

Uniwersytet WSB MERITO w Gdańsku

Wydział Informatyki i Nowych Technologii



## Projekt zaliczeniowy

Przetwarzanie Rozproszone

Roman Gabriel Glegoła

Nr albumu: 34929

Gdańsk, 2024

# Oświadczenie

Ja, niżej podpisany Roman Gabriel Glegoła, student Wydziału Informatyki i Nowych Technologii Uniwersytet WSB MERITO w Gdańsku, oświadczam, że przedłożoną pracę zaliczeniową napisałem samodzielnie.

Oświadczam również, że:

- wszystkie informacje zawarte w niniejszej pracy, które zostały zaczerpnięte z innych źródeł, zostały odpowiednio zaznaczone i zacytowane,
- niniejsza praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Jestem świadomy odpowiedzialności prawnej za fałszywe oświadczenie.



.....  
Gdańsk, dnia 03.12.2023

# 1 Opis projektu

Projekt zaliczeniowy na ćwiczenia "Przetwarzanie Rozproszone" przedstawia aplikację czat w formie serwerów REST API w C# i Python oraz klientów w C#, Python, Powershell.

Forma zaliczenia programowanie rozproszone.

Język programowania c#.

Środowisko: Visual Studio

W ramach zaliczenia należy utworzyć projekt wykorzystujący programowanie rozproszone.

Kryteria zaliczeniowe, które należy uwzględnić w projekcie:

Ocena dst:

- projekt powinien działać w oparciu o RestApi lub bibliotekę SignalR z wykorzystaniem co najmniej dwóch klientów utworzonych w różnych technologiach ((WinForms, Wpf), python (Flask, Django), Razor, Angular, React, aplikacja mobilna ..... )
- projekt należy umieścić na platformie moodle przed ostatnimi zajęciami, w przygotowanej przez prowadzącego lokalizacji.
- do projektu należy dołączyć plik pdf. Nazwą pliku powinno być nazwisko i imię studenta
- zawartość pliku

Nazwa projektu:
Opis projektu:
Omówienie koncepcji projektu
Zamieszczenie przykładowych obrazów przedstawiających funkcjonowanie aplikacji

## 2 Omówienie koncepcji projektu

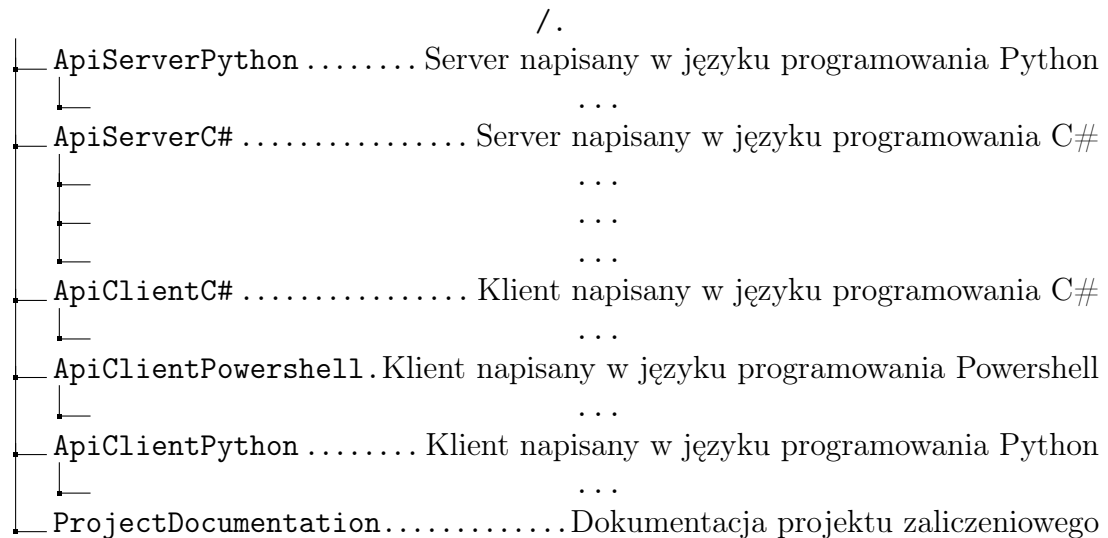
### 2.1 Wstęp

Koncepcja przewodząca powyższemu projektowi polega na zaliczeniu przedmiotu Przetwarzanie Rozproszone na ocenę przynajmniej dostateczną. Obrałem technologie Python, Powershell i C# dlatego, że wymagania wykładowcy obejmują użycie przynajmniej dwóch różnych technologii do utworzenia klienta, a ja wybrałem trzy. Niestety odrzuciłem zakładanie serwera oraz klienta w języku programowania haskell ponieważ ustawianie środowiska dla tego języka programowania może być problematyczne dla wykładowcy skutkując oceną niedostateczną.

