



**Politechnika
Śląska**

Dokumentacja projektowa

Zarządzanie Systemami Informatycznymi

Moduł 4 - Projekt Zaliczeniowy

Kierunek: Informatyka

Członkowie zespołu:

Bartłomiej Janoszka

Piotr Dusiński

Dominik Meisner

Mikołaj Molenda

Gliwice, 2024/2025

Spis treści

1	Wprowadzenie	2
1.1	Temat projektu:	2
1.2	Role w projekcie	2
1.3	Cel projektu	2
2	Założenia projektowe	3
2.1	Założenia techniczne i nietechniczne	3
2.1.1	Założenia techniczne	3
2.1.2	Założenia nietechniczne	3
2.2	Stos technologiczny	3
2.3	Oczekiwane rezultaty projektu	3
3	Realizacja projektu	4
3.1	Sprint 1	4
3.2	Sprint 2	5
3.2.1	DevOps Team	6
3.2.2	Dev Team	11
3.3	Sprint 3	13
3.3.1	DevOps Team	13
3.3.2	Dev Team	23
3.4	Sprint 4	33
3.4.1	DevOps Team	33
3.4.2	Dev Team	38
3.5	Sprint 5	41
3.5.1	DevOps Team	41
3.5.2	Dev Team	44
4	Wnioski	45
4.1	Co nam się udało?	45
4.2	Czego się nauczyliśmy?	45
4.3	Potencjał rozwoju	45
5	Bibliografia	46

1 Wprowadzenie

1.1 Temat projektu:

Symulacja infrastruktury IT w logice biznesowej za pomocą GNS3
+ wdrożenie aplikacji AI

1.2 Role w projekcie

- Bartłomiej Janoszka -> implementacja CI, konteneryzacja aplikacji, dokumentacja projektu, konfiguracja domeny, konfiguracja serwera
- Piotr Dusiński -> konfiguracja GNS3, konfiguracja klientów, konfiguracja domeny, konfiguracja serwera, wdrożenie aplikacji
- Mikołaj Molenda -> dokumentacja projektu, przygotowanie prezentacji projektu, stworzenie UI dla aplikacji, testy aplikacji, organizacja pracy zespołu
- Dominik Meisner -> dokumentacja projektu, stworzenie chatbot'a AI, stworzenie UI, organizacja pracy zespołu

1.3 Cel projektu



Rysunek 1: Logo Projektu

Celem projektu jest odwzorowanie środowiska biznesowego za pomocą narzędzia GNS3 oraz wdrożenie w nim za pomocą technologii konteneryzacji aplikacji z chatbot'em AI, który analizować będzie dostarczone do niego logi i informacje o błędach.

2 Założenia projektowe

2.1 Założenia techniczne i nietechniczne

2.1.1 Założenia techniczne

- Stworzenie wirtualnej infrastruktury informatycznej
- Stworzenie aplikacji z chatbot'em AI
- Wdrożenie aplikacji na serwer oraz umożliwienie klientom korzystania z niej
- Implementacja CI

2.1.2 Założenia nietechniczne

- Nauka logiki pracy w zespole w koncepcji Sacrum
- Nauka logiki działania infrastruktury informatycznej w kontekście biznesowym
- Zdobycie praktycznych umiejętności z zakresu: wirtualizacji, konteneryzacji, konfiguracji serwera, wdrażania aplikacji na serwer, konfiguracji domeny, konfiguracji klientów oraz podłączenia ich do domeny
- Symulacja pracy nad większym projektem w zespole oraz poznanie wykorzystywanych w tym celu narzędzi

2.2 Stos technologiczny

Narzędzia wykorzystane do pracy nad projektem:

- VMWare Workstation Pro - wirtualizacja systemów operacyjnych
- GNS3 - symulacja środowiska biznesowego
- Docker - konteneryzacja
- Repozytorium GitHub - praca nad aplikacją
- GitHub projects - organizacja pracy
- GitHub Actions - CI
- Ollama - uruchomienie modelu językowego
- Gemma3 - wykorzystany model językowy
- Python (FastAPI) - Backend aplikacji
- HTML, CSS, Java Script - frontend aplikacji
- Nginx - serwer frontend

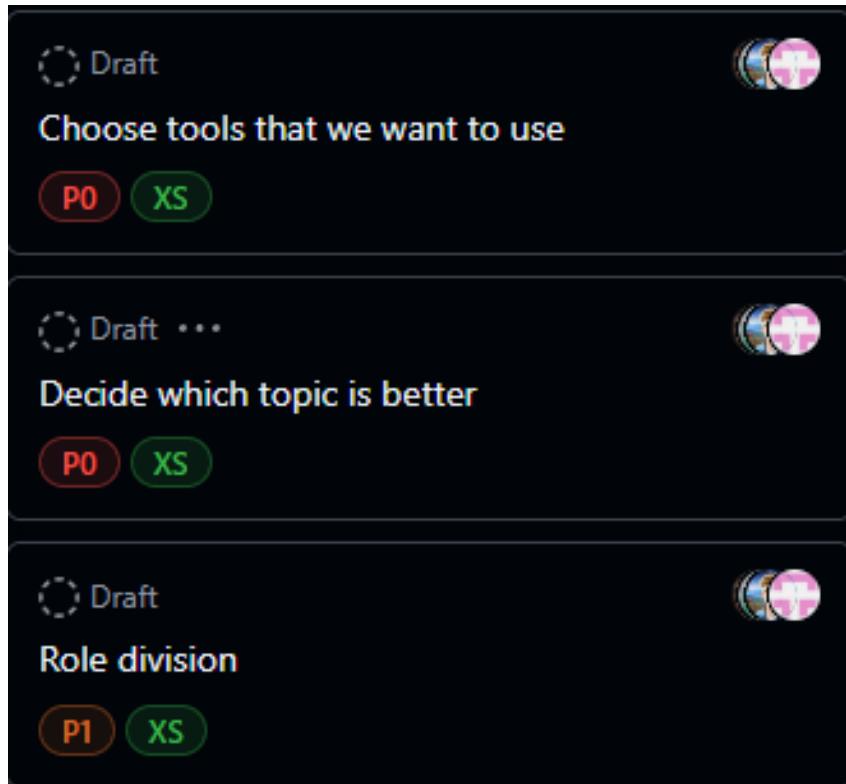
2.3 Oczekiwane rezultaty projektu

- Stworzenie w pełni funkcjonalnego mini-środowiska biznesowego z serwery oraz klientami podpiętymi do domeny.
- Stworzenie działającej aplikacji do analizy logów
- Wdrożenie aplikacji na serwer za pomocą kontenerów

- Dostęp klientów do aplikacji

3 Realizacja projektu

3.1 Sprint 1



Rysunek 2: Wybranie przez drużyny narzędzi.

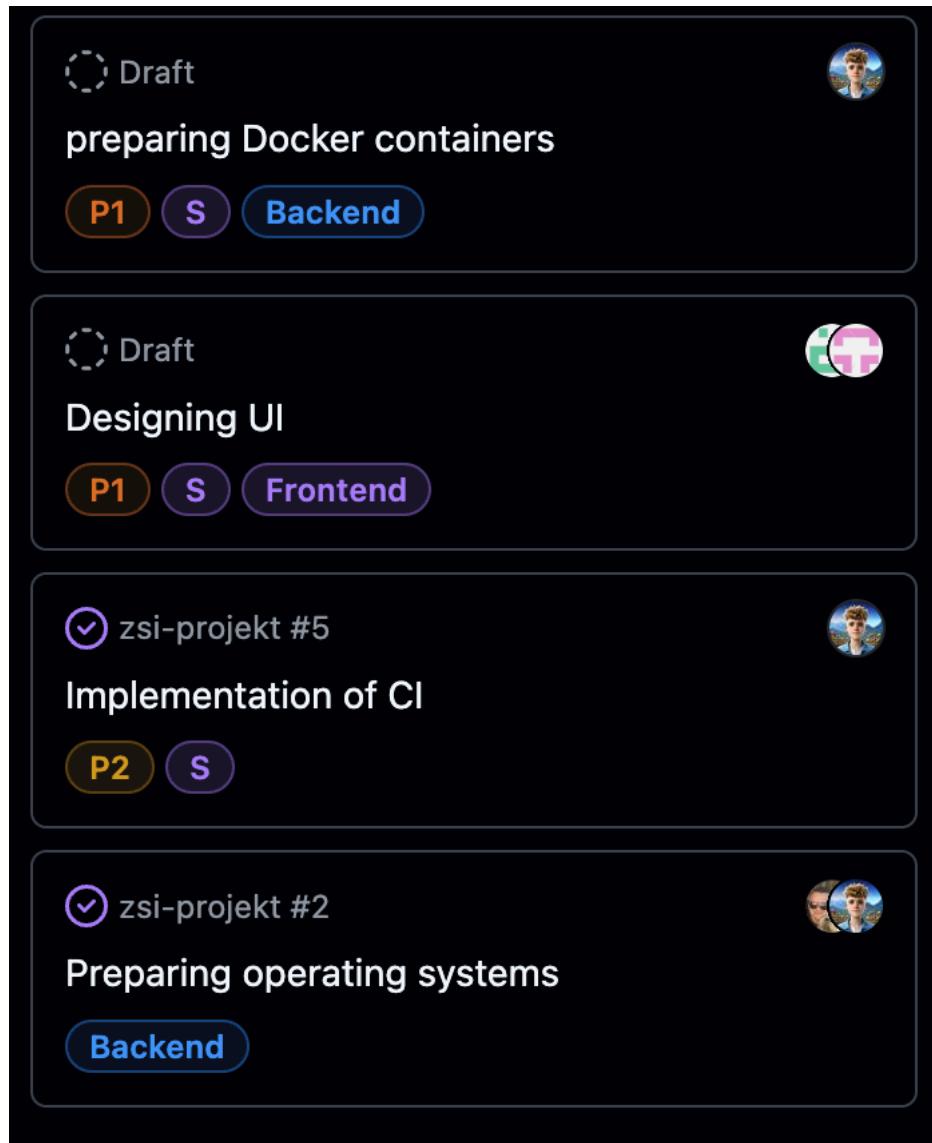
Naszym pierwszym Sprintem było wybranie tematu projektu, podział na role i zespoły (DevOps Team i Dev Team) oraz wybranie (już przez zespoły) odpowiednich narzędzi do pracy.

Podział na zespoły:

DevOps Team: Bartek i Piotr

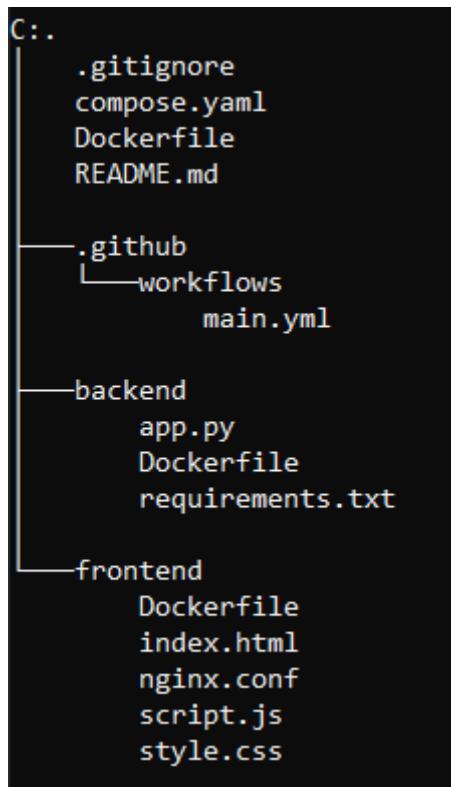
Dev Team: Mikołaj i Dominik

3.2 Sprint 2



Rysunek 3: Pocztatki prac nad projektem

Struktura aplikacji



Rysunek 4: Struktura projektu

3.2.1 DevOps Team

CI

Implementacja Continuous Integration:

```
name: build

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Check code
        uses: actions/checkout@v3
      - name: build app
```

```

    run: docker compose build
- name: tests
  run: |
    docker compose up -d
    sleep 15
    curl -f 127.0.0.0:80 || (echo "Fail!" && exit 1)
    docker compose down

```

Przy wypchnięciu kodu do głównej gałęzi projektu *main*, bądź przy scaleniu innej gałęzi z *main*, GitHub Actions uruchamia testy, które sprawdzają poprawność działania aplikacji. Proces ten polega na automatycznym zbudowaniu obrazu Dockera oraz uruchomieniu środowiska testowego za pomocą docker compose. Następnie wykonywane jest zapytanie HTTP do serwera aplikacji w celu sprawdzenia, czy uruchomiła się ona poprawnie. W przypadku niepowodzenia testu workflow zostaje przerwany z komunikatem o błędzie.

Dzięki temu każda zmiana wprowadzona do głównej gałęzi jest automatycznie weryfikowana, co pozwala szybko wychwycić ewentualne błędy i zapewnia stabilność projektu.

Konteneryzacja aplikacji

compose.yaml

```

services:
  ollama-app:
    build: .
    container_name: ollama-app
    networks:
      - app-net
    volumes:
      - model:/root/.ollama
  develop:
    watch:
      - path: ./Dockerfile
        action: rebuild

backend:
  build: ./backend/.
  container_name: backend
  networks:
    - app-net
  depends_on:
    - ollama-app
  develop:
    watch:
      - path: ./backend

```

```

        action: sync
        target: /app
    - path: ./backend/requirements.txt
        action: rebuild
    - path: ./backend/Dockerfile
        action: rebuild

frontend:
    build: ./frontend/
    container_name: frontend
    networks:
        - app-net
    depends_on:
        - backend
    ports:
        - 80:8080
    develop:
        watch:
            - path: ./frontend
                action: sync
                target: /usr/share/nginx/html
            - path: ./frontend/Dockerfile
                action: rebuild

networks:
    app-net:
        driver: bridge
volumes:
    model:

```

Plik konfiguruje trzy serwisy Docker Compose:

- ollama-app – buduje obraz z katalogu głównego, montuje wolumen na modele i przebudowuje się przy zmianie Dockerfile.
- backend – buduje się z katalogu *backend*, startuje po *ollama-app*, synchronizuje kod na bieżąco oraz przebudowuje obraz po zmianach w Dockerfile i requirements.txt.
- frontend – buduje się z katalogu *frontend*, startuje po *backend*, udostępnia port *80*, synchronizuje pliki frontu i przebudowuje się przy zmianie Dockerfile.

Wszystkie serwisy są połączone w jednej sieci (*app-net*), a dane modeli są przechowywane w wolumenie *model*. Dodatkowo skonfigurowane jest automatyczne śledzenie zmian podczas developmentu (opcja docker compose watch).

DockerFile - Ollama-app

```
FROM ollama/ollama

RUN ollama serve & sleep 5 && ollama pull gemma3:1b
```

Ten kod buduje nowy obraz Dockera na bazie oficjalnego obrazu *ollama/ollama*. W instrukcji *RUN* najpierw uruchamiana jest usługa *ollama serve* w tle i odczeka 5 sekund, a następnie pobierany jest model *gemma3:1b* do środowiska Ollama za pomocą polecenia *ollama pull*. Dzięki temu model jest gotowy do użycia od razu po uruchomieniu kontenera.

