

**Uniwersytet Warszawski**  
Wydział Matematyki, Informatyki i Mechaniki

**Kamil Bugała**

Nr albumu: 417522

**Karolina Laskowska**

Nr albumu: 417706

**Izabela Ożdżeńska**

Nr albumu: 417924

**Jan Paszkowski**

Nr albumu: 418366

# Serwer kompilacji wizualnych skryptów

Praca licencjacka  
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem  
**dr. Roberta Dąbrowskiego**  
Instytut Informatyki

Warszawa, Maj 2023



## **Streszczenie**

W pracy przedstawiamy implementację serwera kompilacji modów do gier. Wykorzystanie technologii Kubernetes w środowisku Google Cloud daje nam wiele możliwości, takich jak zapewnienie bezpieczeństwa całego systemu oraz jego odporność na duże obciążenie. Poruszamy tematy doboru technologii, implementacji poszczególnych etapów serwera oraz jego deployment i zapewnienie bezpieczeństwa. Stworzyliśmy serwer, który kompiluje mody do postaci wykonywalnej oraz jest odporny na potencjalnie niebezpieczne pliki wejściowe.

## **Słowa kluczowe**

skrypt wizualny, modyfikacja gry komputerowej, konteneryzacja, mikroserwisy, Docker, Kubernetes

## **Dziedzina pracy (kody wg programu Socrates-Erasmus)**

11.3 Informatyka

## **Klasyfikacja tematyczna**

D. Software  
D.2.1 Requirements/Specifications  
D.2.9 Management  
D.2.11 Software Architectures

## **Tytuł pracy w języku angielskim**

Visual script compilation server



# Spis treści

<b>1. Wprowadzenie</b>	5
<b>2. Wymagania</b>	7
<b>3. Architektura</b>	9
3.1. Ogólny zarys	9
3.2. Szczegółowy opis	10
3.3. Zalety architektury mikroservisów	11
<b>4. Dobór technologii i implementacja</b>	13
4.1. Konteneryzacja i Docker	13
4.2. Kubernetes	14
4.3. ASP .NET	16
4.4. PostgreSQL, EF Core	17
4.5. Pub/Sub, buckety	17
4.6. Dodatkowe narzędzia bezpieczeństwa	17
4.7. Odbyte kursy	18
<b>5. Bezpieczeństwo</b>	21
5.1. Konteneryzacja	21
5.2. Sanityzacja	22
5.2.1. Skrypt wizualny	22
5.2.2. Plik C#	22
5.2.3. Plik DLL	23
5.3. Limity czasowe	23
<b>6. Deployowanie w środowisku produkcyjnym</b>	25
6.1. Aktualizacja obiektów Kubernetesa	25
6.2. Problemy w trakcie aktualizacji obiektów Kubernetesa	26
6.3. Migracje baz danych na produkcji	27
<b>7. Napotkane trudności</b>	29
7.1. Wielowątkowość a DbContext	29
7.2. Sekwencyjność zmiany statusu pliku	30
7.3. Tagi obrazów w Google Cloud	31
7.4. Restrukturyzacja projektu - Git submodules, .NET solutions i projects	32
<b>8. Efekty końcowe</b>	33

<b>9. Organizacja pracy . . . . .</b>	<b>41</b>
<b>Bibliografia . . . . .</b>	<b>43</b>

# Rozdział 1

## Wprowadzenie

W ramach pracy licencjackiej zrobiliśmy serwer kompilujący modyfikacje gier. Jest to część większego narzędzia upraszczającego tworzenie takich modyfikacji.

Modyfikacje gier są ogółem opłacalne dla twórców i użytkowników gier, jednak modyfikowanie gier wymaga specyficznych umiejętności. Tworzenie modyfikacji może być uproszczone przez narzędzia do tego dedykowane, ale wytworzenie takiego narzędzia może być zajmujące dla twórców gier, zwłaszcza gdy ma w prosty sposób dawać możliwości edycji, ale nie zdradzać niepotrzebnie wewnętrznego kodu. Stąd pomysł zewnętrznego narzędzia do modyfikacji. Gracze mogą dzięki temu mieć narzędzie do modyfikowania gry, które jest uproszczone w obsłudze, a twórcy mają kontrolę nad zmianami w grze przy ograniczonej ilości pracy.

W programie do wytwarzania modów gracz modder edytuje plik, który opisuje, jak powinien działać mod. Ten plik jest zapisywany tak, aby nie obciążał komputera moddera, oraz aby przede wszystkim pamiętał sposób przedstawiania tego wizualnie dla moddera. Z powodu skupienia na przedstawianiu wizualnym, plik ten jest określany dalej skryptem wizualnym. Ten skrypt wizualny, aby stać się modem, musi być odpowiednio skompilowany do pliku, który może być wczytany do gry przez inne części całego narzędzia. Aby nie obciążać komputera moddera i umożliwić modyfikowanie na słabszych komputerach, w narzędziu używany jest nasz serwer, który kompiluje skrypt wizualny do pliku DLL.

Język skryptu wizualnego i przez to też jego kompilator otrzymujemy od firmy Ready-Code. Naszym zadaniem jest przede wszystkim zaimplementowanie architektury serwera. W to wchodzi dobranie i użycie odpowiednich technologii, ale także zapewnienie dostatecznego bezpieczeństwa przeciwko przewidywanym zagrożeniom. Wśród tych zagrożeń przewidujemy to, że do serwera może być wysłany dowolny plik, w tym specyficznie zaprojektowany, aby zagrozić działaniu serwera. W tym zakładamy też, że taki plik może wykorzystywać luki bezpieczeństwa w kompilatorze. Poza zagrożeniem działania samego serwera nasz serwer ma też dawać możliwość ograniczania możliwości wynikowego pliku. Dla wygody społeczności gry korzystającej z całego narzędzia, powstały mod ma mieć pewien poziom gwarancji, że nie przejmuje kontroli nad komputerem gracza, czy nie używa operacji zakazanych na przykład w bibliotekach systemowych.

Zaimplementowaliśmy więc serwer, który otrzymuje plik, w wypadku poprawnego wejścia zwraca wynik kompilacji, sprawdza pod względem bezpieczeństwa zawartość pliku i wyniku oraz jest przygotowany na pewne sposoby atakowania.





## Rozdział 2

# Wymagania

Przechodząc dokładniej do wymagań, do implementacji serwera otrzymujemy dwa kompilatory, ze skryptu wizualnego do pliku z kodem C# od firmy ReadyCode oraz kompilator .NET z pliku z kodem C# do pliku DLL. Nasz serwer dla odpowiedniego pliku ma zwrócić poprawny wynik potokowej kompilacji tymi dwoma kompilatorami. W wypadkach wykrycia jakiegś awarii ma dawać informacje o zatrzymaniu procesu kompilacji. Samo rozwiązanie ma być przygotowane do wystawienia i działania przy różnym natężeniu ruchu.

Główną część wymagań stanowi zapewnienie odpowiedniego poziomu bezpieczeństwa. Zakładamy, że można nam wysłać dowolny plik. Zakładamy też, że programy kompilatorów mogą posiadać luki bezpieczeństwa, w tym, że mogą skompilować plik, który:

- będzie zawierał kod niepożądany dla użytkownika, przykładowo, skompilowany moduł będzie uzyskiwał dostęp do sieci komputera i zbierał dane o użytkowniku,
- wykorzysta błędy etapów kompilacji, aby po którymś etapie dopisać do wyniku jakiś niepożądany kod, lub zapisze kod tak, aby na jakimś etapie nie był wykryty jako niebezpieczny,
- będzie znał etapy kompilacji na tyle dobrze, że będzie mógł wykonać niepożądane operacje na serwerze w trakcie kompilacji (arbitrary code execution).

Wymagamy więc sprawdzania zawartości pliku w poszukiwaniu zagrożeń, także po każdym etapie kompilacji na wypadek pojawienia się lub większej widoczności błędów. Chcemy też w miarę możliwości zapewnić izolację etapów kompilacji, aby w wypadku wykonywania kodu wysłanego przez atakującego:

- wykonywany kod nie miał dostępu do funkcji systemowych ponad to, co jest potrzebne dla przejętego etapu kompilacji
- atak na jednym etapie nie zdobywał dostępu do możliwości innych etapów
- pozostałe części serwera dalej działały w przewidziany sposób, pomijając komunikację z przejętą częścią

Zakładamy też, że poprawne skrypty powinny się kompilować w pewnym ograniczonym czasie. Przez to, chcemy przerywać kompilacje trwające zbyt długo, zakładając, że jest to spowodowane złośliwym plikiem zajmującym zasoby serwera, lub jest to oznaka przejęcia procesu kompilującego przez złośliwy skrypt.