## 3 Omówienie projektu

### 3.1 Drzewo folderów projektu

Tabela 1: Drzewo struktury katalogów



### 3.2 Dlaczego dwa serwery?

Na załączonym powyżej drzewie folderu projektów na pierwszy rzut oka widać, że pomimo jednego wymaganego serwera API REST w języku C# jest zrobiony dodatkowo serwer API REST w języku Python. Zostało to utworzone ze względu na moje komercyjne doświadczenie w korzystaniu z języka programowania Python. Umożliwiło mi to stworzenie w 100% działającego serwera, który mogę od razu wykorzystać w celach testowych.

### 3.3 Dlaczego trzy klienty?

Tak samo jak na powyżej załączonym drzewie folderu projektów widać, że pomimo dwóch wymaganych klientów API REST zrobiony jeden dodatkowy klient. Użyłem języków C#, Python i Powershell zważywszy na to, że w tych pierwszych dwóch językach są już napisane serwery, a ów trzeci język jest muszlą mocy pozwalającą na niesamowite i błyskawiczne czynności zarówno na środowisku Windows jak i Linux przyćmiewając możliwości oferowane przez powłokę linux.

## 4 Uruchomienie serwerów REST API

### 4.1 Setup i uruchomienie Serwera napisanego w Python

Aby przygotować i uruchomić serwer FastAPI napisany w języku programowania Python manualnie, należy wykonać następujące kroki:

1. Upewnij się, że Python w wersji 3.x. jest zainstalowany w systemie.
2. Przejdź do katalogu projektu serwera, gdzie znajduje się plik `.py`:

```
cd sciezka/do/projektu/ApiServerPython
```

3. Zainstaluj potrzebny pakiet `requests` używając `pip`:

```
pip install fastapi uvicorn pydantic
```

4. Uruchom serwer używając `uvicorn`:

```
uvicorn api_server:app  
--host localhost --port 1337 --reload
```

### 4.2 Kompilacja i Uruchamianie Serwera C#

Aby skompilować i uruchomić serwer ASP.NET Core manualnie, należy wykonać następujące kroki:

1. Otwórz terminal (np. CMD, PowerShell lub terminal w systemie Linux/Mac).
2. Przejdź do katalogu, gdzie znajduje się plik `.csproj` serwera projektu:

```
cd sciezka/do/ApiServerC#
```

3. Zbuduj projekt używając narzędzia `dotnet`:

```
dotnet build
```

4. Uruchom serwer używając:

```
dotnet run
```

## 5 Uruchomienie klientów REST API

### 5.1 Kompilacja i Uruchamianie Klienta C#

Aby skompilować i uruchomić klienta API napisanego w C#, wykonaj następujące kroki:

1. Otwórz terminal (np. CMD, PowerShell lub terminal w systemie Linux/Mac).
2. Przejdź do katalogu projektu klienta, gdzie znajduje się plik `.csproj`:

```
cd sciezka/do/projektu/ApiClientCSharp
```

3. Zbuduj projekt używając narzędzia `dotnet`:

```
dotnet build
```

4. Uruchom aplikację:

```
dotnet run
```

### 5.2 Uruchamianie Klienta Chatu Python

Aby przygotować i uruchomić klienta Rest API napisany w języku programowania Python manualnie, należy wykonać następujące kroki:

1. Upewnij się, że Python w wersji 3.x. jest zainstalowany w systemie.
2. Przejdź do katalogu projektu klienta, gdzie znajduje się plik `.py`:

```
cd sciezka/do/projektu/ApiClientPython
```

3. Zainstaluj potrzebny pakiet `requests` używając `pip`:

```
pip install requests
```

4. Uruchom skrypt w terminalu:

```
python python_api_client.py
```

## 5.3 Uruchamianie Klienta Chatu Powershell

1. Upewnij się, że PowerShell jest zainstalowany w systemie.
2. Otwórz terminal PowerShell (lub pwsh w systemie Linux/Mac).
3. Przejdź do katalogu projektu klienta, gdzie znajduje się plik `.ps1`:

```
cd sciezka/do/projektu/ApiClientPowershell
```