DockerFile - backend

```
FROM python:3.13

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000",
"--reload"]
```

Plik Dockerfile buduje obraz kontenera do uruchomienia aplikacji Python, bazując na oficjalnym obrazie Python 3.13. Ustawia katalog roboczy na */app*, kopiuje plik *requirements.txt* i instaluje znajdujące się w nim zależności, a następnie dodaje główny plik aplikacji *app.py*. Kontener po uruchomieniu startuje serwer aplikacji za pomocą Uvicorn, udostępniając ją na porcie *8000*. Opcja *reload* umożliwia automatyczne przeładowanie aplikacji przy zmianach w kodzie, co jest przydatne w trakcie developmentu.

DockerFile - frontend

```
FROM nginx

COPY nginx.conf /etc/nginx/nginx.conf

COPY index.html script.js style.css /usr/share/nginx/html/

CMD nginx -g 'daemon off;'
```

Plik Dockerfile buduje obraz kontenera, który uruchamia statyczną stronę internetową z wykorzystaniem serwera Nginx. Jako bazę wykorzystuje oficjalny obraz Nginx, a następnie podmienia domyślną konfigurację serwera na własny plik *nginx.conf*. Pliki strony (*index.html*, *script.js*, *style.css*) kopiowane są do katalogu */usr/share/nginx/html/*, skąd Nginx serwuje je jako stronę WWW. Serwer uruchamiany jest na pierwszym planie, co jest wymagane w środowisku Docker. Efektem działania tak skonfigurowanego obrazu jest gotowy kontener, który po uruchomieniu udostępnia statyczną stronę WWW z własną konfiguracją Nginx.

nginx.conf

```
events {}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    server {
        listen 8080;

        location / {
            root /usr/share/nginx/html;
            index index.html;
            try_files $uri $uri/ /index.html;
        }

        location /api/ {
            proxy_pass http://backend:8000/;
            proxy_read_timeout 300;
            proxy_connect_timeout 300;
            proxy_send_timeout 300;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

    }
}
```

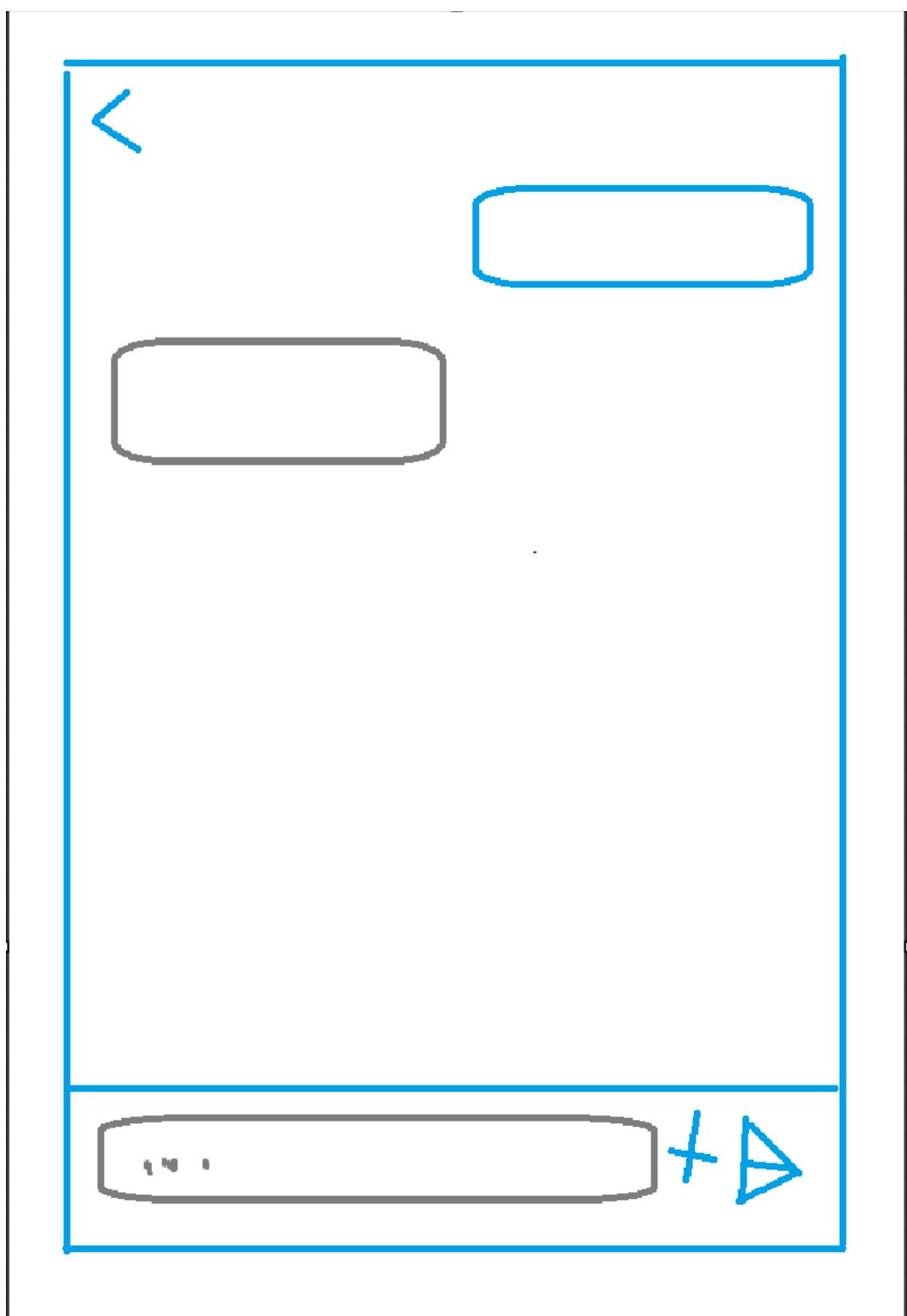
Plik konfiguracyjny Nginx definiuje serwer nasłuchujący na porcie 8080. Dla żądań do katalogu głównego serwuje statyczne pliki z */usr/share/nginx/html*, korzystając z mechanizmu *try_files*, który umożliwia obsługę aplikacji

typu Single Page Application (SPA). Wszystkie żądania do ścieżki `/api/` są przekierowywane do serwera backendowego działającego pod adresem `http://backend:8000/`, z odpowiednimi nagłówkami i zwiększymi timeoutami na potrzeby dłuższych operacji.

3.2.2 Dev Team

Projekt UI

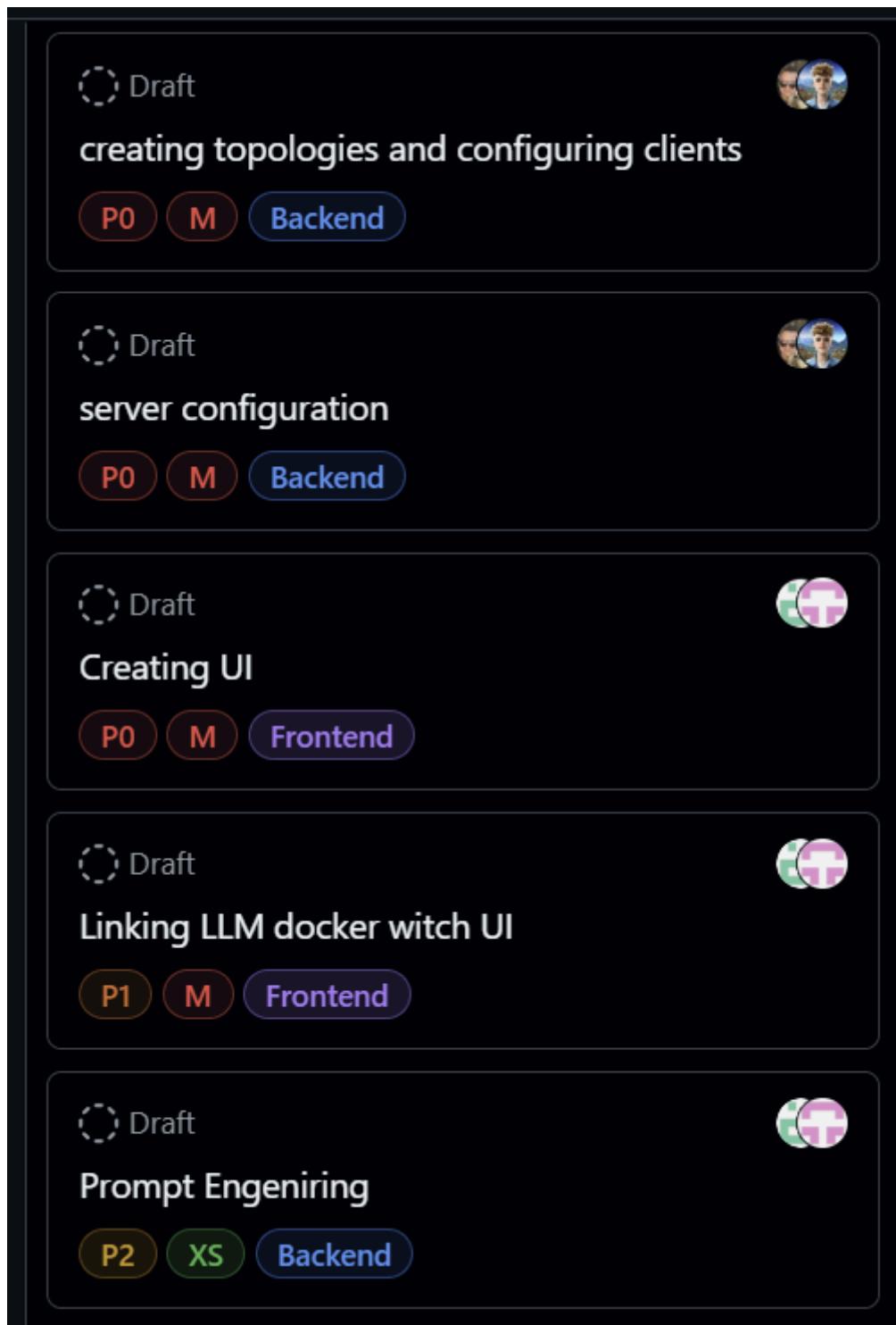
Projektując UI zainspirowaliśmy się wyglądem nowoczesnych chatów AI obsługujących modele LLM takich jak Chat GTP, Gemini, czy też Claude. Chcieliśmy również aby nasz czat był czytelny i przyjazny dla użytkowników, dlatego postanowiliśmy przybliżyć go wyglądem do czatów które używamy na codzień tak, aby odbiorca poczuł się jakby naprawdę rozmawiał z kimś po drugiej stronie. Po analizie paru przykładowych czatów postanowiliśmy wystylizować nasz wygląd UI na bazie iMessage.



Rysunek 5: Pierwotny projekt wizualny w paint

W tym sprincie podjeliśmy również decyzje jak chcemy budować UI. Postawiliśmy na standardowy HTML wsparły przez CSS oraz JavaScript.

3.3 Sprint 3



Rysunek 6: Sprint 3

3.3.1 DevOps Team

Konfiguracja klientów

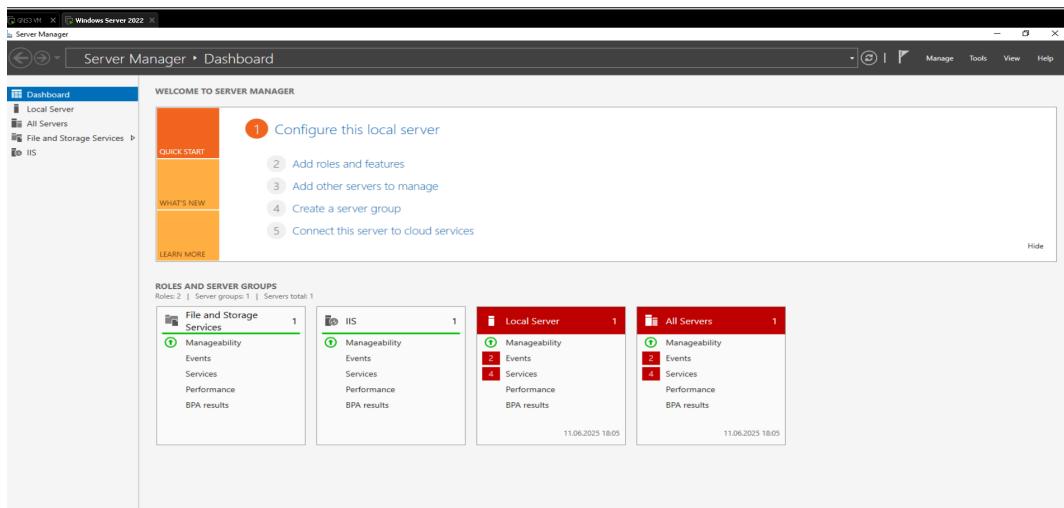
Pracę naszego zespołu zaczęliśmy od konfiguracji klientów, którym będą wirtualne maszyny z Windowsem 11



Rysunek 7: Przykładowy komputer kliencki po konfiguracji

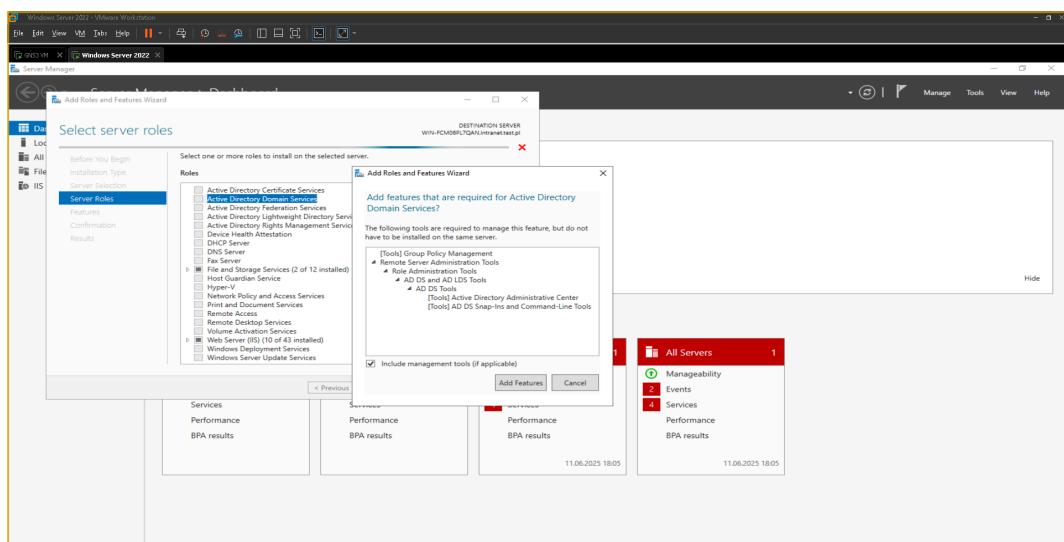
Zdecydowaliśmy się pominąć proces instalacji systemu Windows 11, ze względu na prostotę tej instalacji, która to polega głównie na przeklikiwaniu instalatora za pomocą przycisku *Dalej*. Na powyższym zrzucie ekranu znajduje się już finalny efekt, czyli świeżo zainstalowany system.

