48. do 70. Ping wysyła co 10 sekund wiadomość do serwera zgodnie z dokumentacją [19][20] z typem 6. Logika metody do nasłuchu polega na szukaniu, jakiego typu przychodzą wiadomości. Typ 3 to koniec transmisji od serwera, typ 7 to rozłączenie (następuje w przypadku nieważnego JWT tokenu). Typ 6 (tj. ping od serwera) jest pomijany w linii 66. natomiast pozostałe typy (tj. 3, czyli wiadomość) są odczytywane i wyświetlane użytkownikowi. SignalR przesyła informacje kończąc je znakiem "▲", „\x1e” lub „\u001e” (linie 20. i 53.), wg. których kod może je podzielić.

```python
import asyncio
import websockets
import requests
import json
import uuid
import ssl
import pathlib
import re
import client
from log import logger as aihub_logger

class StreamCmdClient():
    def __init__(self, access_token):
        self.base_url = client.aihub_host # without / at the end
        self.wss_url = self.base_url.replace("https://", "wss://")
        self.hub_url = "streamcmd"
        self.access_token = access_token

    def toSignalRMessage(self, data):
        return f'{json.dumps(data)}\u001e'

    async def connectToHub(self, connectionId, project_name, file_run, howMuchVRAM, run_again):
        ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
        ssl_context.load_verify_locations(
            pathlib.Path(__file__).with_name("cert.pem")
        )

        invocationId = uuid.uuid4()
        projectName = project_name
        command = file_run
        howMuchRAM = howMuchVRAM
        runAgain = run_again

        uri = f"""{self.wss_url}/{self.hub_url}?id={connectionId}
                  &projectName={project_name}&access_token={self.access_token}"""
        async with websockets.connect(uri, ssl=ssl_context) as websocket:

            async def handshake():
                await websocket.send(self.toSignalRMessage({"protocol": "json", "version": 1}))
                handshake_response = await websocket.recv()

            async def start_pinging():
                nonlocal _running
                while _running:
                    await asyncio.sleep(10)
                    await websocket.send(self.toSignalRMessage({"type": 6}))

            async def listen():
                nonlocal _running
                while _running:
                    get_response = await websocket.recv()

                    get_response_re = re.split('\▲|\x1e+', get_response)
                    for get_response_one in filter(None, get_response_re):
                        if get_response_one.find(f"\"type\":3") != -1:
                            print("[INFO] End of livestream. Press CTRL+C or wait a few seconds.")
                            _running = False
                            break

                        if get_response_one.find(f"\"type\":7") != -1:
                            print("[ERROR] Server disconnected you from resources.
                            Your access token is probably expired. Please logout and login again.")
                            _running = False
                            break

                        if get_response_one.find(f"\"type\":6") == -1: # ping
                            get_response_one_json = json.loads(get_response_one)
                            server_response = get_response_one_json["arguments"]
                            if(server_response[0] != None):
                                print(server_response[0]) # There should be only 1 in that arguments

            _running = True
            # First step - Handshake - get connectionId
            await handshake()

            listen_task = asyncio.create_task(listen())
            ping_task  = asyncio.create_task(start_pinging())

            # GETTINGS MESSAGES:
            message = {
                "type": 1, # 1 invocation, 4 stream
                "invocationId": f"{invocationId}",
                "target": "Stream", # Wywoływana funkcja
                "arguments": [ f"{projectName}", f"{command}", float(howMuchRAM), runAgain]
            }
            #print(message)
            await websocket.send(self.toSignalRMessage(message))

            await ping_task
            await listen_task

    def run(self, project_name, file, howMuchVRAM, runAgain):
        try:
            negotiation = requests.post(f"{self.base_url}/{self.hub_url}/negotiate?negotiateVersion=0", verify="cert.pem")
            # print(f"connectionId: {negotiation.json()['connectionId']}")
            asyncio.run(self.connectToHub(negotiation.json()['connectionId'], project_name, file, howMuchVRAM, runAgain))
        except Exception as e:
            aihub_logger.error("[ERROR] %s", e)
            print("[ERROR] Server error. Please try again later or follow the advice given to you earlier.")
```

Listing 28 – Opis *StreamCmdClient* w aplikacji klienckiej.