## Rozdział 3

# Architektura

Przypomnijmy, że nasz serwer kompilacji dostaje od użytkownika na wejściu wizualny skrypt, a na wyjściu zwraca plik DLL. Wizualny skrypt nie jest bezpośrednio kompilowany do DLL, ale najpierw do C#, a dopiero potem z C# do DLL. Aby zrealizować tę funkcjonalność, stworzyliśmy architekturę bazowaną na stylu architektonicznym tzw. mikroserwisów. Architektura mikroserwisów polega na podziale aplikacji na niewielkie, niezależne jednostki, z jasno zdefiniowanymi zadaniami, które wchodzą ze sobą w interakcje za pomocą prostych bramek API. Jako że nasza architektura składa się z wielu jednostek, aby łatwiej było ją zrozumieć, najpierw przedstawimy jej ogólny zarys.

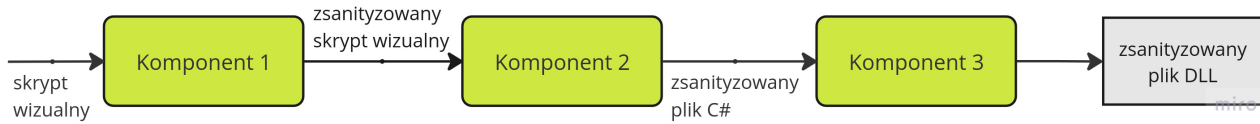
### 3.1. Ogólny zarys

Architektura stworzonej przez nas aplikacji składa się z trzech głównych komponentów. Wejściowy plik jest przekazywany przez kolejne komponenty i modyfikowany przez nie, aż uzyska postać gotowego pliku DLL.

Komponenty mają następujące funkcje:

- Komponent 1
  - Komunikuje się z użytkownikiem (przez protokół HTTP)
    - \* Odbiera wizualny skrypt od użytkownika
    - \* Przekazuje użytkownikowi aktualne informacje o etapie kompilacji pliku i ewentualnych błędach
    - \* Przekazuje użytkownikowi plik DLL, jeśli jest on gotowy
  - Sanityzuje wizualny skrypt
- Komponent 2
  - Odbiera wizualny skrypt od komponentu 1
  - Kompiluje wizualny skrypt do pliku C#
  - Sanityzuje skompilowany plik C#
- Komponent 3
  - Odbiera plik C# od komponentu 2

- Kompiluje plik C# do pliku DLL
- Sanityzuje skompilowany plik DLL

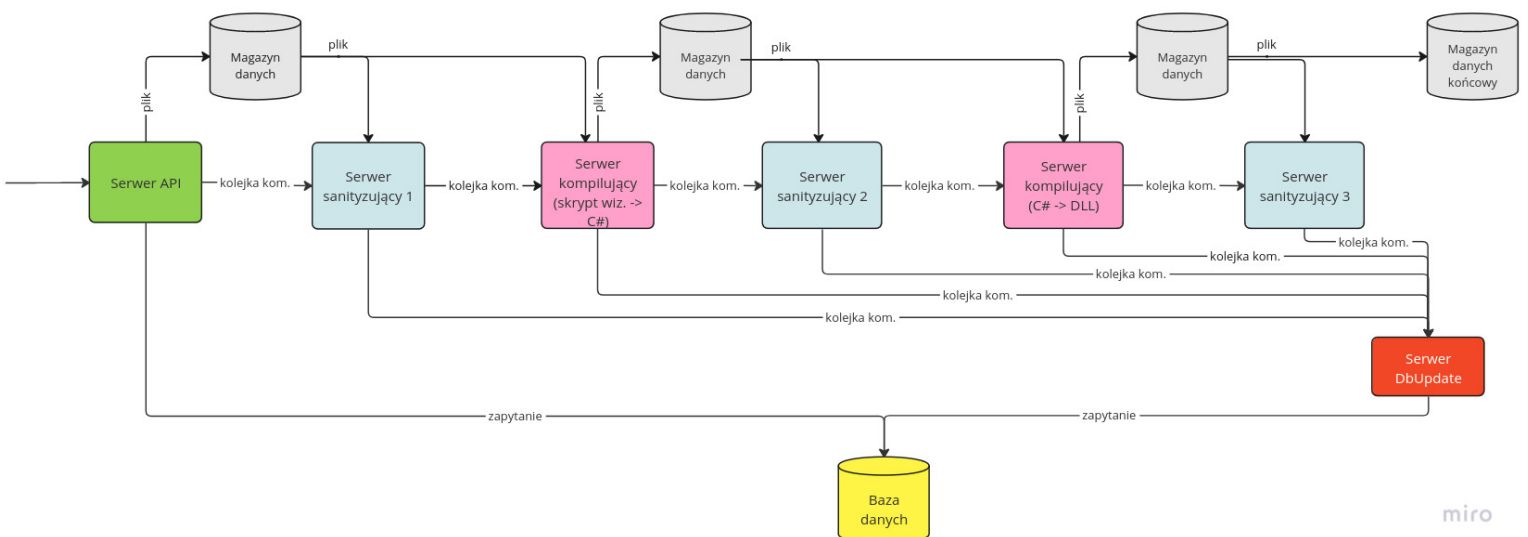


Rysunek 3.1: Ogólny zarys architektury

### 3.2. Szczegółowy opis

W uproszczonej wersji aplikacji, każdy komponent składa się z dwóch serwerów - serwera, który odbiera i kompiluje plik oraz serwera, który sanityzuje plik. Jedynie pierwszy serwer - serwer API - nie kompiluje pliku, a za to odpowiada za komunikację z użytkownikiem. Tak naprawdę aplikacja umożliwia także skalowanie rozwiązania, poprzez tworzenie wielu instancji każdego serwera.

Serwery komunikują się ze sobą za pomocą kolejki komunikatów i używają chmurowego serwisu magazynowania danych do przechowywania plików. Proces przekazania pliku z jednego komponentu do drugiego wygląda w ten sposób, że pierwszy serwer umieszcza plik w magazynie i wysyła na wspólną kolejkę komunikatów wiadomość, która zawiera nazwę tego pliku. Drugi serwer nasłuchuje na kolejce komunikatów, i kiedy dostanie wiadomość, odczytuje z niej nazwę pliku i pobiera odpowiadający plik z magazynu. Na każdy komponent przypada oddzielny magazyn plików. Istnieje też dodatkowy magazyn plików na produkty końcowe, czyli skompilowane pliki DLL. Do tego magazynu ostatni komponent przesyła plik po zakończeniu pracy.



Rysunek 3.2: Architektura

Aplikacja korzysta z bazy danych do przechowywania informacji o aktualnym stanie kompilowanego pliku i ewentualnych błędach. Przechowuje też informacje o czasie ostatniej zmiany

pliku, żeby wykrywać potencjalne timeouty. Pojedynczy serwer, po zakończeniu pracy nad obsługą pliku, przekazuje plik następnemu serwerowi oraz wysyła informację do bazy danych o zmianie stanu pliku. W przypadku wystąpienia błędu lub nieprzejęcia procesu sanityzacji, serwery wysyłają do bazy danych informację o niepowodzeniu. Trzeba też wspomnieć, że w celu zwiększenia wygody i bezpieczeństwa, tylko serwer API komunikuje się bezpośrednio z bazą danych. Pozostałe serwery przesyłają informacje do bazy danych pośrednio, poprzez dodatkowy serwer DbUpdate.

### 3.3. Zalety architektury mikroserwisów

Architektura mikroserwisów, którą wykorzystaliśmy, ma wiele zalet, takich jak:

- Wydajna praca nad aplikacją - dzięki temu, że jednostki architektury są od siebie niezależne, łatwo było podzielić pracę nad implementacją między nasz zespół, bo każdy mógł zająć się osobnym elementem. Zmniejszyło to liczbę konfliktów w kodzie, zwiększając wydajność zespołu. W przyszłości, podczas rozwijania aplikacji, ten styl architektury ułatwi firmie podział na zespoły zajmujące się poszczególnymi jednostkami.
- Łatwe skalowanie - architektura mikroserwisów ułatwia skalowanie aplikacji. Pozwala na łatwe zwiększenie liczby instancji danej jednostki oraz ułatwia rozszerzanie pojedynczych funkcjonalności, dzięki temu, że są one odizolowane.
- Odporność - w przypadku wystąpienia błędu, awarii ulega tylko jeden element. Zmniejsza się ryzyko awarii całej aplikacji.
- Bezpieczeństwo - w przypadku wykorzystania luki w bezpieczeństwie systemu, nawet jeśli zostanie przejęta kontrola nad jednym elementem, znacząco zmniejsza się ryzyko przejęcia kontroli nad pozostałymi.