Konfiguracja serwera Windows w wersji 2022



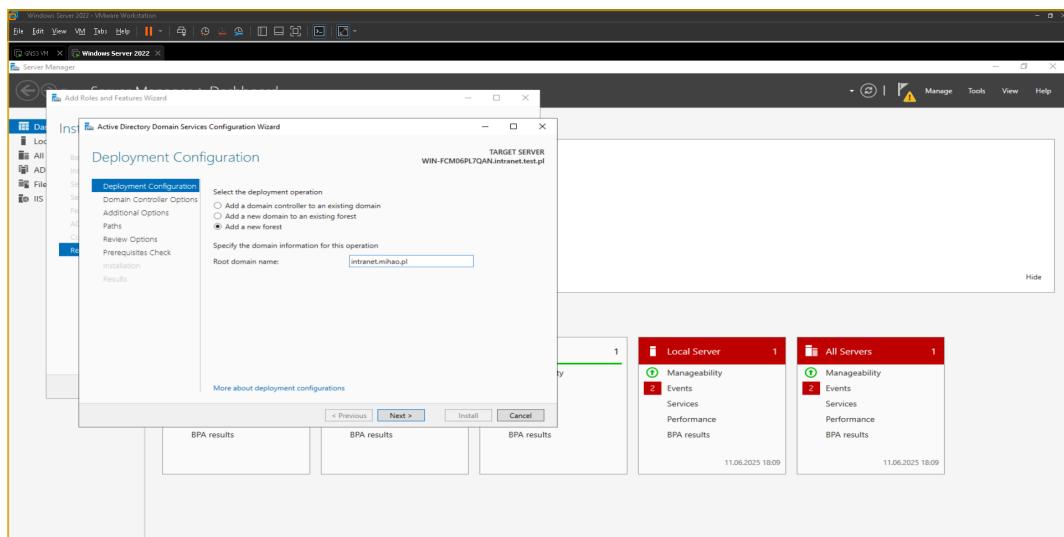
Rysunek 8: Konfiguracja serwera Windows

Po instalacji systemu, do dalszej jego konfiguracji wykorzystamy panel do zarządzania serwerem.



Rysunek 9: Konfiguracja serwera Windows - instalacja Active Directory

Klikamy na zakładkę *Manage* a następnie wybieramy opcję *Add Roles and Features*. Następnie wybieramy instalację na lokalnym serwerze (czyli na tym, na którym aktualnie pracujemy) i spośród dostępnych opcji wybieramy *Uslugi domenowe Active Directory*, dalej już tylko przechodzimy przez kolejne kroki instalatora.



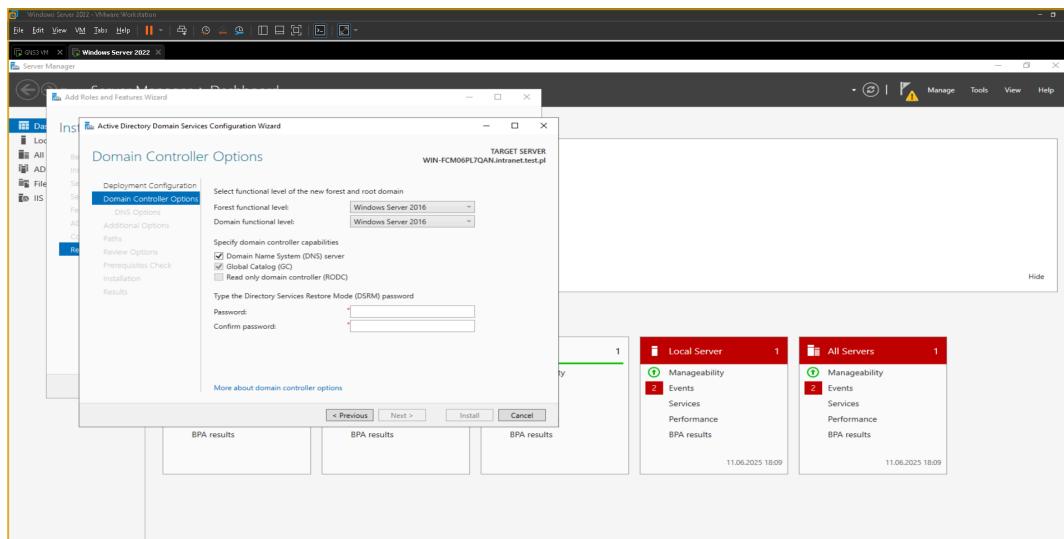
Rysunek 10: promocja serwera do roli kontrolera domeny

Po zakończeniu instalacji musimy utworzyć nową domenę oraz wypromować nasz serwer do roli jej kontrolera. W powiadomieniach (ikonka flagi) otrzymamy stosowne powiadomienie (żółty wykryzykni) o tym, że musimy wykonać tę czynność. Wystarczy wejść w wspomniane powiadomienie, aby otworzyć kreator domeny.

Wybieramy (jak widać na załączonym zrzucie ekranu) *Add new forest*,

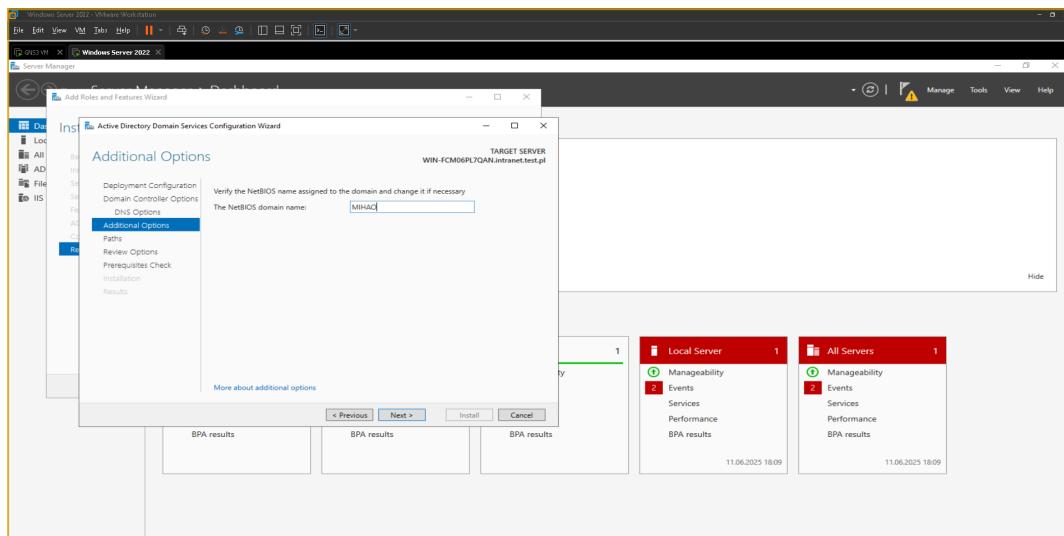
po czym poniżej wpisujemy pełną nazwę domeny - w naszym przypadku *intranet.mihao.pl*.

Warto nadmienić iż wraz z usługami domenowymi AD doinstalowywany jest również serwer DNS, a utworzenie nowego lasu dodaje odpowiednie rekordy pozwalające rozwiązywać nazwę domeny.



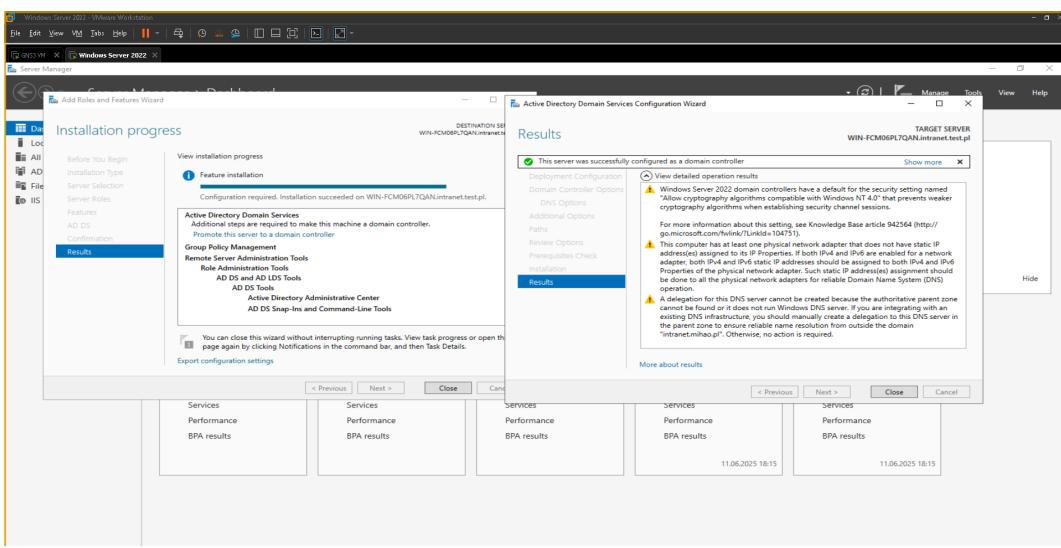
Rysunek 11: promocja serwera do roli kontrolera domeny cz.2

Wybieramy wersję usług domenowych (albo bardziej zostawiamy domyślną) i ustawiamy hasło przywracania.

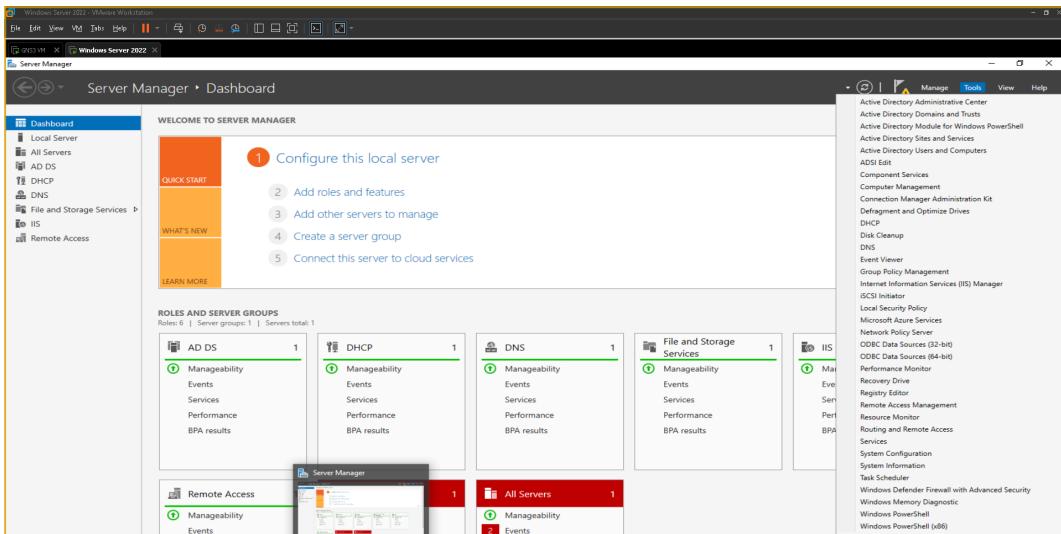


Rysunek 12: promocja serwera do roli kontrolera domeny cz.3

Wpisujemy nazwę NetBIOS *MIHAO* i przeklikujemy kreator do końca z opcjami domyślnymi.



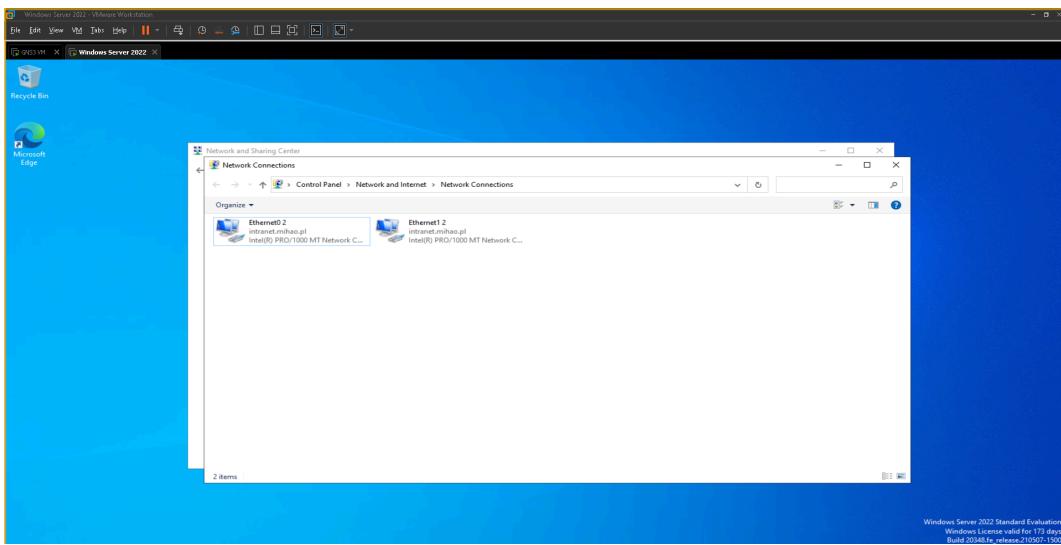
Po wszystkim zobaczymy następujący efekt i konieczne będzie ponowne uruchomienie serwera.



Rysunek 14: rezultat instalacji DHCP oraz NAT

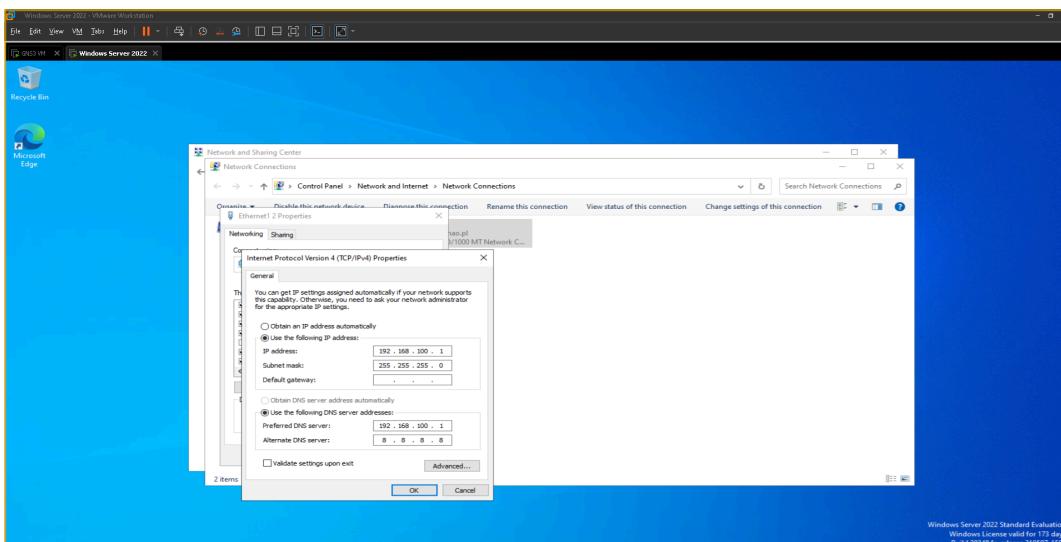
Następnie doinstalowujemy usługę *serwera DHCP* oraz usługę *translacji adresów NAT*. Logika instalacji jest podobna do instalacji usługi *Active Directory*, tylko tym razem wybieramy z listy wyżej wymienione usługi. Dlatego proces instalacji pominiemy i przejdziemy bezpośrednio do konfiguracji tych usług.