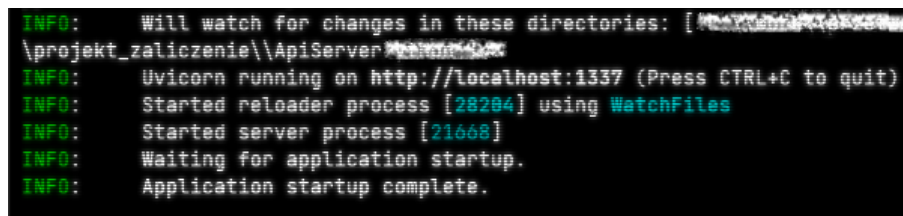
4. Uruchom skrypt w terminalu:

```
.\powershell_api_client.ps1
```

## 6 Używanie serwerów REST API

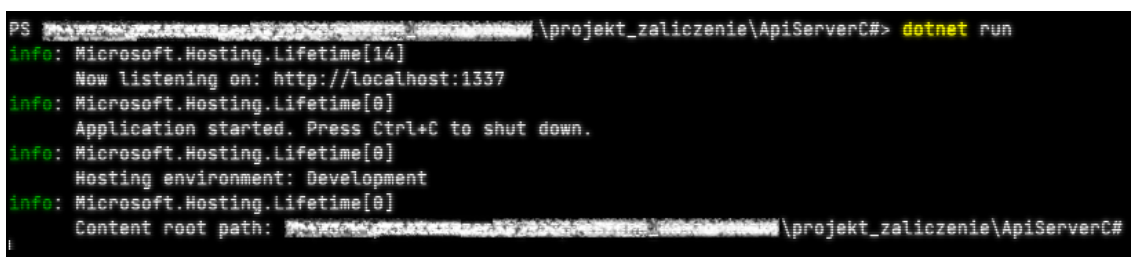
### 6.1 Używanie serwera Rest API

Używanie serwera Rest API jest wyjątkowo proste. Nie trzeba nic robić. Serwer sam zarządza komunikacją na podstawie zmiennych i wpisów, które znajdują się wewnątrz kodu. Nie ma tutaj większej różnicy między serwerem napisanym w Python i C# gdyż oba zostały napisane tak aby dla użytkownika działały podobnie.



```
INFO: Will watch for changes in these directories: [...]  
INFO: Uvicorn running on http://localhost:1337 (Press CTRL+C to quit)  
INFO: Started reloader process [28284] using WatchFiles  
INFO: Started server process [21668]  
INFO: Waiting for application startup.  
INFO: Application startup complete.
```

Rysunek 1: Ekran po uruchomieniu serwera Python



```
PS ...\\projekt_zaliczenie\\ApiServerC#> dotnet run  
info: Microsoft.Hosting.Lifetime[14]  
Now listening on: http://localhost:1337  
info: Microsoft.Hosting.Lifetime[0]  
Application started. Press Ctrl+C to shut down.  
info: Microsoft.Hosting.Lifetime[0]  
Hosting environment: Development  
info: Microsoft.Hosting.Lifetime[0]  
Content root path: ...\\projekt_zaliczenie\\ApiServerC#
```

Rysunek 2: Ekran po uruchomieniu serwera C#

## 6.2 Logowanie na serwerach Rest API

Na poniższym przykładzie na zdjęciach możemy zauważyć, że oba serwery logują poprawnie dane w swoim okienku terminala.. O ile terminal C# zostawia więcej informacji tworzących szum o tyle wszystkie informacje potrzebne tam się znajdują.

```
INFO:      Uvicorn running on http://localhost:1337 (Press CTRL+C to quit)
INFO:      Started reloader process [28204] using WatchFiles
INFO:      Started server process [21668]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:api_server:[2024-01-21 02:04:41] TestUser: Test message
INFO:      127.0.0.1:12265 - "POST /chat/send-message/ HTTP/1.1" 200 OK
INFO:      127.0.0.1:12268 - "GET /chat/get-messages/ HTTP/1.1" 200 OK
```