Logika serwera z racji swojej złożoności zostanie opisana poniższym grafem (rys. 19). Całość tej części serwera działa na *Hubie* z SignalR. W kodzie są zawarte 2 flagi sterujące (prawda / fałsz) o nazwach *IsCmdDone* oraz *RunAgain*, odpowiadające informacji, czy proces już się skończył oraz odpowiednik flagi *--run_again* z aplikacji klienckiej.



Rys. 19 – Graf przedstawiający logikę na serwerze z wykorzystaniem SignalR.

Podczas uczenia, jeżeli proces został prawidłowo uruchomiony, istnieje pętla która czeka dopóki się on nie skończy. W międzyczasie, wszystkie informacje oraz potencjalne błędy z konsoli są jednocześnie wysyłane do aplikacji klienckiej i zapisywane do pliku tekstowego (metoda *PythonOutputHadler*, linia 19. i 27. listingu 29).

```
1. var process = new Process()
2. {
3.     StartInfo = new ProcessStartInfo()
4.     {
5.         FileName = _serverSettings.Python310Path,
6.         Arguments = "-u " + fileToRun, // -u means unbuffered output
7.         RedirectStandardOutput = true,
8.         RedirectStandardInput = true,
9.         RedirectStandardError = true,
10.        UseShellExecute = false,
11.        WorkingDirectory = projectNameFolder,
12.    }
13. };
14.
15. process.OutputDataReceived += new DataReceivedEventHandler(
16.     (sendingProcess, outLine) =>
17.     {
18.         outputHandler(outLine.Data);
19.         PythonOutputHandler(sendingProcess, outLine, cmdOutputFile);
20.     }
21. );
22.
23. process.ErrorDataReceived += new DataReceivedEventHandler(
24.     (sendingProcess, outLine) =>
25.     {
26.         outputHandler(outLine.Data);
27.         PythonOutputHandler(sendingProcess, outLine, cmdOutputFile);
28.     }
29. );
```

Listing 29 – Definicja procesu oraz wyjść z konsoli w kodzie serwera.

6.6.5 Komenda „stop”

Komenda *stop* zatrzymuje aktualnie uruchomione uczenie (po komendzie *run*). Opcja ta nie przyjmuje żadnych dodatkowych parametrów (rys. 20).

```
Usage: client.exe stop [OPTIONS]

  Stop current running process.

Options:
  --help  Show this message and exit.
```

Rys. 20 – Komenda *stop* aplikacji uruchomiona z opcją *--help*.

Kod tej metody został opisany w listingu 30. Na początku sprawdzana jest autoryzacja, kolejno odczytywana jest nazwa projektu z pliku *.ai-exp* i ta poprzez klient HTTPS (linia 12.) wysyłana jest do serwera. Na końcu odczytywana jest odpowiedź zwrotna i pokazywana użytkownikowi.

```python
@click.command()
def stop():
    """
    Stop current running process.
    """
    AuthFileManager.exists_auth()

    experiment_config = ExperimentFileManager.get_config()
    upload_config = UploadConfig(name=experiment_config['name'])

    try:
        stop_response = ExperimentClient().stop(upload_config)
        aihub_logger.info(f"[INFO] {stop_response}")
    except Exception as e:
        aihub_logger.error("[ERROR] %s", e)
```

Listing 30 – Kod w aplikacji klienckiej dot. komendy *stop*.