Powyżej zobaczyć możemy zrzut ekranu z panelu zarządzania serwerem już po doinstalowaniu *serwera DHCP* oraz *translacji adresów NAT*.



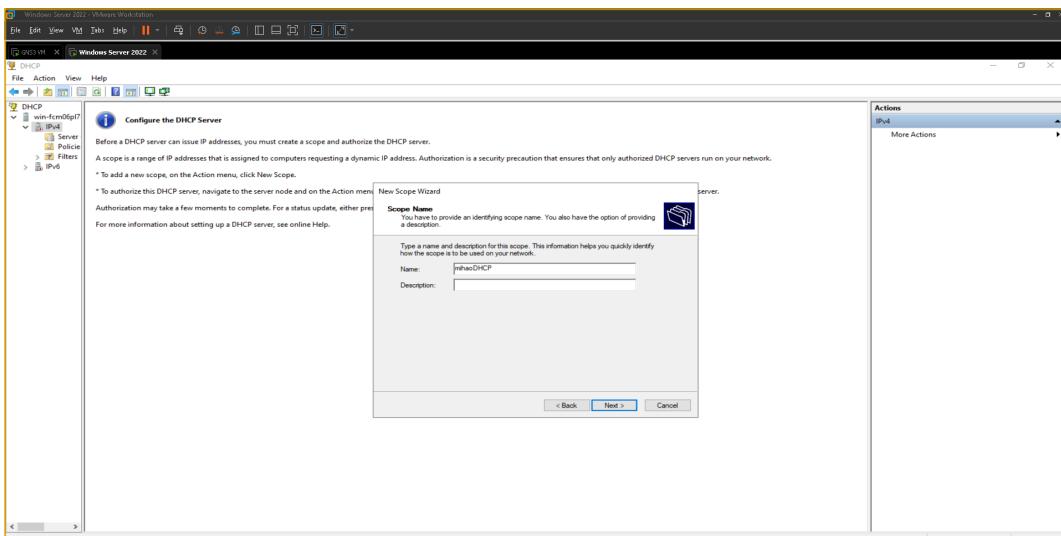
Rysunek 15: interfejsy sieciowe

W naszym serwerze wykorzystywać będziemy dwa interfejsy sieciowe do pierwszego z nich *Ethernet0 2* podłączony będzie internet, dzięki czemu umożliwimy naszej sieci „wyjście na świat”. Drugi zaś będzie interfejsem wewnętrz sieci, to właśnie on będzie wykorzystywany jako interfejs dla usługi serwera DHCP oraz jako brama domyślna.



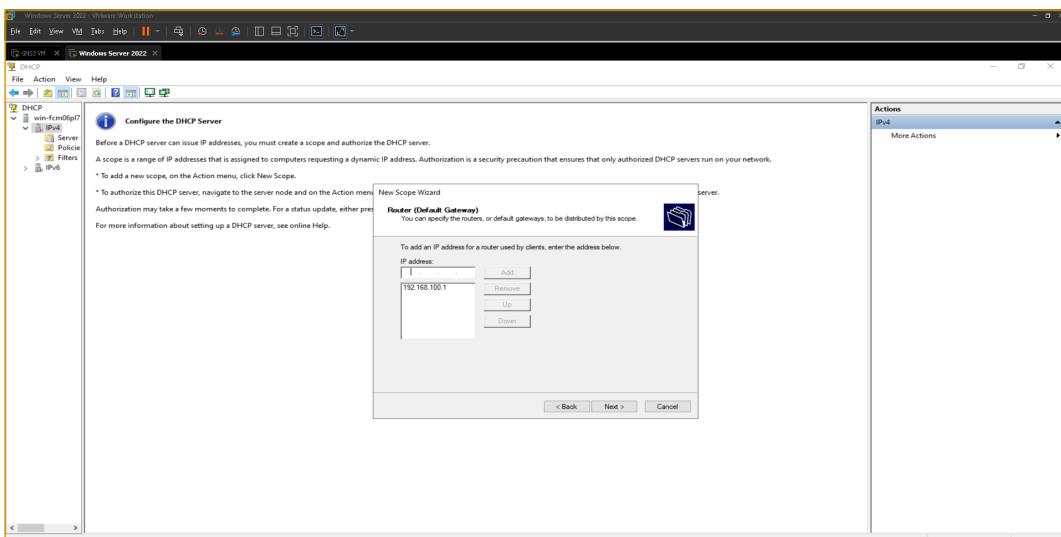
Rysunek 16: konfiguracja wewnętrznego interfejsu sieciowego

Konfigurujemy wewnętrzny interfejs sieciowy serwera. Ten interfejs pracować będzie jako router, a więc przypisujemy mu pierwszy wolny adres w sieci, czyli *192.168.100.1*. Maska podsieci *255.255.255.0*. Brama domyślna zostawiamy puste. DNS jako pierwszy wpisujemy adres naszego serwera, natomiast jako drugi możemy wpisać adres serwera Google, czyli *8.8.8.8*



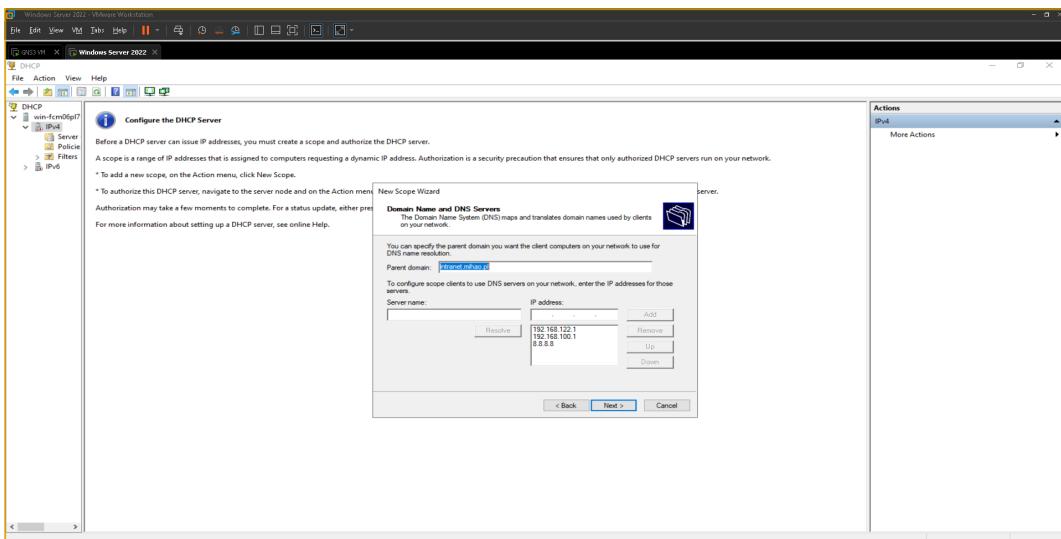
Rysunek 17: konfiguracja serwera DHCP

Teraz przechodzimy do konfiguracji serwera DHCP. Po wejściu na usługę poprzez panel serwera (*tools -> DHCP*) rozwijamy listę (tę z nazwa serwera) i klikamy prawym przyciskiem na *IPv4*, a następnie *configure new scope*. Dalej otworzy się pokazany na zrzucie ekranu konfigurator nowego zakresu puli IPv4. W polu nazwa wpisujemy dowolna nazwę, po której będziemy identyfikować nasz zakres, możemy jeszcze w razie potrzeby dodać opis, ale pole to jest opcjonalne.



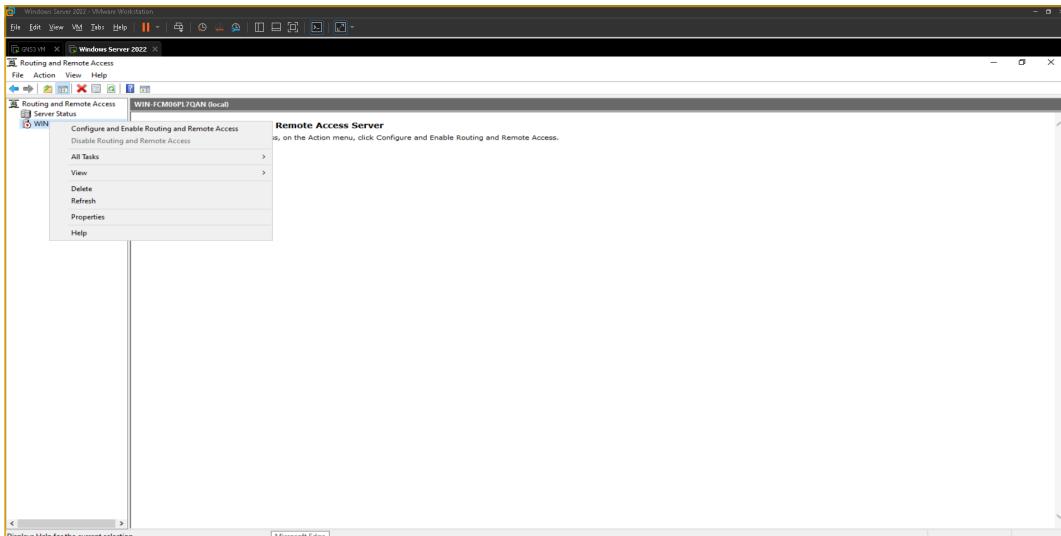
Rysunek 18: konfiguracja serwera DHCP cz.2

Teraz kreator prosi nas po podanie adresu IP bramy domyślnej, w naszym przypadku jest to *192.168.100.1*



Rysunek 19: konfiguracja serwera DHCP cz.3

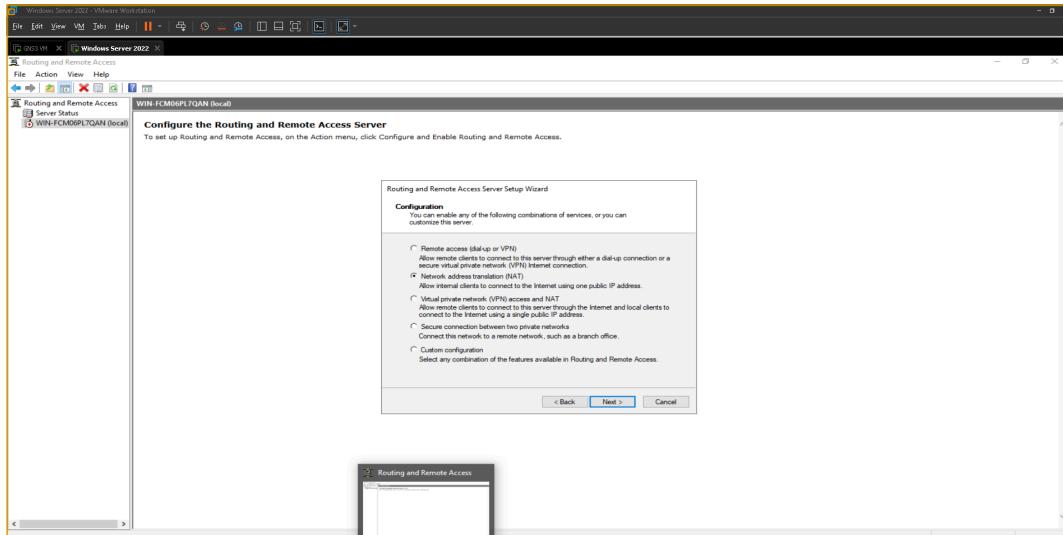
Teraz konfigurator prosi o ustawienie serwerów DNS. Ustawiamy adres naszego serwera (*192.168.100.1*), adres serwera DNS Google (*8.8.8.8*) oraz tutaj system ustawił za nas adres serwera DNS „chmury NAT” z narzędzia GNS3 (*192.168.122.1*). Jest to ostatni etap konfiguratora, o którym wspominamy dalej jest to po prostu przechodzenie przez kolejne etapy poprzez klikanie przycisku *next*. Końcowo konfigurator tworzy nowy zakres i ustawia go jako aktywny, tym samym usługa *serwera DHCP* już działa w naszej sieci wewnętrznej.



Rysunek 20: konfiguracja usługi NAT

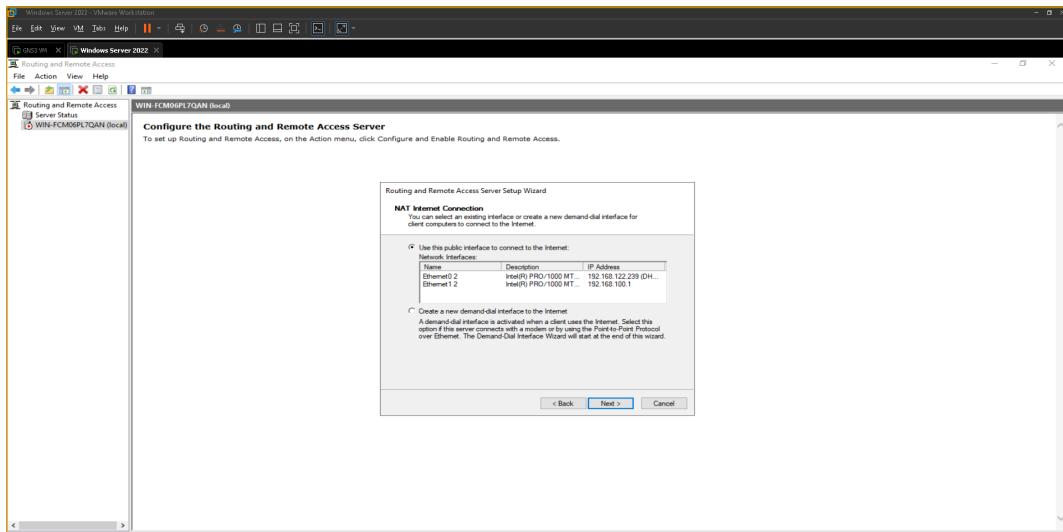
Przechodzimy do konfiguracji usługi *translacji adresów NAT*. W tym celu w panelu zarządzania serwerem wybieramy *tools -> Remote Access and Routing*. Uruchamia się przystawka do zarządzania dostępem zdalnym oraz

routingiem. W bocznym panelu klikamy prawym przyciskiem na nazwę naszego serwera i wybieramy opcję *Configure and Enable Routing and Remote Access*.