Rysunek 3: Ekran logowań serwera Python

```
Info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 POST http://localhost:1337/chat/send-message/ - application/json 48
Info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'ApiServerCSharp.Controllers.ChatController.SendMessage (ApiServerC#)'
Info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[102]
      Route matched with {action = "SendMessage", controller = "Chat"}. Executing controller action with signature Microsoft.AspNetCore.Mvc.IActionResult SendMessage(ApiServerCSharp.Models.Message) on controller ApiServerCSharp.Controllers.ChatController (ApiServerC#).
Info: Microsoft.AspNetCore.Mvc.Infrastructure.ObjectResultExecutor[1]
      Executing OkObjectResult, writing value of type '<>f__AnonymousType0`1[[System.String, System.Private.CoreLib, Version=8.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e]]'.
Info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[105]
      Executed action ApiServerCSharp.Controllers.ChatController.SendMessage (ApiServerC#) in 34.1546ms
Info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'ApiServerCSharp.Controllers.ChatController.SendMessage (ApiServerC#)'
Info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 POST http://localhost:1337/chat/send-message/ - 200 - application/json;+charset=utf-8 85.2946ms
Info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET http://localhost:1337/chat/get-messages/ - - -
Info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'ApiServerCSharp.Controllers.ChatController.GetMessages (ApiServerC#)'
Info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[102]
      Route matched with {action = "GetMessages", controller = "Chat"}. Executing controller action with signature Microsoft.AspNetCore.Mvc.IActionResult GetMessages() on controller ApiServerCSharp.Controllers.ChatController (ApiServerC#).
Info: Microsoft.AspNetCore.Mvc.Infrastructure.ObjectResultExecutor[1]
      Executing OkObjectResult, writing value of type 'System.Collections.Generic.List`1[[ApiServerCSharp.Models.Message, ApiServerC#, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]]'.
Info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[105]
      Executed action ApiServerCSharp.Controllers.ChatController.GetMessages (ApiServerC#) in 7.1819ms
Info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'ApiServerCSharp.Controllers.ChatController.GetMessages (ApiServerC#)'
Info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET http://localhost:1337/chat/get-messages/ - 200 - application/json;+charset=utf-8 9.6173ms
```

Rysunek 4: Ekran logowań serwera C#



## 7 Używanie klienta REST API

### 7.1 Ekran powitalny

Użytkownik po uruchomieniu klienta chatu i jest przywitany komunikatem "Welcome to the Chat Client!". "Enter your username:" gdzie jest proszony o wpisanie nazwy użytkownika jaką będzie się posługiwał

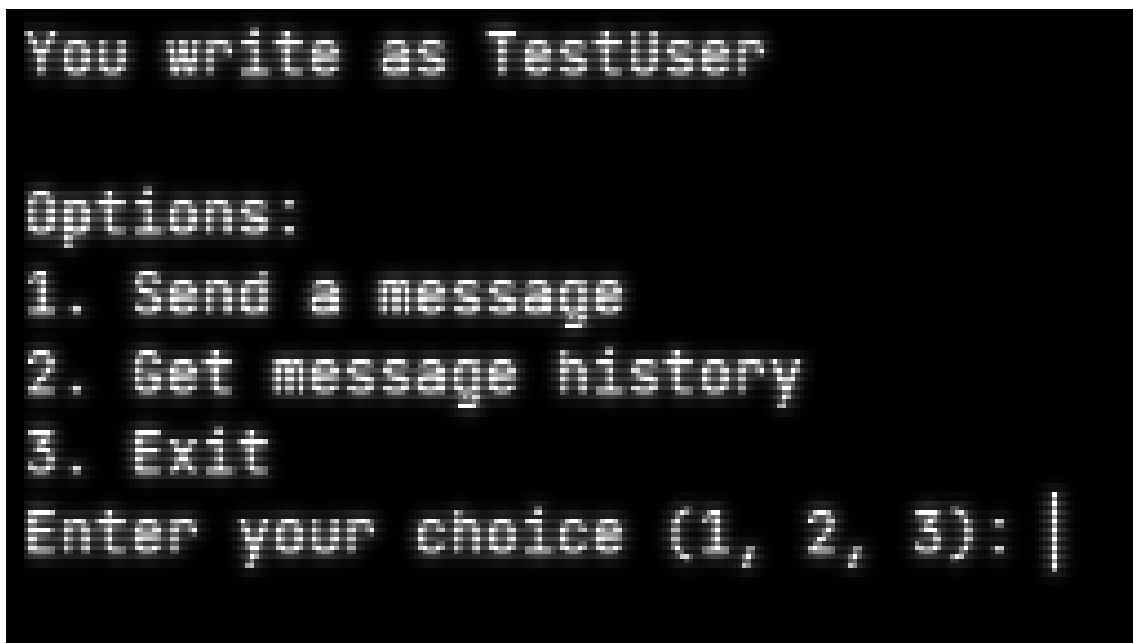


```
Welcome to the Chat Client!  
Enter your username: TestUser|
```