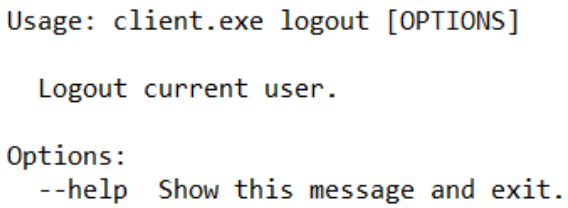
Listing 31 pokazuje, jak wygląda logika po otrzymaniu żądania HTTPS odnośnie zastopowania projektu. W linii 3. pobierany jest element z serwisu zarejestrowanego jako typ *Singleton* (czyli jego instancja jest cały czas taka sama przez czas działania serwera), gdzie zwracany jest obiekt, którego nazwa projektu odpowiada temu przysłanemu od klienta. Następnie, w zależności czy znaleziono element czy nie, proces uczenia jest wyłączany (linia 15.) lub zwracany jest typ *BadRequestObjectResult* (linia 10.), który odpowiada kodowi 400 HTTP (wraz z informacją).

```
1. public async Task<IActionResult> Handle(ExperimentsStopCommand command, CancellationToken cancellationToken)
2. {
3.     var run = _singletonSignalRService.Processes.FirstOrDefault(x => x.ProjectName == command.name);
4.
5.     var result = new ExperimentsStopCommandResult();
6.
7.     if (run == null)
8.     {
9.         result.Result = "There was no process running.";
10.        return new BadRequestObjectResult(result);
11.    }
12.
13.    if (run != null)
14.    {
15.        run.Process.Kill();
16.        result.Result = "Process killed.";
17.    }
18.
19.    return new OkObjectResult(result);
20. }
```

Listing 31 – Logika w kodzie serwera odnośnie komendy *stop*.

6.6.6 Komenda „logout”

Komenda *logout* wylogowuje aktualnego użytkownika. Opcja ta nie przyjmuje żadnych parametrów, jak pokazano na rys. 21.



Rys. 21 – Komenda *logout* aplikacji uruchomiona z opcją *--help*.

Ta operacja jest wykonywana tylko i wyłącznie na poziomie aplikacji klienckiej, opisanej w listingu 32. Sprawdzany jest, czy plik *.ai-auth* istnieje – jeżeli tak, to pobierany jest z niego JWT token. Kolejno, jeśli ten jest pusty, to znaczy że użytkownik już wcześniej był wylogowany, jeżeli nie, to ustawiany jest nowy plik, na całkowicie pusty.

```
1. @click.command()
2. def logout():
3.     """
4.    Logout current user.
5.    """
6.     AuthFileManager.exists_auth()
7.
8.     auth_settings = AuthFileManager.get_auth()
9.     access_token = auth_settings['access_token']
10.
11.    if not access_token:
12.        aihub_logger.info("[INFO] User wasn't logged. There is nothing to logout.")
13.
14.    if access_token:
15.        auth_settings = AuthSettings(access_token="")
16.        AuthFileManager.set_auth(auth_settings) # Create .ai-exp
17.        aihub_logger.info("[INFO] User logged out")
```

Listing 32 – Kod w aplikacji klienckiej dot. komendy *logout*.

7.Badania

Podczas konstruowania całej aplikacji, położono nacisk na wykorzystanie technologii, która jest nowa i jednocześnie szybka. Stworzony system nie powinien dodatkowo nadmiernie przedłużać czasu uczenia. Z tego powodu wybrano SignalR, którego dokumentacja [16] opisuje, że:

- w czasie rzeczywistym umożliwia kodowi po stronie serwera natychmiastowe przesyłanie treści do klientów
- dobrymi kandydatami do tej biblioteki są aplikacje do monitorowania

Aby sprawdzić praktyczny wpływ, postanowiono zmierzyć czas wykonywania procesów na serwerze w trakcie wywoływania komendy *run*. Test przeprowadzano na nieobciążonym laptopie *HP Pavilion* model *15-bw203nw* oraz lokalnym połączeniu sieciowym (którego opóźnienie sieciowe wynosi mniej niż 1 ms). Badanie było przeprowadzone 100 razy. W tym celu wykorzystano testowy plik (listing 33.) symulujący proces treningu modelu. Wykonuje on 2 iteracje pętli for, w trakcie której wyświetlane są informacje w konsoli (interwał 5 sekund). Na końcu wyświetlany jest wynik, ile czasu wykonywania zajął program w języku Python. Czas wykonywania tego listingu powinien wynosić w okolicach od 10 sekund wzwyż.

```
1. import time
2. start_time = time.time()
3. n = 2
4.
5. for x in range(0, n):
6.     print(f"Current x iteration: {x + 1} / {n}")
7.     print("Going to sleep for 5 seconds...")
8.     time.sleep(5)
9.     print("I'm awake!")
10.
11. print(f"Program took: {time.time() - start_time} seconds")
```

Listing 33 – Plik imitujący naukę w języku Python.