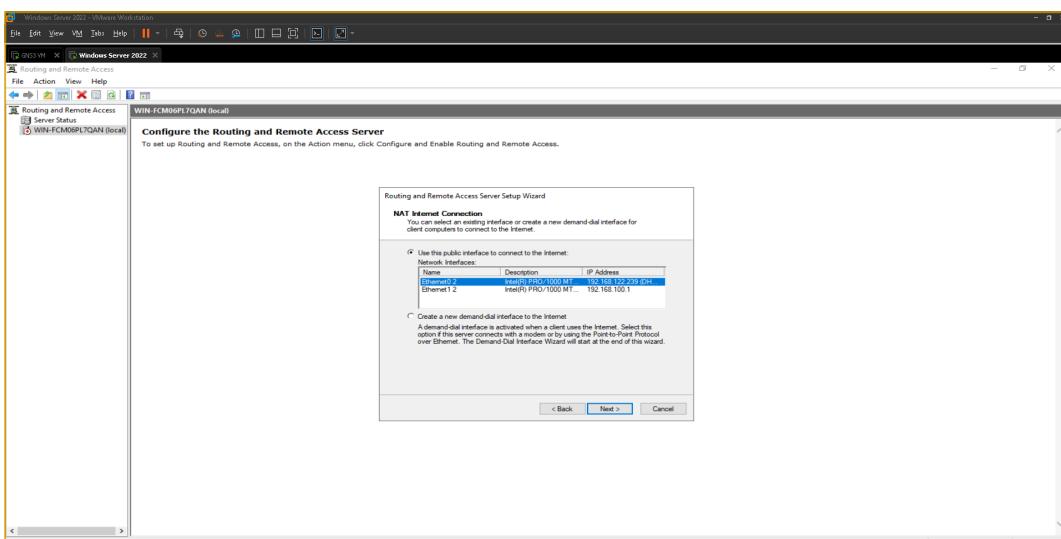
Rysunek 21: konfiguracja usługi NAT cz.2

Ponownie otworzy nam się konfigurator, w którym to podobnie jak na powyższym zrzucie ekranu wybieramy drugie pole radio - *Network address translation (NAT)*, klikamy *Next*.



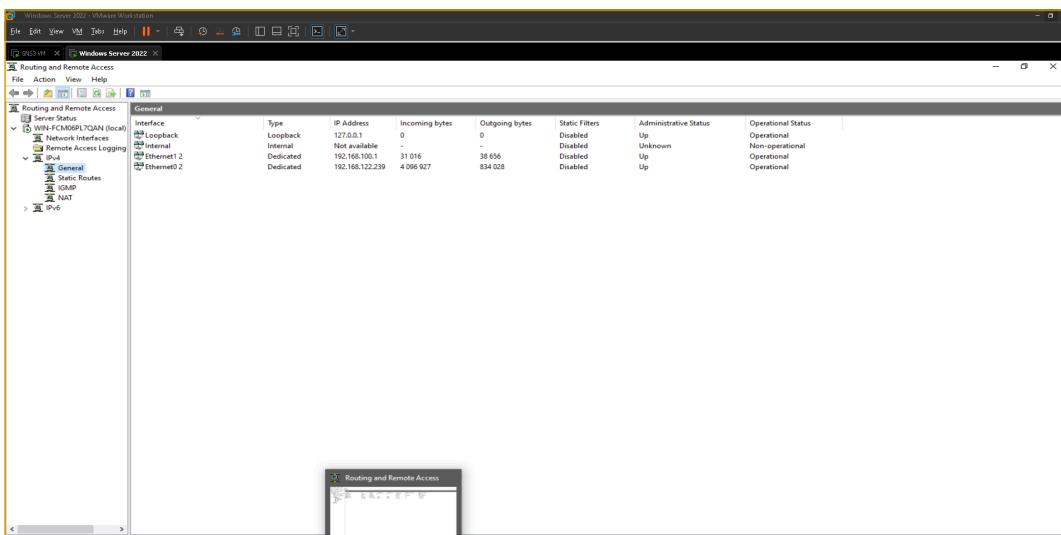
Rysunek 22: konfiguracja usługi NAT cz.3

Teraz wybieramy interfejs, który ma dostęp do internetu.



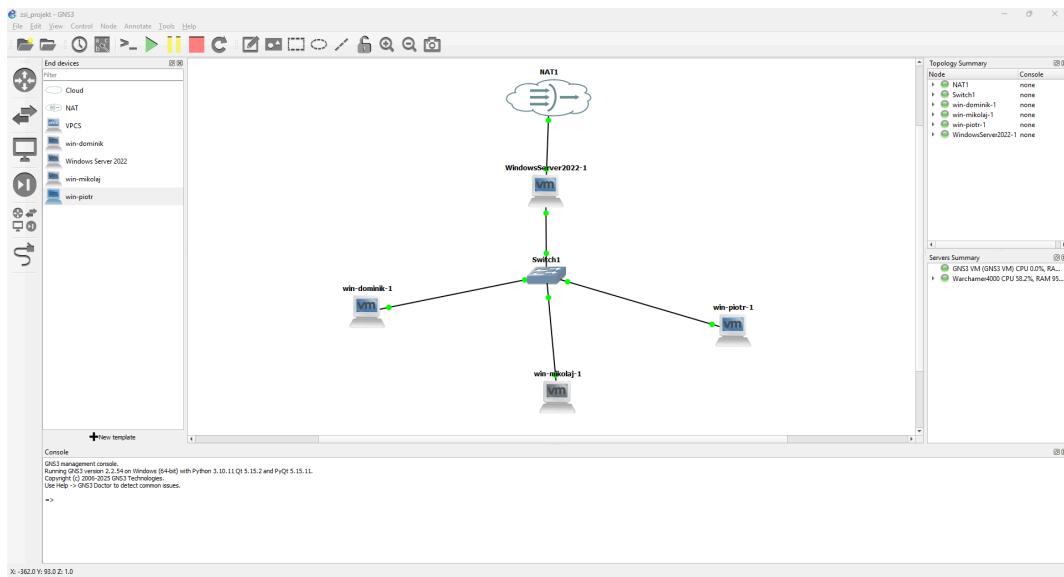
Rysunek 23: konfiguracja usługi NAT cz.4

W naszym przypadku jest to *Ethernet0 2*, wybieramy go i klikamy *Next*.



Rysunek 24: efekt konfiguracji usługi NAT

Kończymy konfiguracje usługi *NAT* przechodząc przez cały konfigurator. Po zakończeniu konfiguracji powinniśmy zobaczyć efekt jak na zrzucie ekranu powyżej.



Rysunek 25: Zdjęcie finalnej topologii

W finalnej wersji topologia wygląda w następujący sposób. Do przełącznika *Switch1* podpinamy *WindowsServer2022-1* (podpinamy drugi interfejs), *win-dominik-1*, *win-mikolaj-1* oraz *win-piotr-1*. Pierwszy interfejs sieciowy serwera podpinamy do „chmury NAT” z GNS3, aby umożliwić serwerowi oraz klientom (po przez NAT) dostęp do internetu. *WindowsServer2022-1*, *win-dominik-1*, *win-mikolaj-1* oraz *win-piotr-1* są maszynami wirtualnymi, które zainportowaliśmy do GNS3.

3.3.2 Dev Team

Informacja: Kod UI umieszczony w dokumentacji został po zakończeniu projektu wicę jest zgodny z aktualnym stanem aplikacji, a nie stanem w momencie jej tworzenia

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Localhost Expert</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
    <script src="https://cdn.jsdelivr.net/npm/marked/marked.min.
js"></script>
    <script src="script.js" defer></script>
  </head>
```

```

<body>
  <div class="chat-container">
    <div class="chat-header">
      
      <div class="username">localhost Expert</div>
    </div>
    <div id="chat-window" class="chat-window"></div>
    <form id="chat-form" class="chat-form">
      <div class="input-group">
        <input id="user-input" type="text" placeholder="Type a
message" autocomplete="off" />
        <div class="file-input-wrapper">
          <label for="file-input" class="file-label">
            <svg xmlns="http://www.w3.org/2000/svg" width="20"
height="20" viewBox="0 0 24 24" fill="none" stroke="currentColor"
stroke-width="2">
              <path d="M21.44 11.05l-9.19 9.19a6 6 0 0
1-8.49-8.49l9.19-9.19a4 4 0 0 1 5.66 5.66l-9.2 9.19a2 2 0 0
1-2.83-2.83l8.49-8.48" />
            </svg>
          </label>
          <input id="file-input" type="file" />
        </div>
        <button type="submit" class="send-button">
          <svg xmlns="http://www.w3.org/2000/svg" width="20"
height="20" viewBox="0 0 24 24" fill="none" stroke="currentColor"
stroke-width="2">
            <line x1="22" y1="2" x2="11" y2="13" />
            <polygon points="22 2 15 22 11 13 2 9 22 2" />
          </svg>
        </button>
      </div>
    </form>
  </div>
</body>
</html>

```

Rozpoczęliśmy prace nad tworzeniem interfejsu użytkownika (UI). Na początku stworzyliśmy prosty model wyglądu UI w HTML – utworzyliśmy odpowiednie kontenery, aby móc je następnie odpowiednio ostylować i dopracować za pomocą CSS. Topologia kontenerów HTML wygląda następująco:

- chat-container
 - chat-header

- username
- ▶ chat-window
- ▶ [Form] chat-form
 - input-group
 - user-input
 - file-input-wrapper
 - [Button] submit

Po utworzeniu pliku HTML zabraliśmy się za konfigurowanie UI przy pomocy CSSa.

style.css

```
*,
*:before,
*:after {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  margin: 0;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
  background-color: #f7f7f8;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
}

.chat-container {
  width: 100%;
  max-width: 600px;
  height: 90vh;
  display: flex;
  flex-direction: column;
  border: 1px solid #ddd;
  background: white;
  border-radius: 10px;
  overflow: hidden;
  box-shadow: 0 4px 20px rgba(0, 0, 0, 0.05);
}

.chat-header {
  display: flex;
  align-items: center;
```

```
gap: 10px;
padding: 15px 20px;
border-bottom: 1px solid #ddd;
background-color: #f1f1f1;
}

.avatar {
width: 40px;
height: 40px;
border-radius: 50%;
}

.username {
font-size: 18px;
font-weight: 600;
}

.chat-window {
flex: 1;
padding: 20px;
overflow-y: auto;
display: flex;
flex-direction: column;
gap: 15px;
}

.message {
max-width: 80%;
padding: 12px 16px;
border-radius: 12px;
line-height: 1.5;
word-wrap: break-word;
}

.user-message {
align-self: flex-end;
background-color: #007bff;
color: white;
border-bottom-right-radius: 0;
}

.bot-message {
align-self: flex-start;
background-color: #e5e5ea;
color: black;
border-bottom-left-radius: 0;
}
```

```

.typing-indicator {
  width: 50px;
  height: 35px;
  display: flex;
  align-items: center;
  justify-content: space-around;
  background-color: #e5e5ea;
  padding: 10px 14px;
  border-radius: 16px;
  align-self: flex-start;
  margin-top: -10px;
}

.typing-indicator span {
  width: 8px;
  height: 8px;
  background-color: #555;
  border-radius: 50%;
  animation: typing 1.2s infinite;
}

.typing-indicator span:nth-child(2) {
  animation-delay: 0.2s;
}

.typing-indicator span:nth-child(3) {
  animation-delay: 0.4s;
}

@keyframes typing {
  0%,
  80%,
  100% {
    transform: scale(0.8);
    opacity: 0.3;
  }
  40% {
    transform: scale(1);
    opacity: 1;
  }
}

.chat-form {
  padding: 15px;
  background: #f5f5f5;
  border-top: 1px solid #e0e0e0;
}

```

```
.input-group {
  display: flex;
  align-items: center;
  gap: 10px;
  background: #fff;
  padding: 8px;
  border-radius: 24px;
  box-shadow: 0 2px 6px rgba(0,0,0,0.1);
}

#user-input {
  flex: 1;
  border: none;
  outline: none;
  padding: 8px;
  font-size: 14px;
  background: transparent;
}

.file-input-wrapper {
  position: relative;
  display: flex;
  align-items: center;
}

#file-input {
  display: none;
}

.file-label {
  cursor: pointer;
  padding: 8px;
  border-radius: 50%;
  display: flex;
  align-items: center;
  justify-content: center;
  transition: background-color 0.2s;
}

.file-label:hover {
  background-color: #f0f0f0;
}

.send-button {
  background: #007bff;
  color: white;
  border: none;
  border-radius: 50%;
```

```

width: 40px;
height: 40px;
display: flex;
align-items: center;
justify-content: center;
cursor: pointer;
transition: background-color 0.2s;
padding: 0;
}

.send-button:hover {
background: #0056b3;
}

.send-button:disabled {
background: #cccccc;
cursor: not-allowed;
}

.file-selected .file-label {
color: #007bff;
background-color: #e6f2ff;
}

```

W powyższym pliku CSS zawarliśmy kluczowe elementy wyglądu naszej aplikacji. Plik zawiera style zapewniające nowoczesny, responsywny układ aplikacji: zaokrąglone kontenery, cieniowanie, kolorystykę rozróżniającą wiedomości użytkownika i bota, animowany wskaźnik pisania oraz estetyczne przyciski i pole do załączania plików. Stylizując zadbaliśmy o czytelność, wygodę obsługi i spójność wizualną całego czatu.

Java Script

```

const chatForm = document.getElementById("chat-form");
const chatWindow = document.getElementById("chat-window");
const userInput = document.getElementById("user-input");
const submitButton = chatForm.querySelector("button");
const fileInput = document.getElementById("file-input");

function generateUserId() {
    return "user_" + Math.random().toString(36).substr(2, 9);
}

function getUserId() {
    let userId = localStorage.getItem("chat_user_id");
    if (!userId) {
        userId = generateUserId();
        localStorage.setItem("chat_user_id", userId);
    }
    return userId;
}

```

```

    }
    return userId;
}

const userId = getUserId();
userInput.focus();

chatForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  const message = userInput.value.trim();
  const file = inputFile.files[0];

  if (!message && !file) return;

  if (message) {
    addMessage(message, "user");
  }
  userInput.value = "";
  userInput.focus();

  userInput.disabled = true;
  submitButton.disabled = true;
  inputFile.disabled = true;

  const typingIndicator = createTypingIndicator();
  chatWindow.appendChild(typingIndicator);
  chatWindow.scrollTop = chatWindow.scrollHeight;

  try {
    if (file) {
      await uploadFile(file);
      inputFile.value = "";
    }
    if (message) {
      const botReply = await fetchResponse(message);
      addMessage(botReply, "bot");
    }
  } catch (error) {
    addMessage("Error: Failed to get response from bot.", "bot");
    console.error(error);
  }

  chatWindow.removeChild(typingIndicator);
  userInput.disabled = false;
  submitButton.disabled = false;
  inputFile.disabled = false;
  userInput.focus();
});

```