Rysunek 5: Ekran powitalny klienta REST API

### 7.2 Ekran menu

Po wprowadzeniu nazwy użytkownika, użytkownik widzi główne menu z opcjami. Opcje do wyboru to "Send a message", "Get message history" i "Exit".



```
You write as TestUser  
  
Options:  
1. Send a message  
2. Get message history  
3. Exit  
Enter your choice (1, 2, 3): |
```

Rysunek 6: Ekran menu klienta REST API

### 7.3 Ekran wysyłania wiadomości

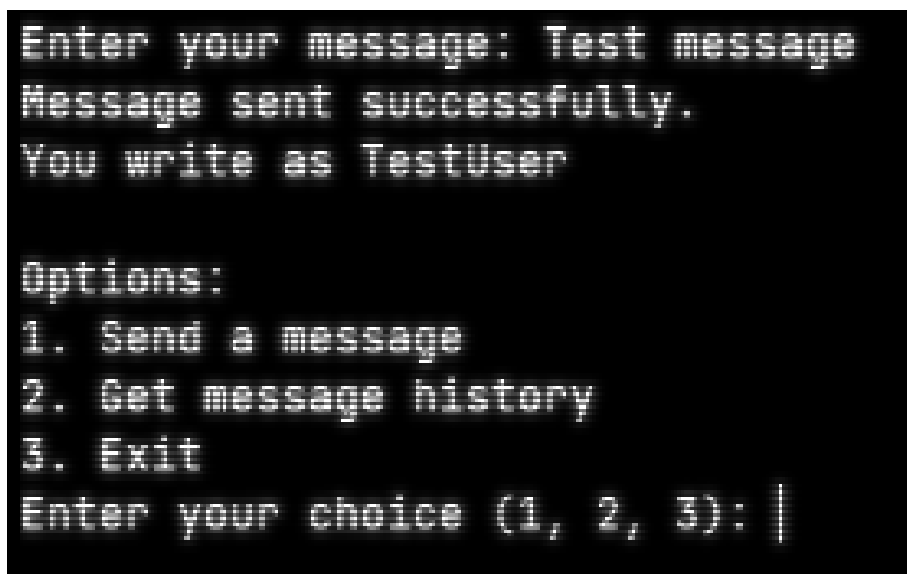
W oknie wysyłania wiadomości, użytkownik zostanie poproszony o wprowadzenie treści wiadomości.



Rysunek 7: Ekran wysyłania wiadomości klienta REST API

### 7.4 Ekran po wysłaniu wiadomości

Po wpisaniu i wysłaniu wiadomości, użytkownik otrzymuje potwierdzenie "Message sent successfully"



Rysunek 8: Ekran po wysłaniu wiadomości klienta REST API

## 7.5 Ekran odczytywania wiadomości

Gdy użytkownik wybierze "Get message history", na ekranie wyświetlona zostanie historia wcześniej wysłanych wiadomości. Każda wiadomość jest wyświetlana wraz z nazwą użytkownika i treścią.

```
TestUser: Test message
TestUser: Test message content #2
You write as TestUser

Options:
1. Send a message
2. Get message history
3. Exit
Enter your choice (1, 2, 3): |
```

Rysunek 9: Ekran odczytywania wiadomości klienta REST API

## 7.6 Obsługa błędnego wyboru

W każdym momencie, jeśli użytkownik wprowadzi nieprawidłową opcję, system wyświetli komunikat "Invalid choice. Please enter 1, 2, or 3.", co zmusza użytkownika do podjęcia ponownej próby wyboru.

```
Invalid choice. Please enter 1, 2, or 3.
You write as TestUser

Options:
1. Send a message
2. Get message history
3. Exit
Enter your choice (1, 2, 3): |
```

Rysunek 10: Ekran obsługi błędnego wyboru klienta REST API

## 7.7 Zakończenie sesji

Jeśli użytkownik zdecyduje się wyjść, wybierając "Exit", klient chatu zakończy działanie, informując użytkownika komunikatem "Exiting chat client.", co można zaobserwować po wykonaniu tej czynności

# 8 Zakończenie

## 8.1 Czego się nauczyłem

Bardzo się cieszę, że mogłem poznać na zajęciach bibliotekę SignalR w związku z tym, że moja kariera zawodowa związana jest z warstwą sieciową internetu to jednak nie widzę przyszłości gdzie będę korzystał z tej biblioteki podczas gdy rynek oferuje o wiele prostsze rozwiązania oparte na api REST, SOAP, WebSocket i inne które można błyskawicznie zastosować.