## Rozdział 4

# Dobór technologii i implementacja

### 4.1. Konteneryzacja i Docker

*Konteneryzacja* to mechanizm pakowania aplikacji razem ze wszystkimi jej zależnościami w ustandaryzowany obiekt - *obraz kontenera*. Po stworzeniu obrazu kontenera, można zbudować z niego obiekt nazywany *kontenerem*, który jest izolowanym środowiskiem z działającą w środku aplikacją. W przeciwieństwie do maszyn wirtualnych, kontenery nie wymagają do działania utworzenia nowego systemu operacyjnego, ale korzystają z niskopoziomowych mechanizmów już istniejącego systemu operacyjnego. Dzięki temu zajmują znacznie mniej pamięci od maszyn wirtualnych, nadal zapewniając wysoki poziom izolacji i uniezależniając aplikację od infrastruktury konkretnego urządzenia.

*Docker* to narzędzie, które udostępnia te funkcjonalności (m.in. pakowanie aplikacji w obraz kontenera i budowanie kontenerów). Użyliśmy akurat Dockera, spośród narzędzi do konteneryzacji, bo jest najpopularniejszym tego typu narzędziem. Użyliśmy go pośrednio, korzystając z narzędzia Google Cloud Build, które używa Dockera.

#### Przygotowanie środowiska kontenera za pomocą Dockerfile

Aby Docker zbudował obraz kontenera z kodu aplikacji, należy w folderze z aplikacją umieścić tzw. *Dockerfile*, czyli plik zawierający instrukcje dla Dockera jak zbudować obraz. Opowiemy krótko o procesie konfiguracji środowiska kontenera, omawiając *Dockerfile* dla kontenera VS→C# (kompilującego wizualne skrypty do pliku C# - patrz: ilustracja architektury str. 10, rys. 3.2).

Kontener VS→C# korzysta z kompilatora dostarczonego przez firmę ReadyCode w prywatnym repozytorium. Oznacza to, że przygotowując środowisko kontenera, należy w *Dockerfile*'u zawrzeć:

- proces autoryzacji umożliwiający dostęp do prywatnego repozytorium,
- klonowanie repozytorium,
- instalację narzędzi potrzebnych do wykonania powyższych zadań (*git*, *git-lfs*, *openssh*),
- zbudowanie pliku wykonywalnego kompilatora.

Aby zautoryzować dostęp do prywatnego repozytorium dodaliśmy prywatny klucz SSH do systemu plików w środowisku kontenera za pomocą Dockerowej komendy **ADD**:

```
ADD id_ed25519 /root/.ssh/id_ed25519
```

Dodaliśmy następnie serwis kontroli wersji, w naszym przypadku Github, do znanych hostów. Komenda RUN umożliwia wykonywanie komend bashowych w środowisku kontenera:

```
RUN ssh-keyscan -H github.com >> /root/.ssh/known_hosts
```

Po sklonowaniu repozytorium, pobraliśmy podrepozytoria (ang. submodules) się w nim znajdujące (więcej o *git submodules* na str. 32):

```
RUN git clone --branch master git@github.com:readycodeio/readycode-compiler.git
WORKDIR /App/readycode-compiler
RUN git submodule update --init
```

Ostatecznie zbudowaliśmy plik wykonywalny kompilatora za pomocą komend `dotnet restore` i `dotnet build`:

```
RUN dotnet restore
RUN dotnet build /clp:ErrorsOnly "readycode-compiler/ReadyCodeCompiler"
    -c Release -o compiler
```

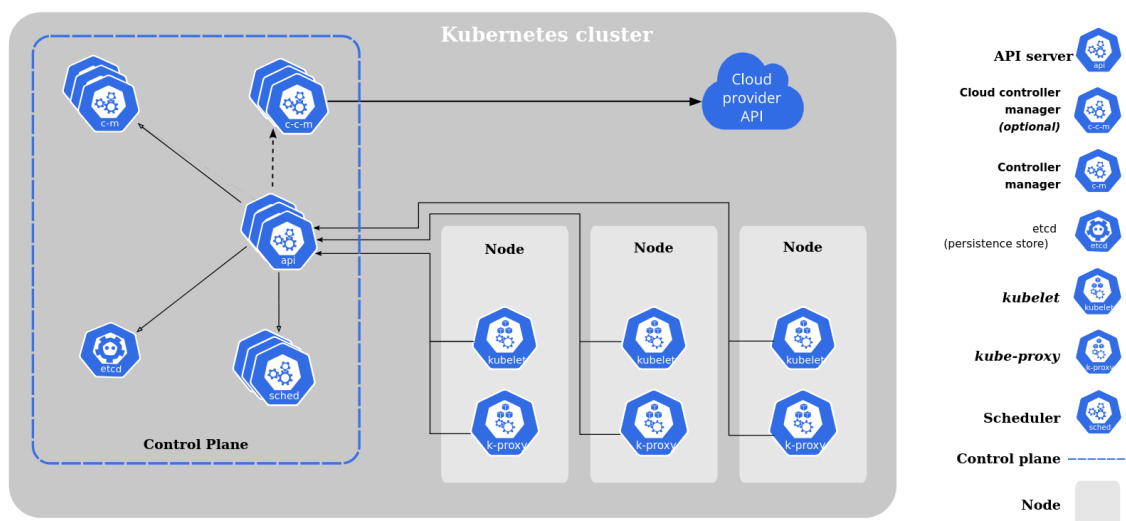
Dzięki temu procesowi mogliśmy uruchamiać plik wykonywalny kompilatora z poziomu kodu C# aplikacji:

```
ProcessStartInfo startInfo = new()
{
    FileName = "dotnet",
    Arguments = "/App/compiler/ReadyCodeCompiler.dll -i " +
                filename + " -o " + compiledLocalFilename,
    ...
};
var proc = Process.Start(startInfo);
```

## 4.2. Kubernetes

Otrzymaliśmy od firmy polecenie stworzenia systemu za pośrednictwem Kubernetes sterowanym w Google Cloud. Kubernetes jest popularną platformą do zarządzania kontenerami. Zapewnia on automatyzację, skalowanie i zarządzanie poszczególnymi aplikacjami opartymi na kontenerach działających w środowisku klastrów. Jest także rozwijany przez firmę Google, która oferuje interfejs do zarządzania tymi klastrami. Posiada on wiele zalet związanych z zachowaniem bezpieczeństwa, które są opisane w rozdziale “Bezpieczeństwo”.





Rysunek 4.1: Części składowe klastra Kubernetes

## Pody

W kontekście kubernetesa, są one najmniejszymi obiektami, reprezentującymi uruchomione mikroserwisy w klastrze. Pojedynczy pod zawiera jeden lub wiele powiązanych ze sobą kontenerów. Współdzielą one zasoby oraz dostęp sieciowy. W przypadku naszej aplikacji, każdy etap znajduje się w osobnym nodzie.

OVERVIEW

OBSERVABILITY

COST OPTIMIZATION

Filter

Is system object : False

Filter workloads

<input type="checkbox"/>	Name ↑	Status	Type	Pods	Namespace	Cluster
<input type="checkbox"/>	<a href="#">api-server</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>
<input type="checkbox"/>	<a href="#">csharp-file-sanitization-stage</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>
<input type="checkbox"/>	<a href="#">db-access</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>
<input type="checkbox"/>	<a href="#">dll-file-sanitization-stage</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>
<input type="checkbox"/>	<a href="#">to-csharp-worker</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>
<input type="checkbox"/>	<a href="#">to-dll-worker</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>
<input type="checkbox"/>	<a href="#">visual-script-sanitization-stage</a>	✓ OK	Deployment	1/1	default	<a href="#">api-server-cluster</a>

Rysunek 4.2: Działające node'y w klastrze

## Pliki YAML i konfiguracja

W celu konfiguracji i deklaracji żądanych stanów klastra, Kubernetes wykorzystuje pliki YAML. Pliki te zawierają opisy używanych przez system zasobów, takie jak pod, Deployment, ReplicaSet itp. Zasoby te definiują działanie aplikacji uruchomionej w klastrze. Wykorzystanie plików YAML w naszej aplikacji możemy podzielić na dwa rodzaje:

- ustawienie odpowiednich konfiguracji do zdeployowania systemu

- zaktualizowanie obrazów podów, w przypadku zmian w implementacji.