```

async function uploadFile(file) {
  const formData = new FormData();
  formData.append("file", file);
  formData.append("user_id", userId);

  const response = await fetch("/api/upload", {
    method: "POST",
    body: formData,
  });

  if (!response.ok) {
    throw new Error("File upload failed");
  }
  const data = await response.json();
  addMessage(`(file sended: ${data.filename})`, "user");
}

function addMessage(text, sender) {
  const msg = document.createElement("div");
  msg.classList.add(
    "message",
    sender === "user" ? "user-message" : "bot-message"
  );
  msg.innerHTML = marked.parse(text);
  chatWindow.appendChild(msg);
  chatWindow.scrollTop = chatWindow.scrollHeight;
}

function createTypingIndicator() {
  const container = document.createElement("div");
  container.classList.add("typing-indicator");
  container.innerHTML = `
    <span></span>
    <span></span>
    <span></span>
  `;
  return container;
}

async function fetchResponse(userMsg) {
  const response = await fetch("/api/generate", {
    method: "POST",
    body: JSON.stringify({
      question: userMsg,
      user_id: userId,
    }),
    headers: {

```

```

    "Content-Type": "application/json",
  },
});

if (!response.ok) {
  throw new Error("Network response was not OK");
}

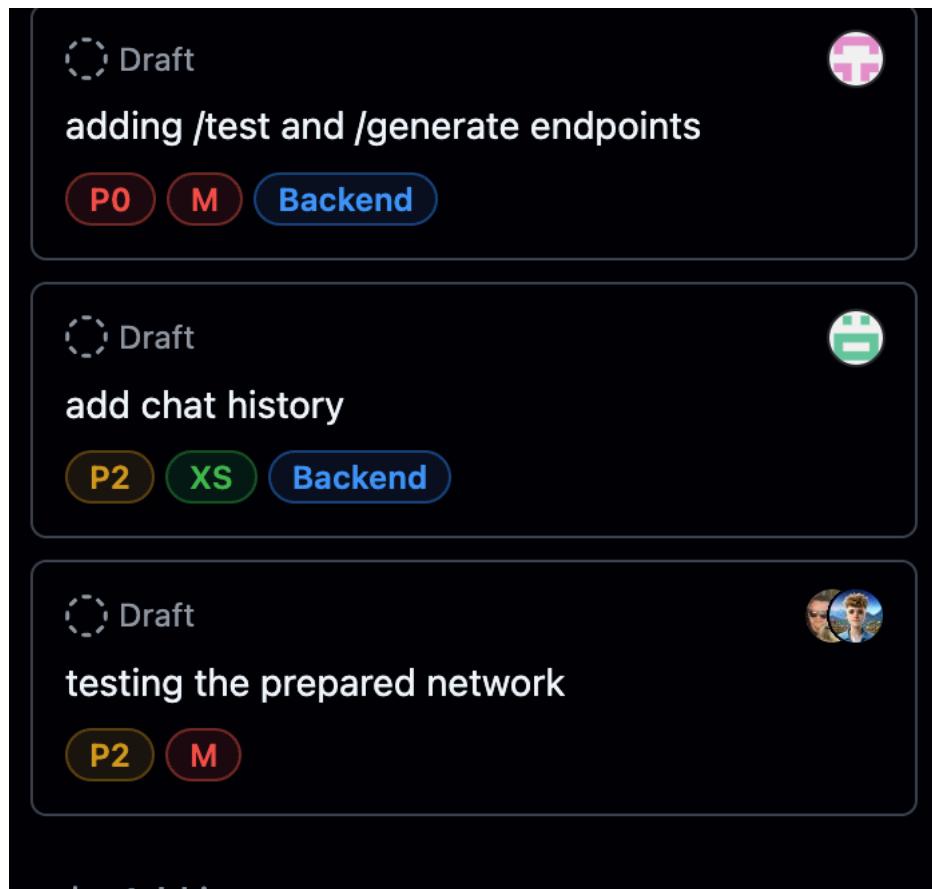
const data = await response.json();
return data.response || "(No response)";
}

```

Powyższy kod Java Script odpowiada za:

- Generowanie i trwałe przechowywanie unikalnego identyfikatora użytkownika (user_id) w localStorage w celu rozróżniania sesji i historii czatu.
- Obsługa asynchronicznego wysyłania wiadomości i plików do backendu z użyciem fetch oraz FormData (dla plików), z odpowiednim zarządzaniem stanem UI (blokowanie pól podczas oczekiwania na odpowiedź).
- Dynamiczne dodawanie wiadomości do okna czatu oraz animowany wskaźnik „pisania” podczas oczekiwania na odpowiedź AI.
- Obsługa odpowiedzi z backendu oraz błędów sieciowych, z informowaniem użytkownika o niepowodzeniach.

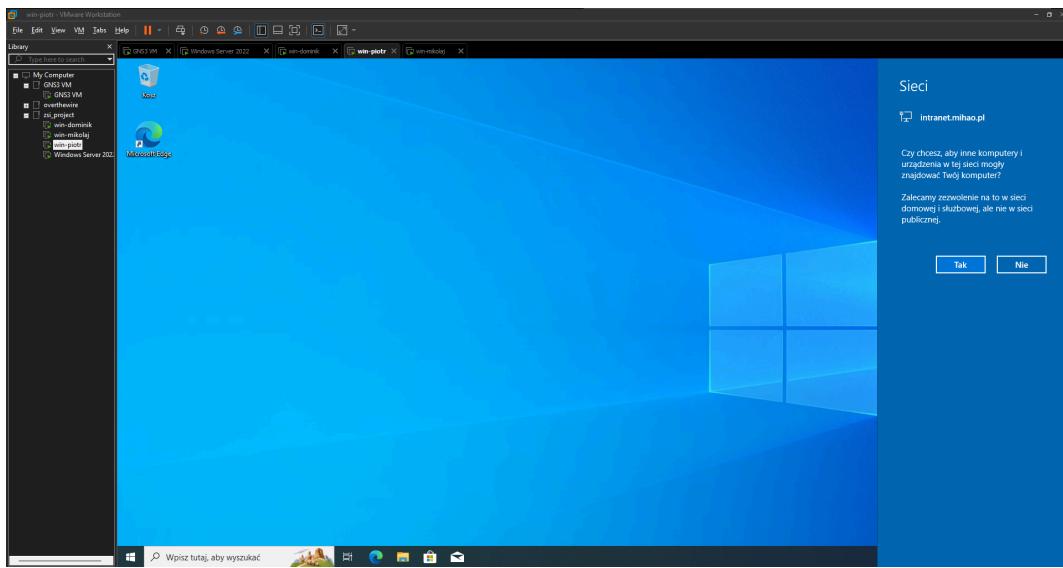
3.4 Sprint 4



Rysunek 26: Sprint 4

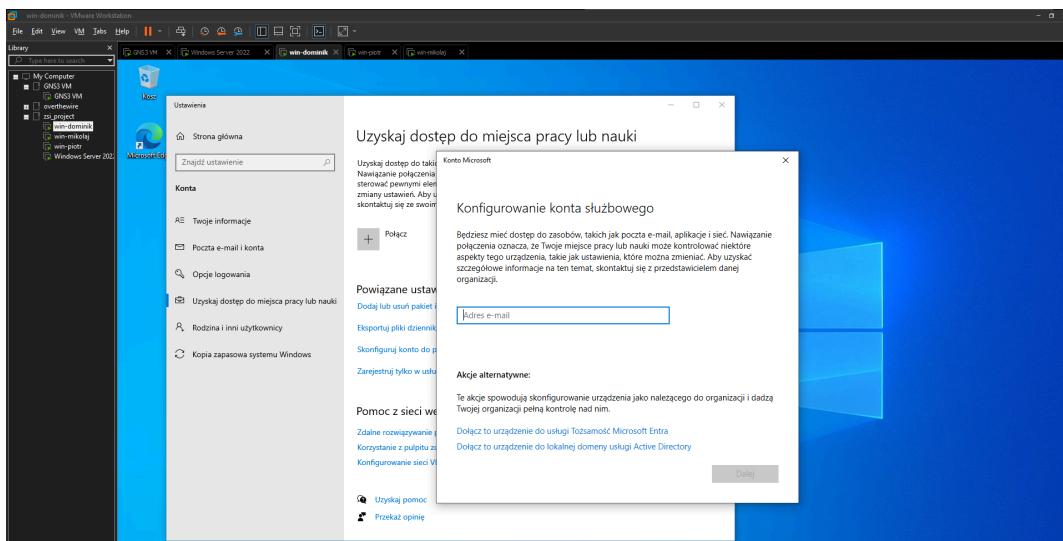
3.4.1 DevOps Team

By przetestować w pełni skonfigurowany asortyment serwerowy należało podłączyć do w pełni działającego systemu Active Directory dane Windowsy. Pokażmy system testu za pomocą wdrożenia `win-piotr`.



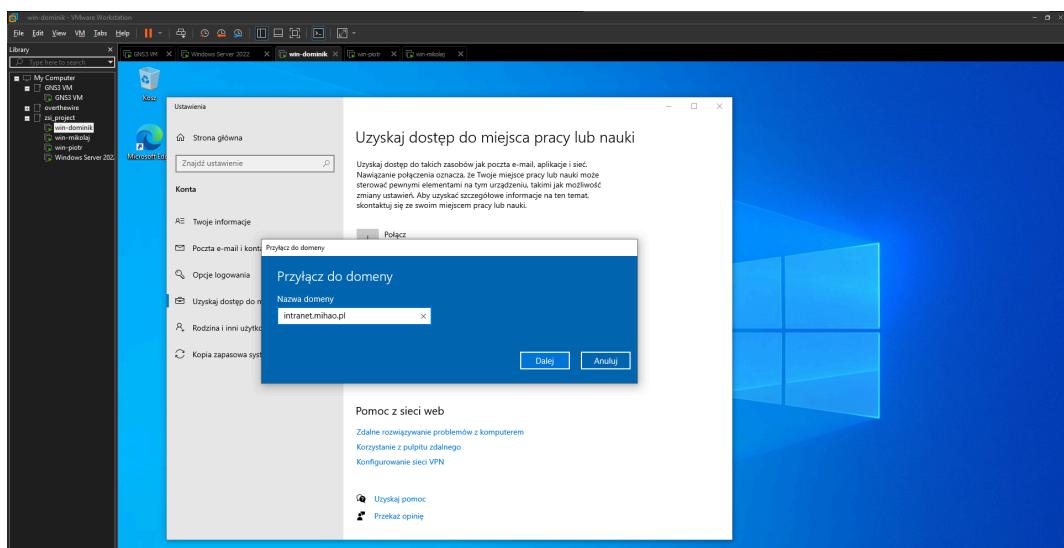
Rysunek 27: Wykrycie sieci intranet.mihao.pl

Pierwsze co pojawia się po starcie `win-piotr` to dany screen, jest to dowód na poprawne podłączenie danego urządzenia oraz konfiguracji **Windows Server**.



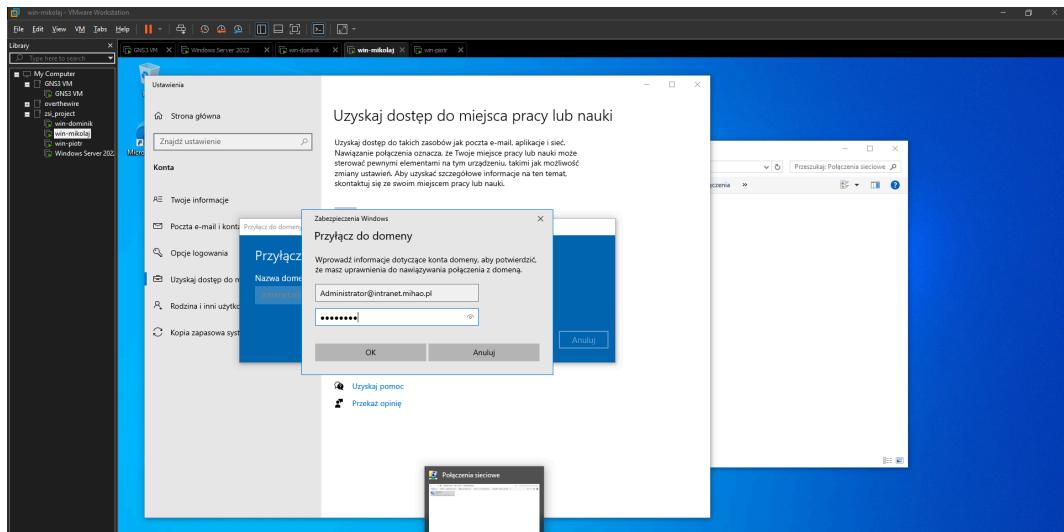
Rysunek 28: Podłączenie pod domenę

Następnie próbujemy podłączyć `win-piotr` pod domenę. Po przez wybranie opcji *Dolacz do urządzenia do lokalnej domeny usługi Active Directory*



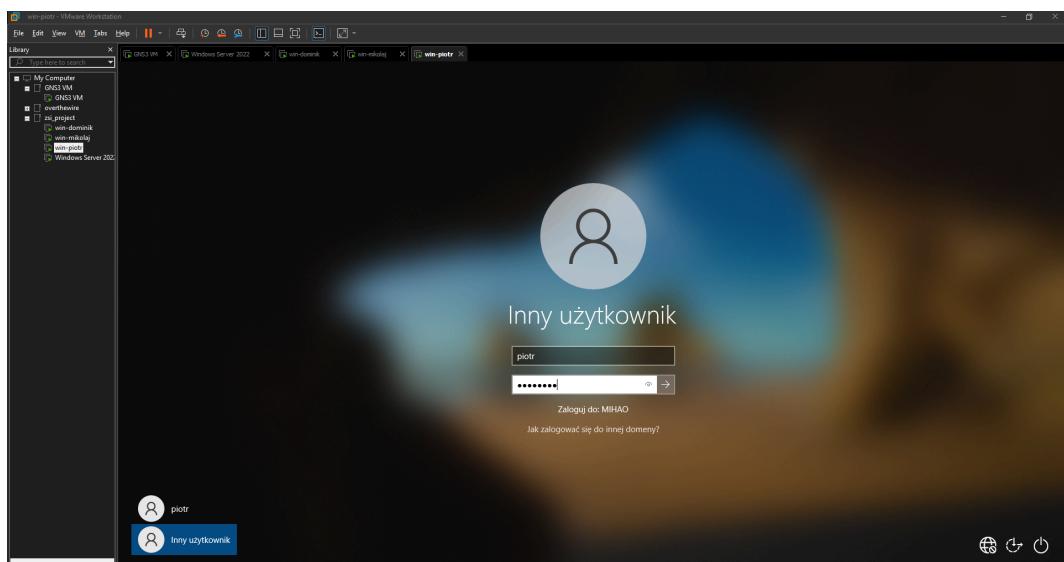
Rysunek 29: Podłączenie pod domenę - sprawdzenie

Wpisujemy nazwę domeny, do której chcemy podłączyć klienta `win-piotr`, w naszym przypadku będzie to `intranet.mihaop.pl`.



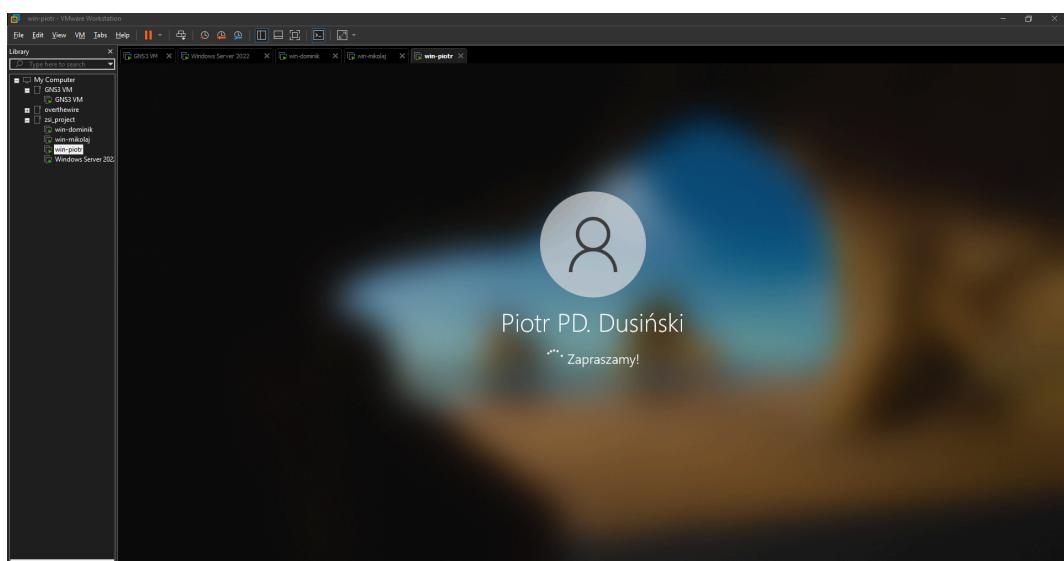
Rysunek 30: Podłączenie pod domenę - wdrożenie

Jak widać klient znalazł domenę, więc następnie logujemy `win_piotr` użytkownikiem, którym ma prawa do przyłączania klientów do domeny. W tym przypadku jest to Administrator. Później serwer poprosi nas o wykonanie restartu urządzenia.



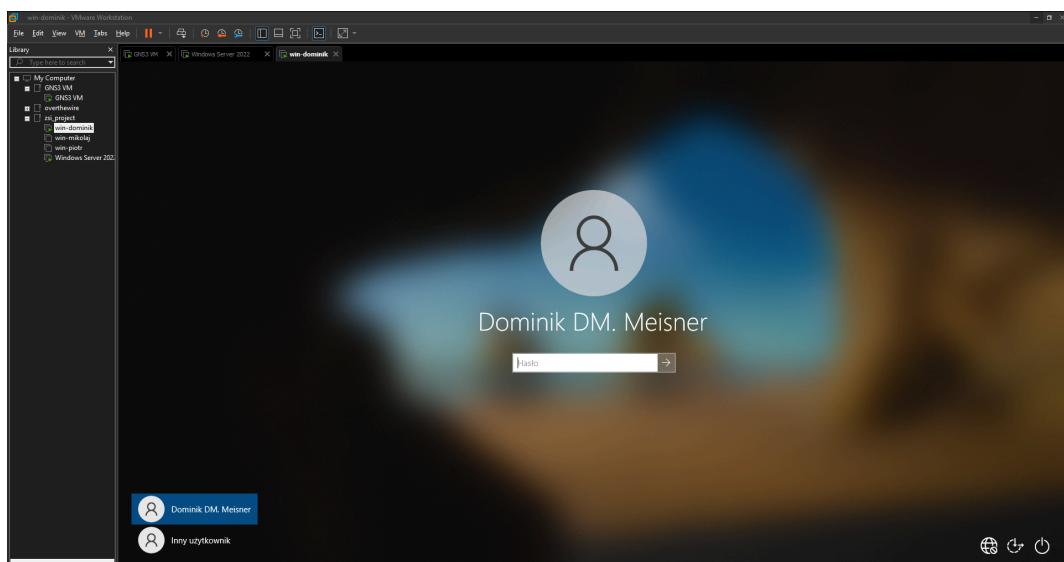
Rysunek 31: Podłączenie pod domenę - logowanie

Po restarcie, następuje próba logowania `win-piotr` jako użytkownika domeny MIHAO.



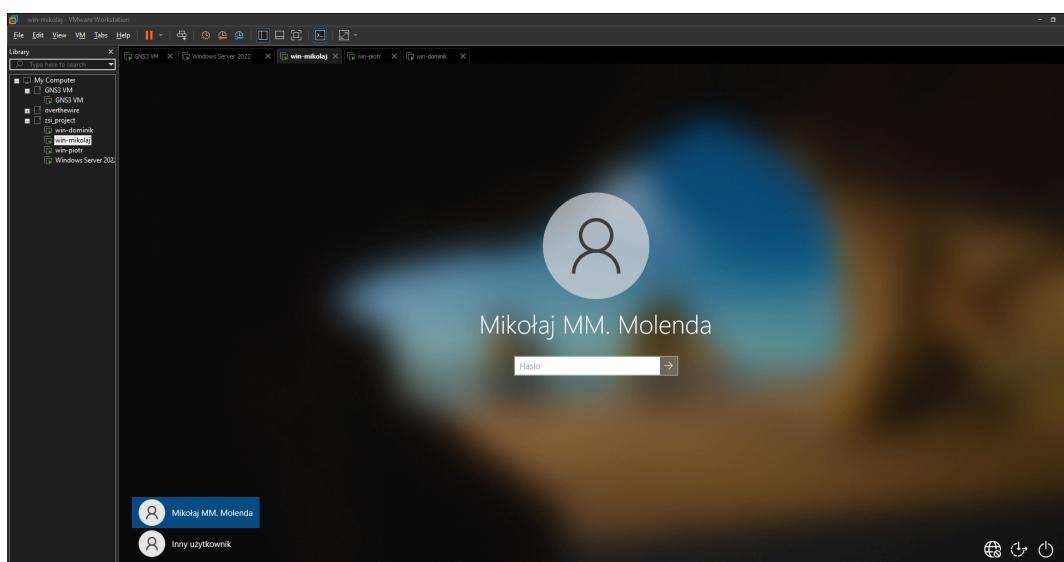
Rysunek 32: Podłączenie pod domenę - sukces

Logowanie `win-piotr` kończy się sukcesem. Komputer jest połączony do domeny. Poniżej są przykłady skonfigurowanych systemów Windows.



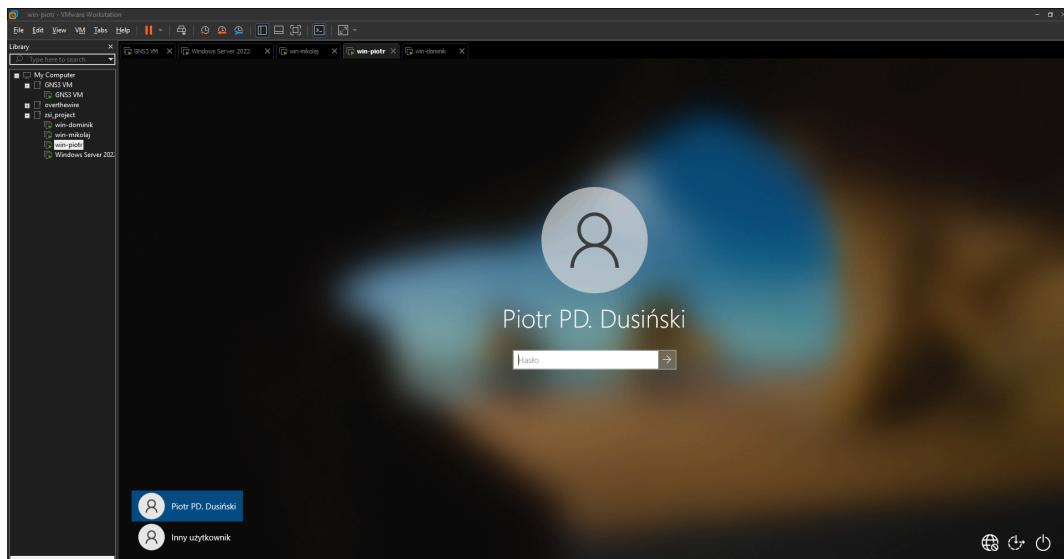
Rysunek 33: Podłączenie pod domenę - Dominik w domenie

Testujemy logowanie dla użytkownika Dominik



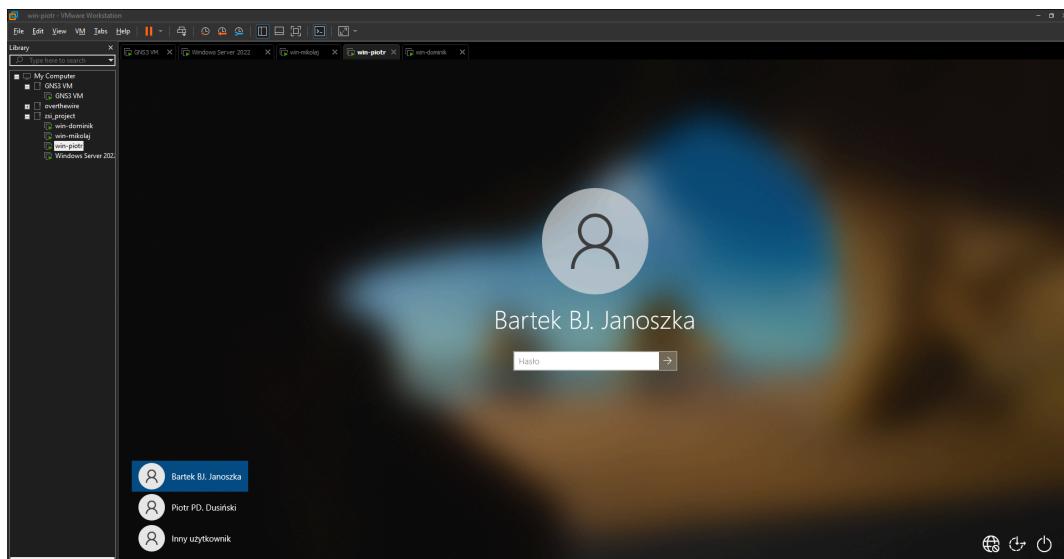
Rysunek 34: Podłączenie pod domenę - Mikołaj w domenie

Testujemy logowanie dla użytkownika Mikołaj



Rysunek 35: Podłączenie pod domenę - Piotr w domenie

Testujemy logowanie dla użytkownika Piotr



Rysunek 36: Podłączenie pod domenę - Bartek w domenie

Testujemy logowanie dla użytkownika Bartek

3.4.2 Dev Team

Po stworzeniu UI nadszedł czas na stworzenie backendu. Stworzyliśmy dwa endpointy: `/test` oraz `/generate`. Endpoint `/test` zwraca po prostu status `OK` informując nas, że backend działa.