Kolejno, zgodnie z logiką serwera (graf na rys. 19), postanowiono zmierzyć czasy w 3 sposoby, opisane w tabeli 1. Metoda *OnConnected* sprawdza JWT token oraz uruchamia timer.

Tabela 1 – Scenariusz komendy *run* wraz z opisem mierzenia czasów.

Skrypt uruchomiony pierwszy raz	**Skrypt uruchomiony drugi raz z komendą *--run_again***	**Skrypt uruchomiony drugi raz bez komendy *--run_again***
1. Na końcu metody *OnConnected*	1. Na końcu metody *OnConnected*	1. Na końcu metody OnConnected
2. Uzupełnienie i wystartowanie procesu	2. Po sprawdzeniu logiki	2. Po sprawdzeniu logiki
3. Koniec procesu	3. Proces uzupełniony o nowe dane	3. Po odczycie logów z pliku
-	4. Wystartowanie procesu	-
-	5. Koniec procesu	-

Czasy zapisywano w konsoli aplikacji serwerowej poprzez metodę języka C#, tj. *Console.WriteLine*, gdzie jako parametr podawano czas, który upłynął w liczniku od początku wywoływania zapytania do SignalR.

W tabelach 2, 3 i 4 zaprezentowano średnie wyniki z 100 pomiarów jakie otrzymywano na poszczególnych etapach (tabela 1).

Tabela 2 – Średnie czasy poszczególnych etapów dla skryptu włączonego pierwszy raz

Proces	**Średni czas [s]**
1. Na końcu metody *OnConnected*	0.284
2. Uzupełnienie i wystartowanie procesu	0.417
3. Koniec procesu	10.494

Tabela 3 – Średnie czasy poszczególnych etapów dla skryptu włączonego drugi raz z komendą

Proces	**Średni czas [s]**
1. Na końcu metody *OnConnected*	0.000752
2. Po sprawdzeniu logiki	0.00107
3. Proces uzupełniony o nowe dane	0.00285
4. Wystartowanie procesu	0.083
5. Koniec procesu	10.145

Tabela 4 – Średnie czasy poszczególnych etapów dla skryptu włączonego drugi raz bez komendy

Proces	**Średni czas [s]**
1. Na końcu metody *OnConnected*	0.000752
2. Po sprawdzeniu logiki	0.00139
3. Po odczycie logów z pliku	0.0041

Analizując wyniki można zauważyć, że pierwszy krok, który jest zawsze taki sam, najwięcej czasu zajmuje w przypadku pierwszego uruchomienia. Jest to spowodowane konfiguracją oraz przypisaniem do listy procesów. Kolejne uruchomienia są szybsze, ponieważ jest odczytywany już utworzony proces z listy. Średni czas całkowitego uruchomienia i wysłania danych do użytkownika w przypadku pierwszego

uruchomienia to 10.494 sekund, w przypadku kolejnych uruchomień to 10.145 sekund. Odczyt logów z już wcześniej uruchomionego uczenia zajmuje 0.0041 sekund.

Średni czas wykonania samego programu w języku Python to około 10.024 sekund (wynik ten odczytano z logów). Mając to na uwadze, można stwierdzić, że średni dodatkowy czas w przypadku:

- pierwszego uruchomienia to 0.47 sekundy
- kolejnych uruchomień to 0.12 sekundy

Badania przeprowadzono na nieobciążonym serwerze (w sieci lokalnej), tj. bez innych użytkowników bądź uruchomionych procesów. Wyniki badania obciążenia zależałyby od zasobów, miejsca uruchomienia (np. laptop a dedykowany serwer w firmie) czy szybkości internetu.