Poniżej opiszemy wykorzystaną konfigurację z wykorzystanych plików - `api-deployment.yaml`, użyty do konfiguracji deploymentu poda z serwerem API. Plik ten konfiguruje Deployment i Service w systemie Kubernetes.

Na początku określamy wersję API i rodzaj zasobu jako Deployment. Następnie ustalamy metadane dla tego zasobu, takie jak użyta nazwa i etykiety. Następnie definiujemy specyfikację, w której określamy liczbę replik dla poda, wybieramy pody do zarządzania na podstawie etykiet. Definiujemy template poda, który będzie tworzony przez Deployment. Następnie określamy woluminy oraz definiujemy kontenery. Określamy ustawienia pojedynczego kontenera, poprzez:

- określenie jego nazwy,
- podanie obrazu kontenera z rejestru Container Registry,
- ustalenie portu kontenera,
- dodanie zmiennych środowiskowych, takie jak ścieżka do klucza uwierzytelniającego.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: api-server
5    labels:
6      app: api-server
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: api-server
12    template:
13     metadata:
14       labels:
15         app: api-server
16     spec:
17       serviceAccountName: ksa-cloud-sql
18       volumes:
19         - name: google-cloud-key
20           secret:
21             secretName: pubsub-key
22       containers:
23         - name: api-server
24           image: gcr.io/${GOOGLE_CLOUD_PROJECT}/api-server:1.0.0
25           imagePullPolicy: Always
26           volumeMounts:
27             - name: google-cloud-key
28               mountPath: /var/secrets/google
29           ports:
30             - containerPort: 80
31           env:
32             - name: GOOGLE_APPLICATION_CREDENTIALS
```

Rysunek 4.3: Fragment zawartości pliku `api-deployment.yaml`

### 4.3. ASP .NET

Przy wyborze wielu technologii dobieraliśmy narzędzia dostępne w C#. Było to spowodowane znajomością tego języka w naszej grupie i w firmie, a także tym, że jakieś podstawy tego języka

i tak warto było poznać w ramach rozumienia procesu kompilacji.

Do pisania serwera API i jego komunikacji z siecią zdecydowaliśmy się skorzystać z ASP.NET. W ramach tego były zawarte narzędzia zarządzające komunikacją przez protokoły sieciowe dostępne z poziomu C#. Program serwera otrzymuje odpowiednie dane w inicjalizacji, takie jak dostęp do bazy danych, adres, na którym nasłuchuje i udostępniane ścieżki adresów, i na podstawie tego uruchamia odpowiednie metody z ich danymi wejściowymi, a jako odpowiedź wysyła odpowiednio opakowane dane zwracane przez podane metody. Dzięki temu przemianą danych z protokołów HTTP i TCP na odpowiednie zmienne w C# zajmował się framework. My natomiast podawaliśmy metody z odpowiednimi operacjami na tych danych i przesyłaniem ich dalej w naszym obiegu.

## 4.4. PostgreSQL, EF Core

Do postawienia bazy danych skorzystaliśmy z systemu PostgreSQL. Zdecydowaliśmy się na tę technologię z powodu jej znajomości, a także potencjalnego łatwego przeniesienia jej w przyszłości na dowolną platformę, na której byłby uruchamiany serwer. Z poziomu kontenerów, komunikację z bazą danych obsługujemy przez narzędzia Entity Framework Core (EF Core). Udostępniają one metody do komunikacji z bazą danych. Przede wszystkim zdecydowaliśmy się na to z powodu łatwości użycia w porównaniu z ręcznym połączeniem z bazą danych i wpisywaniem odpowiednich zapytań. Jest to też dodatkowe zabezpieczenie przed potencjalnym atakiem SQL injection.

## 4.5. Pub/Sub, buckety

W celu poprawnego działania naszej aplikacji, musieliśmy zapewnić komunikację pomiędzy jej instancjami. W tym celu wykorzystaliśmy usługę Google Cloud Storage oraz asynchroniczną usługę Google Cloud Pub/Sub. Google Cloud Storage pozwolił nam na przechowywanie plików w magazynach, tzw. bucketach. Pub/Sub umożliwił przesyłanie wiadomości pomiędzy niezależnymi instancjami naszej aplikacji. Komunikacja pomiędzy instancjami działa na zasadzie wysyłania wiadomości przez wydawców (publisher) i odbierania ich przez subskrybentów (subscriber). Do komunikacji publisher z subscriberem wykorzystujemy temat (topic) oraz subskrypcję (subscription). Temat jest miejscem, do którego wysyłane są wiadomości, a subskrypcja miejscem, z którego są pobierane.

### Działanie w kontenerach

Subscriber pojedynczej instancji odbiera nowe wiadomości, które otrzymuje na skutek wysłania ich przez publishera uruchomionego w innej instancji. Publisher wysyła wiadomość, kiedy w bucketcie zostanie zamieszczony nowy plik do obsługi. Instancja pobiera plik z bucketa, oraz przetwarza go według implementacji. Następnie umieszcza wynikowy plik do kolejnego bucketa oraz wysyła informację o gotowości wynikowego pliku przez publisher. Dzięki temu następna instancja może pobrać ten plik oraz obsłużyć go.

## 4.6. Dodatkowe narzędzia bezpieczeństwa

Istnieją dodatkowe technologie zapewniające większe bezpieczeństwo w obsłudze kontenerów. Na początku planowania architektury projektu, mieliśmy zamiar wybrać jedną z trzech poniżej opisanych projektów.

## **GVisor**

Jest to izolowany runtime do kontenerów, zapewniający dodatkową warstwę izolacji dla aplikacji działających w kontenerach. Bardzo dużą zaletą jest tworzenie tej izolacji na poziomie użytkownika, czyli zamiast tworzenia dodatkowej warstwy zasobów dla kontenera (analogicznie do maszyny wirtualnej), filtruje on komunikację kontenera z systemem. Wykorzystuje on tzw. *sandboxing*, który pozwala na uruchamianie kontenerów w bardziej izolowanej i bezpieczniejszej przestrzeni. Zapewnia dzięki temu ograniczenie aplikacji do zasobów systemowych.

## **Firecracker**

Firecracker jest technologią podobną do GVisor. Różni się jednak tworzeniem tak zwanych lekkich maszyn wirtualnych, czyli zapewnianiem części zasobów dla kontenerów w celu ograniczenia komunikacji z jądrem systemu.

## **Kata Containers**

Ostatnią możliwością jest projekt open-source, który integruje kontenery z wirtualizacją na poziomie jądra systemu, przez uruchomienie ich na swojej maszynie wirtualnej. Jak poprzednie dwie opisane technologie, zapewnia większą separację między kontenerami a systemem i zasobami serwera.

Wspomniane trzy technologie różnią się od siebie w wirtualizacji na różnych poziomach, a także w izolacji przy dostępie do pamięci, funkcji, oraz innych zasobów systemowych. Braliśmy pod uwagę, która z technologii będzie dla nas najbardziej wartościowa. Znaczenie miały czynniki, takie jak łatwość obsługi, czy popularność i wiarygodność oprogramowania. Ostatecznie z powodu ograniczeń czasowych i trudności z integracją nie wykorzystaliśmy żadnej technologii w praktyce. Wykonaliśmy natomiast research, który może przydać się firmie, jeżeli wystąpi potrzeba dodania dodatkowego bezpieczeństwa dla systemu w przyszłości.

## **4.7. Odbyte kursy**

Przed rozpoczęciem projektu licencjackiego, nie posiadaliśmy wiedzy potrzebnej do implementacji naszego systemu. W celu zdobycia praktycznej, firma zlecająca wykupiła dla naszego zespołu kursy na platformie Google.

### **Kubernetes w Google Cloud**

Kurs składał się z wielu modułów, pomagającym w nauce podstawowych i zaawansowanych technik organizacji kontenerów w Google Cloud. Mającymi największe znaczenie w zrozumieniu i obsłudze technologii były poniższe etapy.

- Wstęp do oprogramowania Docker
  - budowanie, uruchamianie i zarządzania kontenerami Docker
  - pobieranie obrazów Docker z platformy Docker Hub
  - przysyłanie obrazów Docker do Google Artifact Registry
- Wstęp do Kubernetes Engine
  - administracja klastrów za pomocą Google Kubernetes Engine

- Zarządzanie chmurą Google z użyciem Kubernetes
  - udostępnienie kompletnego klastra przy użyciu Kubernetes Engine
  - deployowanie kontenerów Docker i zarządzanie nimi za pomocą narzędzia wiersza poleceń `kubectl`
  - podzielenie aplikacji na mikroserwisy przy użyciu Deployments and Services
- Zarządzanie deployowaniem
  - tworzenie plików YAML, obsługiwanych przez Kubernetes w celu deploymentu poszczególnych fragmentów aplikacji
  - uruchamianie, aktualizacja i skalowanie deploymentów

Kurs *Kubernetes in Google Cloud* zapewnił nam wiedzę przydatną do obsługi używanej przez nas technologii. Po zakończeniu kursu, byliśmy bardziej obcy z interfejsem Google Cloud i jego funkcjami. Bardzo przydatna okazała się nauka obsługi Cloud Shell. Jest to środowisko online z dostępem z wiersza poleceń (ang. *command-line*) do zarządzania infrastrukturą i edytorem kodu online.

### **Zarządzanie bazą danych PostgreSQL w Cloud SQL**

Kolejnym istotnym kursem był "Manage PostgreSQL Databases on Cloud SQL". Umożliwił nam zdobycie umiejętności w obszarze zarządzania bazami danych PostgreSQL w środowisku Cloud SQL. Dzięki kursowi nauczyliśmy się, jak tworzyć bazy danych oraz jak automatyzować ich zarządzanie. W aplikacji wykorzystujemy nieskomplikowaną model bazy, dlatego kurs przydał się nam pod kątem nauki podstawowych funkcjonalności jakie oferuje Cloud SQL.