```
@app.get("/test")
def test():
    return {"status": "OK"}
```

Wysyłając zapytanie do `/generate` należy podać pytanie (`question`) oraz id użytkownika (`user_id`). Najpierw na podstawie id użytkownika znaleziona zostanie historia czatu z tymże użytkownikiem, a następnie zostanie ona dołączona do finalnego zapytania jako kontekst. Całość zostaje wysłana do modelu, a odpowiedź, którą od niego otrzymujemy jest zapisywana w historii, a następnie zwracana w odpowiedzi na zapytanie, aby mogła zostać wyświetlona użytkownikowi.

Powyższe działanie backendowe zostało zawarte w pliku `app.py`.

```
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

model = OllamaLLM(model="gemma3:1b", base_url="http://ollama-app:11434")

template = """
You are an AI assistant. Your primary goal is to answer the user's
questions.
If the user's message includes content from a file, it will be
clearly demarcated with '--- Content of file [filename] ---' and
'--- End of file [filename] ---'.
You MUST treat the content within these markers as part of the
user's current query and use it to inform your response.
Do not just acknowledge the file; actively use its content if
relevant to the question.

You have access to the history of your chat with the user. Here it
is: {history}

User messages in the history might also contain file content
formatted in the same way.

Here is the user's current question (it may include file content as
```

```

described above):
{question}
"""

prompt = ChatPromptTemplate.from_template(template)
chain = prompt | model

chat_histories = {}


class GenerateQuery(BaseModel):
    question: str
    user_id: str


@app.get("/test")
def test():
    return {"status": "OK"}


@app.post("/generate")
def generate(query: GenerateQuery):
    history = chat_histories.get(query.user_id, [])
    context = "\n".join(history)

    response = chain.invoke({"history": context, "question": query.question})

    history.append(f"User: {query.question}")
    history.append(f"AI: {response}")
    chat_histories[query.user_id] = history