8. Wnioski

Zgodnie z tematem pracy magisterskiej („Rozproszony system uczenia maszynowego z wykorzystaniem kart graficznych”) udało się zrealizować cele, tj.:

- przegląd technologii, literatury (dokumentacji), bibliotek, architektury i wzorców projektowych, które pozwoliły stworzyć działający system
- oprogramowanie umożliwiające uczenie na komputerze (serwerze), w którym karta graficzna nie jest w pełni obciążona

Oba składniki systemu są rozproszone (zarówno aplikacja kliencka jak i serwer). Dodatkowo, serwer może być uruchomiony na np. wirtualnej maszynie, ponieważ ASP.NET pozwala na użycie rozwiązań do konteneryzacji typu Docker oraz łatwą integrację z usługodawcami rozwiązań typu Cloud, takich jak Microsoft Azure, Amazon czy Google Cloud Platform.

Aplikacja konsolowa przeznaczona dla klientów posiada wszystkie niezbędne komendy potrzebne do uruchomienia uczenia, tj. założenie projektu, przesłanie plików, nauka oraz możliwość stopowania. Dodatkowo istnieje także system autoryzacji oparty na JWT tokenach, co pozwala na uwierzytelnianie użytkowników systemu, bezpieczeństwo oraz kontrolę.

Serwer (w ASP.NET) jest łącznikiem pomiędzy dostępnymi zasobami (dysk, pamięć karty graficznej), a użytkownikiem. Tworzy on folder, umieszcza w nim przesłane pliki oraz uruchamia wybrany do uczenia w konsoli. Możliwe jest wyłączenie nauki w każdym momencie, dodatkowo wyniki są zapisywane do plików oraz pokazywane w czasie rzeczywistym poprzez zastosowanie Websockets oraz biblioteki SignalR.

W obu przypadkach zastosowano architekturę pozwalającą na dalsze rozbudowywanie aplikacji i modułowe ich rozszerzanie. Elementy są od siebie niezależne poprzez wykorzystanie np. Dependency Injection czy MediatR w przypadku serwera.

Powstała aplikacja posiada interfejs przypominający, w pewnych aspektach, popularny system kontroli wersji Git, który też można obsługiwać w konsoli. Pozwala to na szybkie wdrożenie rozwiązania i łatwą interakcję.

Zaproponowany system nie obciąża czasowo uczenia. W przypadku pierwszego uruchomienia badania wykazały całkowity czas na poziomie 0.47 sekundy, a w kolejnych 0.12 sekundy. Jest to niewielki czas w stosunku do samej nauki, która w zależności od problemu może trwać godzinami. Korzyści płynące z użytkowania aplikacji – głównie to, że można uruchomić uczenie na innych zasobach niż własnych – przewyższają dodatkowy czas procesu.

Aplikacja ta ułatwia proces trenowania modelu uczenia maszynowego i zarządza zasobami oraz dostępami. Pozwala ona dzielić zasoby karty graficznej pośród wielu użytkowników, oszczędza to koszty zakupu sprzętu i potrzebnej energii elektrycznej w przypadku gdyby każdy z nich musiał kupić oddzielnie własną kartę graficzną do nauki modeli. Ponadto, to rozwiązanie redukuje koszty korzystania z podobnych usług chmurowych, których koszt miesięcznie może sięgać ponad 2000$ (w przypadku Google Cloud, maszyna z dostępną w ofercie kartą NVIDIA Tesla A100) [22].