## Rozdział 5

# Bezpieczeństwo

Założeniem serwera jest możliwość jego wykorzystania przez dowolne osoby w sieci. Jak wspominaliśmy wcześniej, otrzymuje on od użytkowników pliki będące skryptami wizualnymi. W przypadku, kiedy użytkownik wysyła plik na serwer, może wystąpić wiele podatności, które mogą być potencjalnie wykorzystywane przez wysyłającego w celu uzyskania nieautoryzowanego dostępu lub do wykonania innych złośliwych działań. W tym celu, jednym z ważniejszych wymogów jest zapewnienie bezpieczeństwa przed takimi czynnościami. W naszym projekcie rozwiązaliśmy to poprzez:

- wykorzystanie konteneryzacji
- sanityzację poszczególnych etapów kompilacji
- nałożenie limitu czasowego na niektóre instancje.

### 5.1. Konteneryzacja

Jak wspominaliśmy wcześniej, serwer działa za pośrednictwem Kubernetes w Google Cloud. Takie podejście zapewnia wiele zalet ze względu na zwiększenie bezpieczeństwa systemu. W przypadku działania naszego systemu, najważniejszymi z nich są poniższe cztery:

- **Izolacja**  
System jest podzielony na poszczególne instancje, działające w oddzielnych kontenerach. Kontenery są odizolowane od siebie nawzajem i od hosta. Oznacza to, że aplikacje są chronione przed atakami z innych kontenerów. Załóżmy, że użytkownikowi udało się uzyskać dostęp do pierwszego etapu kompilacji, poprzez wysłanie złośliwego pliku. Poprzez izolację, atakujący nie jest w stanie zyskać kontroli nad innymi kontenerami i resztą systemu.
- **Skalowalność**  
Dzięki organizacji kontenerów przez Kubernetes, można w łatwy sposób dodawać i usuwać kontenery. Może istnieć wiele kopii działających tak samo aplikacji w kontenerach. W przypadku awarii kontenera, Kubernetes podmienia go na inną kopię kontenera. Dzięki temu, nie trzeba również uruchamiać ponownie całego serwera. Ponadto, w sytuacji, kiedy serwer będzie wykorzystywany przez większą liczbę użytkowników jednocześnie, istnieje możliwość równoległej pracy kilku kontenerów. Takie podejście chroni system przed przeciążeniem w potencjalnym wąskim gardle procesu kompilacji.

- **Automatyzacja**

Kubernetes pozwala na automatyzację aktualizacji instancji w kontenerach. Umożliwia szybką reakcję na nowe, potencjalne niebezpieczeństwa i błędy w oprogramowaniu. Możliwa jest także płynna aktualizacja instancji, czyli krokowe aktualizowanie kolejnych podów, dzięki czemu można uniknąć przerw w działaniu aplikacji.

- **Monitorowanie działania**

Dzięki czytelnemu i łatwemu w obsłudze interfejsowi w Google Cloud, można szybko sprawdzić działanie kontenerów i działających w nich aplikacji. Możemy dzięki temu szybko dostrzec powstałe błędy w działaniu aplikacji, a następnie podjąć próby rozwiązania zauważonych problemów.

## 5.2. Sanityzacja

System otrzymuje plik od użytkownika, następnie kompiluje go w pierwszym etapie kompilacji do pliku pośredniego. W drugim etapie kompilacji, kompiluje plik pośredni do pliku końcowego. W przypadku potencjalnego ataku, możemy wyróżnić trzy scenariusze:

1. Nadesłany plik w postaci skryptu wizualnego będzie niebezpieczny.
2. Nadesłany plik skompiluje się do pośredniego pliku, który będzie niebezpieczny.
3. Nadesłany plik przejdzie przez dwa etapy kompilacji, a następnie skompiluje się do niebezpiecznego pliku wykonywalnego.

W celu ochrony przed trzema powyższymi możliwościami, wykonujemy sanityzację plików w trzech etapach.

1. Sanityzacja skryptu wizualnego.
2. Sanityzacja pliku w języku C#.
3. Sanityzacja pliku o rozszerzeniu DLL.

### 5.2.1. Skrypt wizualny

Skrypt wizualny, podawany przez użytkownika, jest napisany w języku YAML. Jest on językiem przeznaczonym do reprezentowania danych w formacie czytelnym zarówno dla człowieka, jak i komputera. Aby nie przekierowywać nieodpowiedniego pliku do dalszego etapu kompilacji, sprawdzamy, czy plik jest w poprawnym formacie dla tego języka. Wykorzystaliśmy do tego publiczne oprogramowanie (ang. *opensource*) Yamlint. Narzędzie Yamlint sprawdza, czy plik jest zgodny z określonymi standardami i specyfikacjami YAML. W serwerze sanitującym wizualne skrypty uruchamiany jest nowy proces, który uruchamia narzędzie Yamlint dla rozpatrywanego pliku, a następnie sprawdza, czy na standardowym strumieniu błędów nie zostało nic wypisane. Wówczas uznaje plik jako poprawny.

### 5.2.2. Plik C#

Kolejnym etapem jest sprawdzenie pliku pośredniego, napisanego w języku C#. Na początku chcemy zapewnić, że plik posiada znaki wyłącznie w formacie ASCII. Sprawdzamy to poprzez stworzenie kopii pliku i enkodowanie do postaci UTF-8. Jeżeli oryginalny plik i jego kopia



różnią się od siebie, oznacza to, że w kodzie C# występują niedozwolone znaki, wtedy uznajemy plik za potencjalnie niebezpieczny. Następnie dokonujemy tokenizacji kodu. Dzielimy go na mniejsze elementy, czyli tokeny, reprezentujące odpowiednie fragmenty kodu. Dzięki temu kod w pliku jest pogrupowany na oddzielne grupy elementów, takie jak nazwy zmiennych, operatory, znaki interpunkcyjne itp. Tokenizacja do efektywnego przetworzenia kodu wykorzystuje wyrażenia regularne (ang. *regular expression*, w skrócie *regex*). Kolejną częścią jest sprawdzenie, czy przyłączone do kodu biblioteki są zaufane. Zezwolenie na korzystanie ze wszystkich bibliotek mogłoby stworzyć potencjalne niebezpieczeństwa, takie jak możliwość tworzenia innych procesów lub nawiązywanie połączenia sieciowego. Utworzyliśmy białą listę (ang. *whitelist*) bibliotek, którą możemy dopuścić w pliku. Dzięki tokenizacji możemy w łatwy sposób sprawdzić, czy każda biblioteka należy do listy. Ostatnią walidacją jest sprawdzenie długości nazw w kodzie. Wszystkie nazwy zmiennych jak i funkcji muszą posiadać co najwyżej określoną stałą długość. Wykonujemy to w celu likwidacji podatności związanych z przepełnieniem bufora przechowującego nazwę w pamięci.

### 5.2.3. Plik DLL

Ostatnim etapem jest sanityzacja końcowego pliku, będącego w postaci DLL (z ang. *Dynamic Link Library*). Do walidacji wykorzystaliśmy Mono.Cecil - bibliotekę open source dla platformy .NET. Umożliwia ona analizę informacji zawartych w plikach DLL. Szczególne przydatne okazały się zestawy referencyjne (Reference Assemblies). Są to zestawy zawierające minimalną ilość metadanych wymaganych do reprezentowania interfejsu API biblioteki.

Na podstawie wymogów firmy, utworzyliśmy whitelistę tych zestawów. Sanityzacja opiera się na sprawdzeniu, czy nie zostały wykorzystane potencjalnie niebezpieczne moduły Reference Assemblies. Z każdego pliku na początku pobieramy metadane, czyli Reference Assemblies. Następnie sprawdzamy, czy każdy moduł z metadanych należy do whitelisty. Jeżeli jakkolwiek moduł nie będzie należał do whitelisty, plik zostaje uznany jako potencjalnie niebezpieczny.

## 5.3. Limity czasowe

Może wystąpić sytuacja, że użytkownik prześle plik powodujący zapętlenie wewnętrznych kompilatorów na serwerze. Aby temu zapobiec, nałożyliśmy limit czasowy (ang. *timeout*) na kompilację plików na serwerze. W przypadku przekroczenia czasu, proces kompilacji zostaje zatrzymany i zabity. Serwer zwróci wtedy użytkownikowi informację, że nadesłany przez niego plik spowodował przekroczenie limitu dozwolonego czasu.



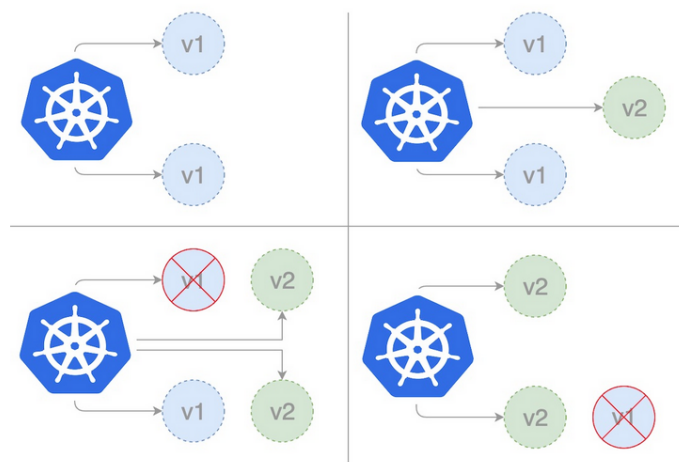
## Rozdział 6

# Deployowanie w środowisku produkcyjnym

Kolejnym ważnym problemem, który postanowiliśmy się zająć było zbadanie tematu wprowadzania zmian w aplikacjach oraz bazy danych w środowisku produkcyjnym. Chcielibyśmy, aby wprowadzanie zmian powodowało możliwie małą przerwę czasową dla użytkowników produktu, a także aby było możliwie bezpieczne.

### 6.1. Aktualizacja obiektów Kubernetesa

Najpierw rozpatrzyliśmy sytuację, w której aktualizujemy aplikację, ale bez żadnych zmian w schemacie bazy danych. Bardzo użytecznym do tego zadania okazał się być Kubernetes, ponieważ aplikując zmiany z jego użyciem, mamy zapewnioną płynną aktualizację. Kubernetes wykorzystuje mechanizm “rolling updates”, który umożliwia aktualizację unikając przerwy w działaniu aplikacji dzięki krokowemu aktualizowaniu kolejnych podów. Aplikacja ma wyznaczoną liczbę swoich replik, większą od jeden, a także maksymalną liczbę podów, które mogą być niedostępne w trakcie aktualizacji wyrażoną w wartości bezwzględnej lub procentowo. Po wykonaniu odpowiedniej instrukcji narzędzia “kubectl” powstaje pod z nowszą wersją aplikacji, a kiedy jest już on w stanie gotowości, usuwany jest pod ze starą wersją. Taki proces odbywa się dla wszystkich podów w starej wersji. Kubernetes dba o to, aby przekierowywać ruch tylko do tych podów, które są w stanie gotowości, dzięki czemu użytkownik nie zostanie dotknięty niepożądanym błędem.



Rysunek 6.1: Rolling updates

Konfiguracje obiektów Kubernetesa naszej struktury mieliśmy zapisane w plikach YAML. Jeżeli aktualizacja obiektu polegała na zmianie konfiguracji, np. na dodaniu nowej zmiennej środowiskowej, aplikowaliśmy odpowiednie zmiany w pliku konfiguracyjnym, a następnie komendą:

```
kubectl apply -f <UPDATED_FILE_NAME>
```

aktualizowaliśmy dany obiekt. Natomiast często aktualizacja obiektu dotyczyła jedynie zmiany używanego obrazu, wówczas wystarczyło uruchomić komendę:

```
kubectl set image
  deployment/<DEPLOYMENT_NAME> <CONTAINER_NAME>=<IMAGE_NAME>:<TAG>
```

## 6.2. Problemy w trakcie aktualizacji obiektów Kubernetesa

Podczas aktualizacji obiektu Kubernetesa może okazać się, że nowsza wersja obrazu bądź nowsza konfiguracja powoduje, że pod nie osiągnął stanu gotowości. Jego aktualny status wskazuje na błąd. Wówczas chcielibyśmy zaniechać aktualizacji do nowszej wersji i pozostawić uprzednio działający wariant. W tym przypadku również możemy liczyć na narzędzia Kubernetesa i wystarczy, że wywołamy komendę:

```
kubectl rollout undo <OBJECT_KIND>/<OBJECT_NAME>
```

Ważnym dla programisty w rozwiązywaniu problemów oraz zarządzaniu aktualizacją w środowisku produkcyjnym może być także zatrzymanie oraz wznowienie aktualizacji “rolling updates”. Te działania możemy wykonać odpowiednio komendami:

```
kubectl rollout pause <OBJECT_KIND>/<OBJECT_NAME>
kubectl rollout resume <OBJECT_KIND>/<OBJECT_NAME>
```

Natomiast aktualny status możemy sprawdzić uruchamiając:

```
kubectl rollout status <OBJECT_KIND>/<OBJECT_NAME>
```

### 6.3. Migracje baz danych na produkcji

Czasami jednak aktualizacje aplikacji wymagają także migracji baz danych, co przynosi więcej problemów. Musieliśmy poszukać bardziej złożonych rozwiązań, które pomogłyby zminimalizować przerwy w działaniu aplikacji bądź błędy.

Pierwszym krokiem było oddzielenie dokonywania zmian w schemacie bazy danych od działania naszych serwerów. Do łączenia się z bazą danych używaliśmy wcześniej wspomnianej platformy “Entity Framework Core”. Jej funkcjonalności pozwalały na łatwe utworzenie tabeli SQL z kodu w C# na podstawie zdefiniowanego modelu - klasy w C#. W dodatku narzędzie EF Core Migrations umożliwia ewolucję bazy danych w miarę zmieniania się modelu. Dzięki temu, po zmianie kodu aplikacji naruszającej aktualny schemat bazy danych, wystarczy wywołać komendę