    return {"response": response, "history": history}

```

Powyższy kod można opisać w punktach w następujący sposób:

- FastAPI i CORS:

Aplikacja korzysta z frameworka FastAPI do budowy API oraz middleware CORS, który umożliwia komunikację z frontendem uruchomionym na dowolnym adresie (brak ograniczeń co do źródła żądań).

- Integracja z modelem LLM (Ollama):

Inicjalizowany jest obiekt `OllamaLLM`, który łączy się z serwisem modelu językowego (LLM) przez HTTP. Model ten będzie generował odpowiedzi na pytania użytkownika.

- Szablon promptu (`ChatPromptTemplate`):

Tworzony jest szablon promptu, który określa, jak model ma interpretować kontekst rozmowy i pytanie użytkownika. Prompt zawiera miejsce na historię czatu oraz aktualne pytanie.

- Łańcuch przetwarzania (**chain**):

Łańcuch (**chain**) łączy szablon promptu z modelem LLM, umożliwiając generowanie odpowiedzi na podstawie zadanego kontekstu i pytania.

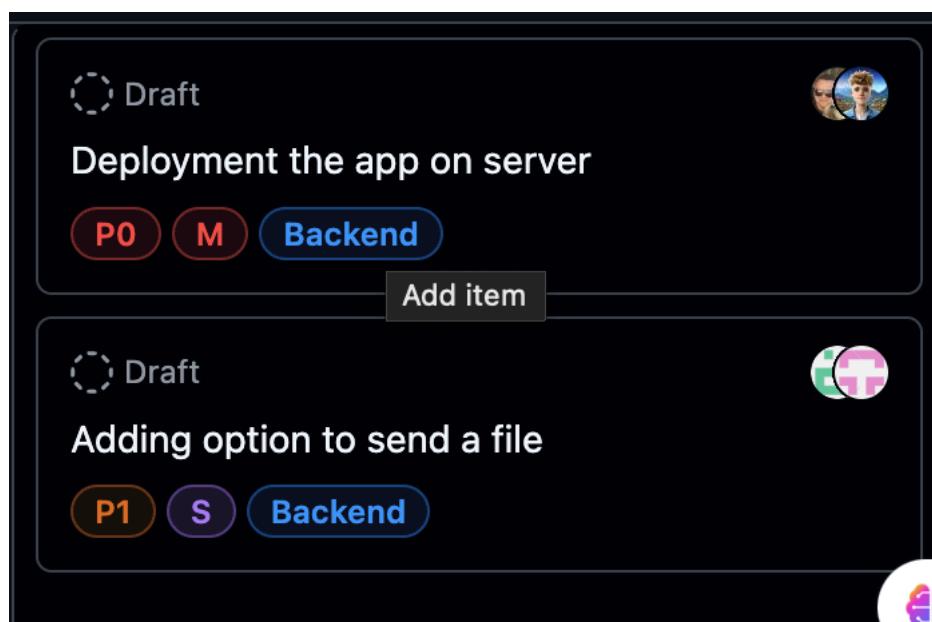
- Model danych (**GenerateQuery**):

Klasa **GenerateQuery** służy do walidacji danych wejściowych dla endpointu /**generate** (wymagane: **question** i **user_id**).

Dodatkowy mechanizm historii działa w następujący sposób:

- Słownik **chat_histories** przechowuje listy wiadomości dla każdego **user_id**.
- Przy każdym zapytaniu /**generate** historia użytkownika jest pobierana i przekazywana do modelu jako kontekst.
- Po wygenerowaniu odpowiedzi, pytanie i odpowiedź są dopisywane do historii.

3.5 Sprint 5

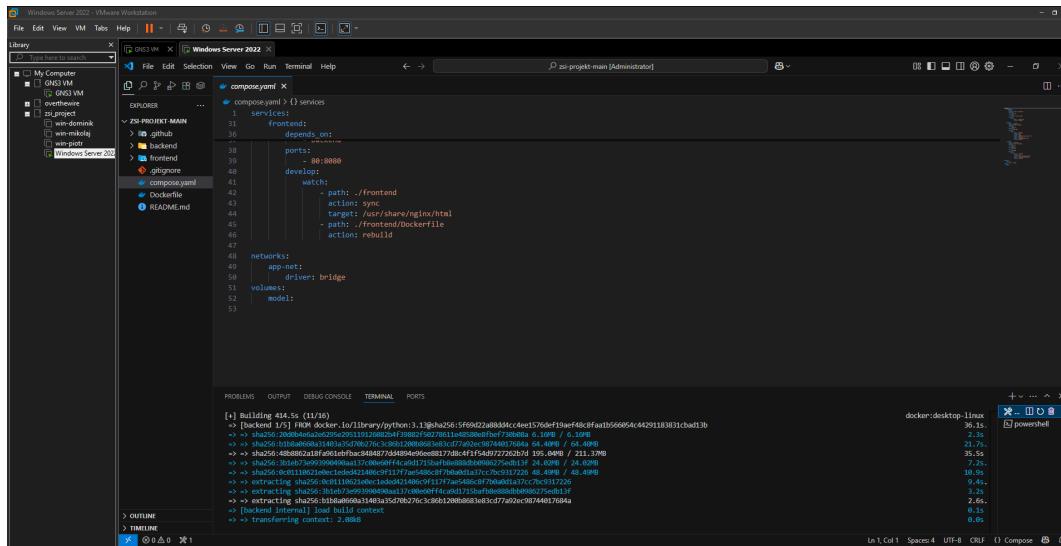


Rysunek 37: Ostatni Sprint

3.5.1 DevOps Team

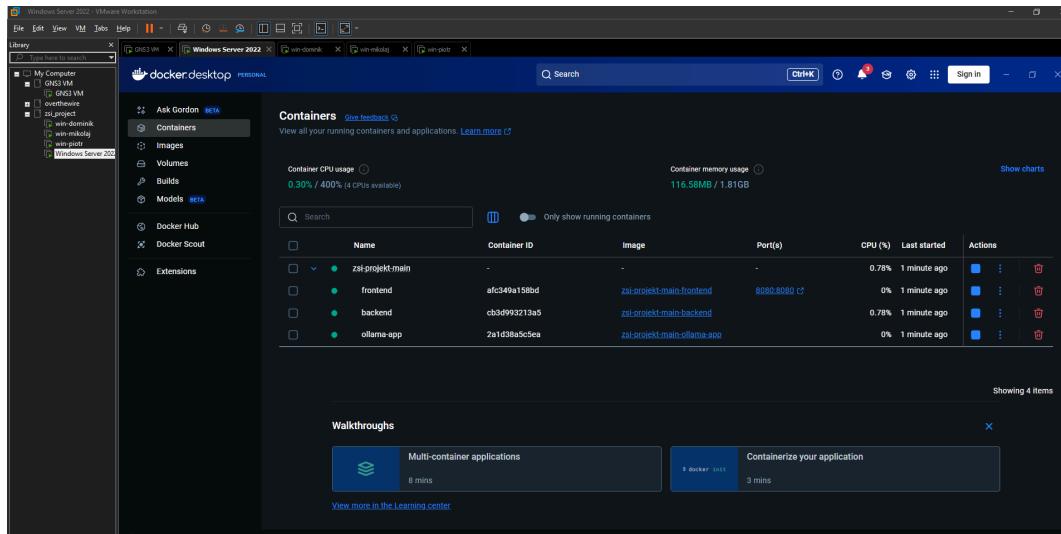
Ostatnim krokiem DevOps Team było skonfigurowanie aplikacji tak, by działa dla wszystkich Developerów. Przystąpiono do wdrożenia skontene-

ryzowanej aplikacji `log-analyzer`. Należało zbudować kontener według instrukcji `compose.yaml`.



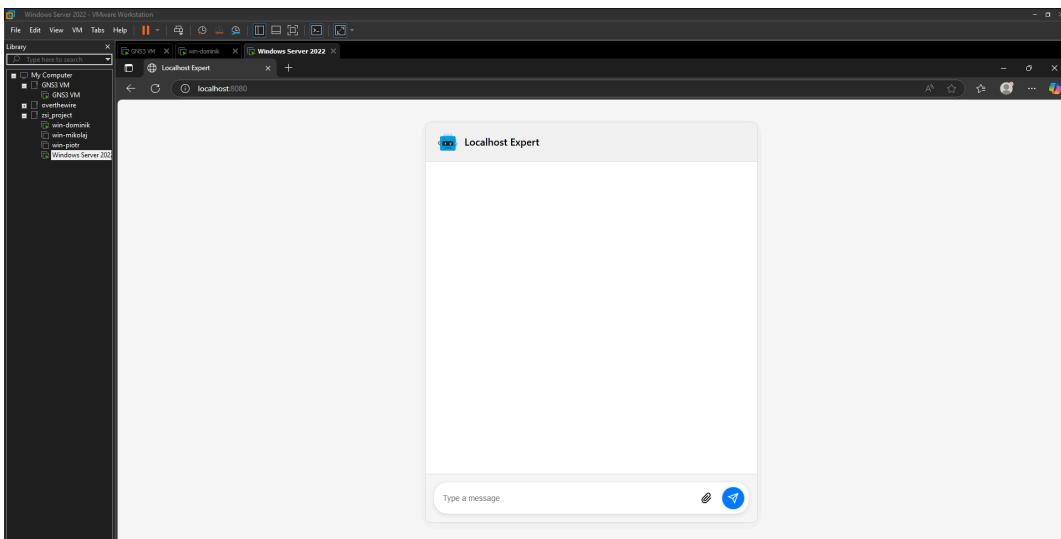
Rysunek 38: Budowanie kontenera

Po zbudowaniu kontenera upewniono się, że działa jak powinien.



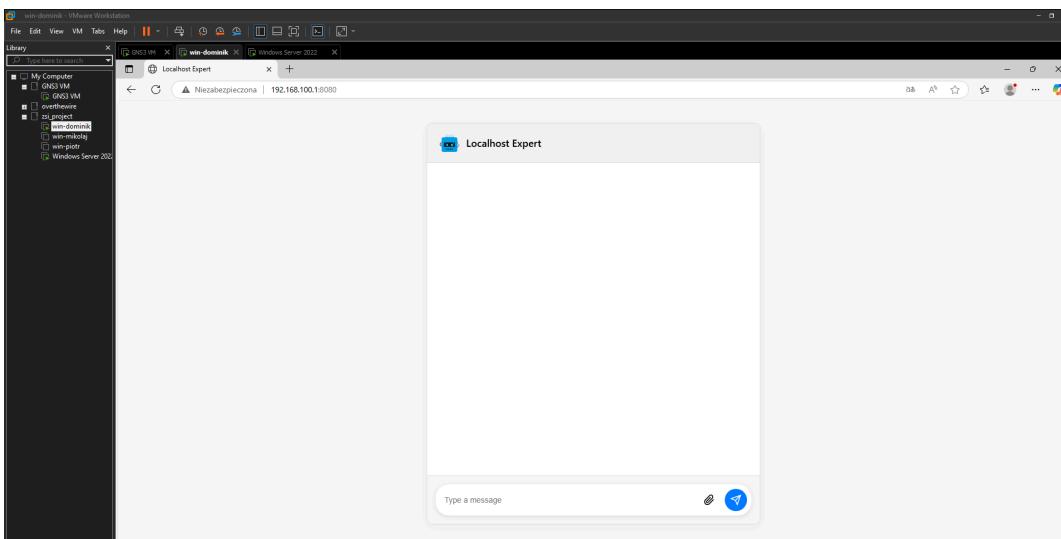
Rysunek 39: Kontener pracuje

Wiedząc, że kontener operuje na porcie 8080 należy przetestować jego działanie.



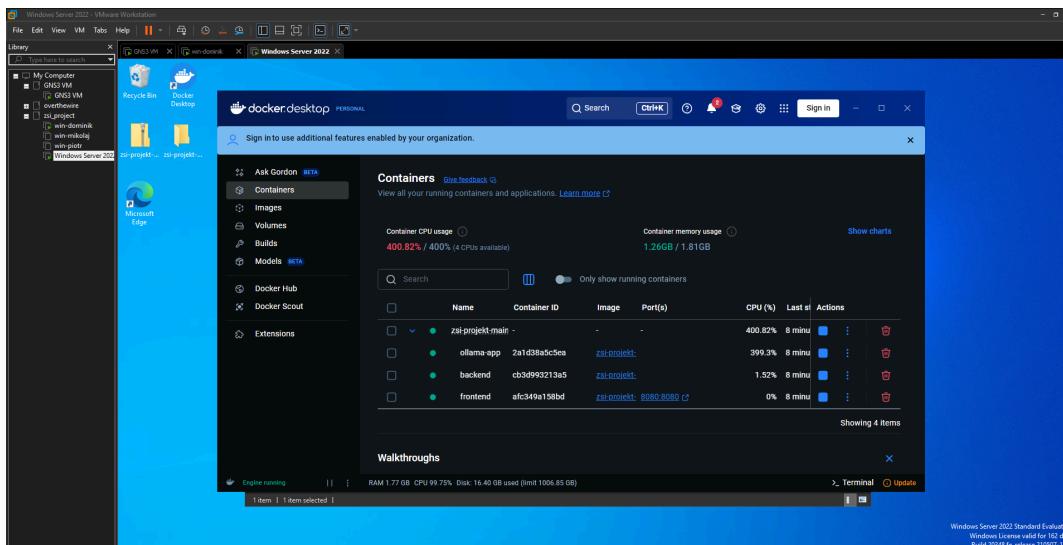
Rysunek 40: Kontener odpowiednio hostuje dla Windows Server aplikacje

Jeżeli Windows Server jest w stanie wyświetlić aplikacje na porcie 8080, to użytkownicy domeny również będą w stanie. Należy to sprawdzić. Logujemy się na **win-dominik** i wpisujemy w przeglądarce adres, który wskazuje na serwer oraz port.



Rysunek 41: Windows Server hostuje aplikacje dla użytkowników domeny

Windows Server dla wszystkich użytkowników jest teraz w stanie hostować oraz procesować wszelkie zapytania zwrócone do aplikacji. Oto przykład działania aplikacji po zapytaniu ze wszystkich 3 użytkowników topologii.



Rysunek 42: Przykład obciążenia aplikacji po 3 zapytaniach

3.5.2 Dev Team

Sprint 5 był już końcowym sprintem, dlatego skupiliśmy się na ostatnich szlifach oraz debugowaniu kodu. Dodaliśmy również ostatnią, ale jedną z najważniejszych funkcji naszego chatbota – możliwość przesyłania pliku do analizy. Za tę funkcjonalność odpowiada kod umożliwiający przesyłanie plików. Pliki są odczytywane po stronie frontendu za pomocą funkcji `readFileAsText()`

```
async function readFileAsText(file) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = (event) => resolve(event.target.result);
    reader.onerror = (error) => reject(error);
    reader.readAsText(file);
  });
}
```

Zawartość pliku jest zapisana w specjalnych znacznikach ułatwiając modelowi znajdowanie jego treści, a następnie, jeśli użytkownik wpisał jakąś wiadomość, zawartość pliku jest doklejana na jej koniec.

```
const fileBlock = `\\n\\n--- Content of file ${fileNameForDisplay}
---\\n${fileContent}\\n--- End of file ${fileNameForDisplay} ---`;

combinedMessageForBot = message
? `${message}${fileBlock}`
: fileBlock.trim();
```

4 Wnioski

4.1 Co nam się udało?

Udało nam się zasymulować środowisko biznesowe oraz logikę jego działania za pomocą oprogramowania jakim jest GNS3. Wdrożyć w nim nasza własną aplikację do analizy logów oraz umożliwić klientom domenowym korzystanie z niej. Do wdrożenia aplikacji wykorzystaliśmy technologię konteneryzacyjną, a konkretniej docker'a. Dodatkowo całość została przeprowadzona za pomocą sprintów w logice SACRUM, do tego celu wykorzystaliśmy GitHub Projects. Pozwoliło nam to na lepsze zapoznanie się z logika pracy w zespole. Wartym wspomnienia jest również mechanizm CI, który zaimplementowaliśmy jeszcze przed rozpoczęciem pracy nad aplikacją, użyliśmy do tego GitHub Actions.

4.2 Czego się nauczyliśmy?

- GitHub Projects - Nauczyliśmy się logiki pracy w zespole oraz narzędzi do tego wykorzystywanych
- GitHub Actions - Zapoznaliśmy się z koncepcja Continous Integration (CI) oraz wdrożyliśmy prosty skrypt, który sprawdza poprawność działania kontenerów po dokonaniu zmian w repozytorium.
- GNS3 - odświeżyliśmy oraz pogłębiliśmy wiedzę w kontekście narzędzia jakim jest GNS3. Przy tej okazji również mogliśmy odświeżyć wiedzę na temat sieci komputerowych oraz wirtualizacji
- Docker - odświeżyliśmy oraz pogłębiliśmy wiedzę w kontekście narzędzia jakim jest docker, jak i samej konteneryzacji.
- Nauczyliśmy się logiki działania infrastruktury informatycznej w kontekście biznesowym.
- Windows Server oraz jego usługi + domena Active Directory - Nauczyliśmy się konfiguracji kilku wybranych usług systemu Windows Server oraz logiki działania domeny Active Directory

4.3 Potencjał rozwoju

- implementacja Continous Deployment - jeżeli aplikacja otrzyma aktualizacje zmiana ta zostanie natychmiast wdrożona na serwer.
- utworzenie clustra w technologii K8's - ograniczenie zużycia zasobów serwera przez aplikacji oraz automatyczne reparacje. Cluster również byłby użyty do wcześniej wspomnianego CI/CD
- optymalizacja aplikacji

5 Bibliografia

- [1] „Internet”.