9. Bibliografia

[1] BCC Research. Machine Learning: Global Markets to 2026 [online]. March 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://www.bccresearch.com/market-research/information-technology/machine-learning-global-markets.html

[2] Sarmah Simanta Shekhar. Artificial Intelligence in Automation [online]. RESEARCH REVIEW International Journal of Multidisciplinary. June 2019 [przeglądany 10 lutego 2023]. Dostępny w: https://www.researchgate.net/publication/336085049_Artificial_Intelligence_in_Automation

[3] Yash D. Patel. Machine Learning in HealthCare [online]. May 2021 [przeglądany 10 lutego 2023]. Dostępny w: https://www.researchgate.net/publication/351428558_Machine_Learning_in_HealthCare

[4] Youssef L. Machine learning in digital marketing. Machine learning in digital marketing [online]. 11-14-2021 [przeglądany 10 lutego 2023]. Dostępny w: https://www.researchgate.net/publication/364950346_Machine_learning_in_digital_marketing_Machine_learning_in_digital_marketing

[5] Thierry W. Machine Learning in Finance: A Metadata-Based Systematic Review of the Literature [online]. July 2021 [przeglądany 10 lutego 2023]. Dostępny w: https://www.researchgate.net/publication/352969157_Machine_Learning_in_Finance_A_Metadata-Based_Systematic_Review_of_the_Literature

[6] – Python Software Foundation. About Python [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://www.python.org/about/

[7] – Pallets. Click documentation [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://click.palletsprojects.com/en/8.1.x/

[8] – Steven Loria. Marshmallow dokumentacja [online], 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://marshmallow.readthedocs.io/en/stable/

[9] – Kenneth Reiz. Requests dokumentacja [online], 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://requests.readthedocs.io/en/latest/

[10] – Aymeric Augustin. Websockets dokumentacja [online], 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://websockets.readthedocs.io/en/stable/

[11] - Python Software Foundation. Asyncio dokumentacja [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://docs.python.org/3/library/asyncio.html

[12] - Python Software Foundation. Logging dokumentacja [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://docs.python.org/3/library/logging.html

[13] – Microsoft. ASP.NET dokumentacja [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://learn.microsoft.com/pl-pl/aspnet/core/?view=aspnetcore-6.0

[14] – Microsoft. ASP.NET Swashbuckle dokumentacja [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://learn.microsoft.com/pl-pl/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-6.0&tabs=visual-studio

[15] – Jimmy Bogard. MediatR biblioteka [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://github.com/jbogard/MediatR

[16] – Microsoft. SignalR dokumentacja [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-6.0

[17] – Microsoft. Dependency Injection dokumentacja [online]. 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

[18] – Microsoft. Clean Architecture dokumentacja [online]. 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://learn.microsoft.com/pl-pl/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures#clean-architecture

[19] – Microsoft. SignalR HubProtocol dokumentacja [online]. 2020 [przeglądany 10 lutego 2023]. Dostępny w: https://github.com/dotnet/aspnetcore/blob/main/src/SignalR/docs/specs/HubProtocol.md

[20] – Microsoft. SignalR Transport Protocols dokumentacja [online]. 2019 [przeglądany 10 lutego 2023]. Dostępny w: https://github.com/dotnet/aspnetcore/blob/main/src/SignalR/docs/specs/TransportProtocols.md

[21] - Seungmin Cho. Pandemic Effects on the Graphics Card Market [online]. 2022 [przeglądany 10 lutego 2023]. Dostępny w: https://onstarplus.com/archives/4208

[22] – Google Cloud Pricing Calculator [online]. 2023 [przeglądany 10 lutego 2023]. Dostępny w: https://cloud.google.com/products/calculator/#id=93bb8044-3b34-42b5-b220-3acfd2027fe8

10. Spis ilustracji z zewnętrznych źródeł

Poniżej znajduje się spis ilustracji użytych w pracy (w rys. 1):

- Użytkownik – app.moqups.com
- Python – python.org/community/logos/
- Plik z napisem PY - flaticon.com/free-icon/py-file-format_28807
- ASP.NET - bykowski.pl/wp-content/uploads/2013/10/aspnet.png
- Swagger - github.com/swagger-api
- Web API - https://tinyurl.com/3ukx5vz4
- SignalR - dotnet.microsoft.com/en-us/apps/aspnet/signalr
- „https://” - https://tinyurl.com/yyb6bv6k
- JWT - https://tinyurl.com/mr9tfd49