```
dotnet ef migrations add <MIGRATION_NAME> --project <CSPROJ_FILENAME>
```

Wówczas powstaną pliki zapisujące zmiany w schemacie bazy danych względem poprzedniej. Było to dla nas bardzo wygodnym narzędziem oraz pozwalało zachować czystość kodu. Jednak początkowo, niepotrzebnie często używaliśmy EF Core, ponieważ powstające zmiany w schemacie bazy danych od razu aplikowaliśmy do istniejącej już bazy danych wraz z uruchomieniem aplikacji. Np. w kodzie naszych serwerów zawarte były instrukcje tworzące tabele, które się uruchamiały wraz z tworzeniem “DbContext” - reprezentanta sesji z bazą danych. Możliwe jest także użycie instrukcji “DbContext.Database.Migrate()” podczas startu aplikacji, która automatycznie zaaplikuje zmiany schematu w bazie danych. W środowisku produkcyjnym nie jest to właściwe, ponieważ wywołanie kodu zmieniającego schemat bazy danych z poziomu aplikacji, może powodować błędy, a nawet utratę danych. Jest to spowodowane tym, że gdy uruchomimy parę instancji tej samej aplikacji na raz, to w tym samym momencie mogą chcieć się jednocześnie wykonać skrypty SQL dokonujące zmian w schemacie, co jest niewspierane przez narzędzia do migracji oraz może powodować uszkodzenie danych. Innym problemem tego rozwiązania jest to, że jeśli chcemy z aplikacji wykonywać kod, który zmienia schemat, to użytkownik bazy danych, który ma dostęp z tej aplikacji, musi mieć uprawnienia do wykonywania instrukcji SQL typu “DROP”. Naraża to bezpieczeństwo danych, bo wówczas przy jakimś ataku na aplikację, ktoś może usunąć wszystkie dane. Lepszą praktyką jest, aby mieć osobnego użytkownika bazy danych do wykonywania zmian w schemacie.

Aby uniknąć wyżej opisanego problemu, postanowiliśmy wydzielić dokonywanie zmian w schemacie bazy danych do oddzielnego kontenera. Pomysłem było, aby tuż przed aktualizacją aplikacji uruchamiać obiekt Kubernetesa Job, którego zadaniem byłoby wykonanie odpowiedniego skryptu SQL. Nie chcieliśmy jednak pisać ręcznie skryptów SQL, co byłoby pracochłonne i podatne na błędy, ale znów posłużyć się Entity Framework Core. EF Core zachowuje informacje o migracjach w odpowiednich plikach oraz nadaje migracjom identyfikatory. W dodatku to narzędzie udostępnia funkcjonalność generowania skryptów SQL. Skrypty te zawierają w sobie również automatycznie wygenerowane instrukcje, które aktualizują tabele zawierające informacje o dokonanych migracjach. Dzięki temu łatwiej kontrolować przeprowadzanie migracji na produkcji, ponieważ można odczytać ich historię. Zatem planem postępowania w przypadku zmian w schemacie bazy danych byłoby:

1. dokonać zmiany w implementacji aplikacji (w kodzie C#);
2. wygenerować automatycznie plik zawierający informacje o zmianach w schemacie przy pomocy EF Core Migrations;
3. wygenerować automatycznie skrypt SQL na podstawie ww. pliku przy pomocy EF Core Migrations;
4. wywołać skrypt SQL na produkcyjnej bazie danych;

## 5. zaktualizować aplikację.

Aby to zrealizować, utworzyliśmy plik konfiguracyjny dla obiektu Kubernetesa Job, który miałby uruchamiać wyżej opisany skrypt SQL. Tutaj również napotkaliśmy trudności.

W naszym rozwiązaniu do stworzenia bazy danych użyliśmy narzędzia Google - Cloud SQL. Do połączenia się z bazą danych używaliśmy proxy, którego obraz także mieliśmy dostarczony przez Cloud SQL. Dzięki temu również dbaliśmy o bezpieczeństwo, ponieważ serwer proxy Cloud SQL umożliwia bezpieczne połączenia poprzez szyfrowanie ruchu do instancji bazy danych, a także nakłada ograniczenia na adresy IP, z których można się połączyć. Dla serwerów, które były uruchomione trwale, mogliśmy w tym samym podzie uruchomić także proxy. Jednak dla obiektu Job było to problematyczne. Job ma się zakończyć po wykonaniu swojego zadania, czyli w naszym przypadku po wykonaniu skryptu SQL. Jednak jeśli w tym samym Jobie dodamy kontener, który będzie stanowił proxy do łączenia z bazą danych, to Job się nigdy nie wykona, ponieważ proxy jest serwerem, który się "nie kończy". W związku z tym wybraliśmy rozwiązanie, które nie wymaga łączenia się do bazy danych poprzez proxy. Jednak aby to było możliwe, trzeba było spełnić wymagania Cloud SQL do łączenia się z bazą danych inaczej niż poprzez proxy. Job musiał się łączyć z bazą danych po prywatnym IP, przy czym Job oraz instancja Cloud SQL musiały należeć do tej samej VPC (Virtual Private Cloud). VPC jest to logiczna i odizolowana przestrzeń sieciowa. Musieliśmy zatem zmienić rodzaj naszego klastra na klaster VPC-native i tam tworzyć potrzebne węzły. Wówczas możliwe było połączenie poda z instancją bazy danych po prywatnym IP. Tym sposobem udało nam się utworzyć narzędzie, które wykonywało skrypt SQL na produkcji oraz plan przeprowadzania migracji.

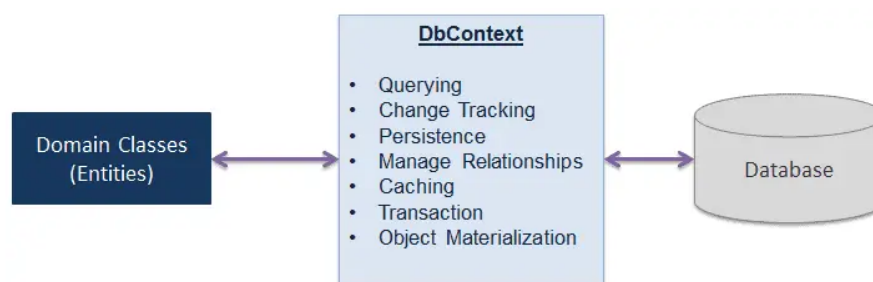
Niestety, nasze rozwiązanie nadal nie jest idealne, ponieważ wymaga chwilowego zatrzymania serwisu na czas przeprowadzania aktualizacji z migracjami. Musimy uruchomić Job wykonujący skrypt SQL, sprawdzić czy się prawidłowo wykonał, a następnie zaktualizować aplikację. W tej chwili nie jest to w pełni zautomatyzowane. Jednym z potencjalnych rozwiązań byłoby użycie Init Containers od Kubernetesa. Działają one w taki sposób, że gdy tworzymy pod, najpierw uruchamiają się Init Containers, a dopiero gdy one się wykonają, uruchamiane są właściwe kontenery z aplikacją. U nas Init Containerem byłby wyżej opisany Job. Jednak w tym rozwiązaniu Init Container utworzyłby się dla każdego poda, a zakładamy że będziemy chcieli mieć wiele podów dla danego serwera. To powodowałoby jednoczesne uruchomienie wielu skryptów SQL na raz, co mogłoby znów doprowadzić do błędów. Rozwiązaniem, które rozwiązywałoby ten problem, byłoby skorzystanie z narzędzia Helm, które jest menedżerem pakietów dla Kubernetesa. Helm umożliwia uruchomienie Init Container tylko dla jednego poda, co zapobiega konfliktom. Jednak implementację takiego rozwiązania pozostawiliśmy do czasu dalszego rozwijania produktu.

## Rozdział 7

# Napotkane trudności

### 7.1. Wielowątkowość a DbContext

Wspomnieliśmy wcześniej, że do połączeń z bazą danych z aplikacji .NET używaliśmy platformy “Entity Framework Core”. Przy korzystaniu z EF Core kluczową klasą jest DbContext. Klasa ta reprezentuje sesję z bazą danych – używana jest do zapytań oraz poleceń do bazy danych.



Rysunek 7.1: Zastosowanie DbContext

Aby wygodnie móc korzystać z DbContext w innych klasach naszych aplikacji .NET, chcieliśmy, żeby dostęp do instancji DbContext był umożliwiony dzięki dependency injection, które jest wspierane przez .NET. W pliku “Program.cs” (podprogram główny dla aplikacji ASP.NET) konfigurowaliśmy serwisy dla budowniczego aplikacji webowej (WebApplicationBuilder). Właśnie tam dodaliśmy DbContext jako serwis, aby inne klasy mogły wygodnie używać tej klasy. Początkowo użyliśmy metody “AddDbContext<TContext>()” ze wskazaniem w opcjach na użycie Npgsql (dostęp .NET do PostgreSQL) oraz podaniem parametrów połączenia do naszej bazy danych. Pozostałe opcje pozostawiliśmy domyślne. Jednak wraz z rozwijaniem i rozszerzaniem naszych aplikacji napotkaliśmy się na poniższy komunikat.

"A second operation started on this context before a previous operation completed. This is usually caused by different threads using the same instance of DbContext, however, instance members are not guaranteed to be thread-safe."

Okazało się, że ta sama instancja DbContext była dzielona między różnymi wątkami tej samej klasy. Jednak DbContext nie jest bezpieczne wątkowo (thread-safe) i stąd wynikał napotkany błąd. Aby rozwiązać sytuację, przyjrzelśmy się bliżej metodzie “AddDbContext<TContext>()”.

Okazuje się, że metoda ta domyślnie dodaje DbContext do serwisów jako “scoped object”, co oznacza, że instancja DbContext jest ta sama w ramach jednego żądania, ale inna w ramach różnych żądań. Nasz problem dotyczył dzielenia DbContext między różnymi wątkami tej samej klasy, w związku z czym potrzebowaliśmy użyć innego rodzaju żywotności obiektu. Poszukiwanym rodzajem obiektu z pasującą nam żywotnością był “transient object”, co oznacza, że instancja takiego obiektu jest tworzona za każdym razem inna. Rozwiązaniem problemu było przekazanie w opcjach metody “AddDbContext<TContext>()” informacji o pożądanym trybie żywotności obiektu tj. “transient”. Wówczas aplikacje mogły prawidłowo korzystać z bazy danych.

## 7.2. Sekwencyjność zmiany statusu pliku

W bazie danych chcieliśmy przechowywać aktualny stan, w jakim znajduje się dany plik, czyli na jakim etapie kompilacji się znajduje, czy został już zwalidowany i czy przeszedł dany etap sanityzacji. W zaplanowanej przez nas architekturze do bazy danych miał dostęp jedynie ApiService oraz DbUpdate. Dane o pliku w początkowym stanie i na pierwszym etapie wpisywał ApiService bezpośrednio do bazy danych. Jednak na kolejnych etapach, plik był modyfikowany przez serwisy niemające dostępu do bazy danych. W związku z czym, musiały one jakoś przekazywać informację do DbUpdate, który następnie miałby zaktualizować dany rejestr. Do komunikacji między nimi użyliśmy wspomnianej wcześniej kolejki komunikatów Pub/Sub. Cały proces aktualizacji statusu pliku przebiegał tak, że dany serwis, który wykonał pracę na danym pliku, wstawiał o tym informację do kolejki komunikatów. Komunikat ten zawierał w sobie identyfikator skryptu oraz nowy status skryptu. Wszystkie serwisy zmieniające stan plików wstawiały wiadomości do tej samej kolejki komunikatów. Natomiast subskrybentem tej kolejki był serwis DbUpdate. Pobierał on komunikat z kolejki, a następnie w bazie danych aktualizował rekord dotyczący danego skryptu, czyli zmieniał jego status.

Początkowo serwis DbUpdate naiwnie zmieniał status pliku, czyli zawsze kiedy odebrał wiadomość z kolejki komunikatów, zmieniał status pliku w bazie danych, nie biorąc pod uwagę z jakiego stanu na jaki zachodzi zmiana. W większości testów takie rozwiązanie działało, jednak czasami obserwowaliśmy, że status pliku wskazuje na inny stan niż stan, w którym rzeczywiście znajdował się plik. Po zbadaniu tematu okazało się, że kolejka Pub/Sub domyślnie nie zapewnia dostarczania komunikatów w kolejności ich wstawiania do kolejki. Jeżeli byśmy chcieli, aby tak się działo, to moglibyśmy przy tworzeniu subskrypcji Pub/Sub dodać odpowiednią flagę.