## 8.2 O projekcie

Powyższy projekt uważam za ciekawy choć przyznaję, że wybrałem projekt czatu gdyż taki projekt był wykonywany podczas zajęć. Zastanawiałem się nad projektem dokonującym autoryzacji lub innym command&control jednakże ostatecznie uznałem, że nie powinienem eksperymentować na zaliczenie, a ambitne projekty zostawić sobie na githuba.

## 8.3 O ćwiczeniach

Jeśli chodzi o ćwiczenia to nie mam nic do zarzucenia prowadzącemu. Trochę żałuję, że zajęcia nie przystawały w moim mniemaniu do nazwy "przetwarzanie rozproszone" gdyż to sugeruje bardziej chmurowe kwestie aczkolwiek to nie jest problem. Zadaniem uczelni jest pokazać horyzonty, a to już w kwestii samego studenta jest to aby obrać kierunek w którym dalej będzie się specjalizował.

# Spis treści

<b>Strona Tytułowa</b>	<b>1</b>
<b>1 Opis projektu</b>	<b>2</b>
<b>2 Omówienie koncepcji projektu</b>	<b>2</b>
2.1 Wstęp . . . . .	2
<b>3 Omówienie projektu</b>	<b>3</b>
3.1 Drzewo folderów projektu . . . . .	3
3.2 Dlaczego dwa serwery? . . . . .	3
3.3 Dlaczego trzy klienty? . . . . .	3
<b>4 Uruchomienie serwerów REST API</b>	<b>4</b>
4.1 Setup i uruchomienie Serwera napisanego w Python . . . . .	4
4.2 Kompilacja i Uruchamianie Serwera C# . . . . .	4
<b>5 Uruchomienie klientów REST API</b>	<b>5</b>
5.1 Kompilacja i Uruchamianie Klienta C# . . . . .	5
5.2 Uruchamianie Klienta Chatu Python . . . . .	5
5.3 Uruchamianie Klienta Chatu Powershell . . . . .	6
<b>6 Używanie serwerów REST API</b>	<b>6</b>
6.1 Używanie serwera Rest API . . . . .	6
6.2 Logowanie na serwerach Rest API . . . . .	7
<b>7 Używanie klienta REST API</b>	<b>8</b>
7.1 Ekran powitalny . . . . .	8
7.2 Ekran menu . . . . .	8
7.3 Ekran wysyłania wiadomości . . . . .	9
7.4 Ekran po wysłaniu wiadomości . . . . .	9
7.5 Ekran odczytywania wiadomości . . . . .	10
7.6 Obsługa błędnego wyboru . . . . .	10
7.7 Zakończenie sesji . . . . .	11

<b>8 Zakończenie</b>	<b>11</b>
8.1 Czego się nauczyłem . . . . .	11
8.2 O projekcie . . . . .	11
8.3 O ćwiczeniach . . . . .	11
<b>Spis treści</b>	<b>12</b>
<b>Spis literatury</b>	<b>14</b>
<b>Spis ilustracji</b>	<b>15</b>
<b>Spis tabel</b>	<b>16</b>

## Literatura

## Spis rysunków

0	Logo uczelni WSB . . . . .	1
0	Tekst zadania zaliczeniowego . . . . .	2
1	Ekran po uruchomieniu serwera Python . . . . .	6
2	Ekran po uruchomieniu serwera C# . . . . .	6
3	Ekran logowań serwera Python . . . . .	7
4	Ekran logowań serwera C# . . . . .	7
5	Ekran powitalny klienta REST API . . . . .	8
6	Ekran menu klienta REST API . . . . .	8
7	Ekran wysyłania wiadomości klienta REST API . . . . .	9
8	Ekran po wysłaniu wiadomości klienta REST API . . . . .	9
9	Ekran odczytywania wiadomości klienta REST API . . . . .	10
10	Ekran obsługi błędnego wyboru klienta REST API . . . . .	10



## Spis tabel

1	Drzewo struktury katalogów . . . . .	3
---	--------------------------------------	---