```
gcloud pubsub subscriptions create SUBSCRIPTION_ID --enable-  
message-ordering
```

Jednak my postanowiliśmy zastosować inne rozwiązanie, które akurat do naszej specyfikacji pasuje. Otóż wiemy, że w naszym przypadku statusy można uporządkować rosnąco tak, by było spełnione, że jeśli plik jest w dalszym statusie, to już nie może się cofnąć do wcześniejszego statusu.



```
Models > ScriptForm.cs
1 namespace Models;
2
3 public enum ScriptForm
4 {
5     Failed,
6     VisualScript,
7     VisualScriptSanitized,
8     CSharpFile,
9     CSharpFileSanitized,
10    DllFile,
11    DllFileSanitized
12 }
```

Jedynym wyjątkiem był status “Failed”, który mógł osiągnąć plik w każdym stanie. Gdy jakiś plik otrzymał status “Failed”, to nie chcieliśmy mu już zmieniać statusu na żaden inny.

W związku z tym serwis DbUpdate po otrzymaniu komunikatu o zmianie stanu pliku, najpierw sprawdzał, czy dany status jest w kolejności po aktualnym stanie i jeśli tak, to zmieniał status. Natomiast jeśli komunikat informował, by zmienić stan na “Failed”, to zawsze był on zmieniany, a gdy aktualny status był statusem “Failed”, to nigdy nie był on zmieniany.

Zdecydowaliśmy się na to rozwiązanie, ponieważ nasze serwisy wykonujące pracę nad plikiem, czyli wszystkie stacje kompilacji i sanityzacji, najpierw wykonywały pracę nad plikiem, następnie informowały kolejny serwis, że może on już przejąć pracę nad tym plikiem, a dopiero na koniec wstawiały wiadomość o zmianie statusu do kolejki komunikatów. Przyjęliśmy taką kolejność operacji, aby kompilacja przebiegała możliwie szybko. Jednak to skutkowało tym, że wiadomości niekoniecznie musiały być wstawiane do kolejki komunikatów w kolejności, która by nam odpowiadała. Mogłoby się zdarzyć np. tak, że po powiadomieniu kolejnego serwisu o możliwości przejęcia pracy nad plikiem ten serwis wykona pracę oraz wstawi swoją wiadomość do kolejki komunikatów, a dopiero wtedy pierwszy serwis wstawi swoją wiadomość do kolejki komunikatów. Wówczas rozwiązanie jedynie z użyciem flagi “--enable-message-ordering” nie zadziałałoby.

## 7.3. Tagi obrazów w Google Cloud

Jedną z napotkanych trudności była zmiana tagów w obrazach kontenerów w Google Cloud. W celu naniesienia zmian na produkcji, musieliśmy między innymi zaktualizować obrazy dla podów, w których wystąpiły modyfikacje. Warto wspomnieć, że dla każdego node’a jest ustalany unikalny obraz. W tym celu budowaliśmy nowy obraz komendą:

```
gcloud builds submit --config cloudbuild-X.yaml
```

gdzie X jest nazwą node’a, którego obraz chcemy zmodyfikować. Przed wykonaniem komendy modyfikowaliśmy plik cloudbuild-X.yaml, zmieniając tag obrazu występujący w jego treści. W celu zmiany obrazu, z którego miał korzystać Deployment, wykorzystywaliśmy komendę:

```
kubectl set image deployment/api-server api-server=gcr.io/$
{GOOGLE_CLOUD_PROJECT}/api-server:x.x.x,
```

gdzie x.x.x jest podanym tagiem. W powyższej komendzie podawaliśmy tag dla aktualnie działającej wersji obrazu. W przypadku, kiedy w konfiguracji Deploymentu był akurat obraz o tym samym tagu, system Kubernetes uznawał, że w konfiguracji nie wystąpiła żadna zmiana. Kubernetes nie pobierał przez to nowego obrazu z rejestru obrazów. Przez to nie widzieliśmy pożądaných zmian.

Na początku nie byliśmy świadomi tego problemu. Nie rozumieliśmy, dlaczego w niektórych przypadkach naniesione zmiany nie zmieniają w żaden sposób dotychczasowego działania aplikacji. Ostatecznie doszliśmy do rozwiązania błędu. Od tego momentu, każdej nowej modyfikacji nadawaliśmy nowy tag, a w sytuacji, kiedy wystąpiła duża ilość kopii, usuwaliśmy ich niepotrzebną część.

## 7.4. Restrukturyzacja projektu - Git submodules, .NET solutions i projects

### Zduplikowany kod

W miarę jak aplikacja się rozwijała, tworzyliśmy kolejne elementy architektury, a kod każdego elementu umieszczany był w oddzielnym repozytorium. Kiedy dodaliśmy do aplikacji bazę danych, aby umożliwić pozostałym elementom komunikację z nią, każdy z elementów musiał dysponować definicją modelu bazy danych. Naszym początkowym rozwiązaniem było dodanie definicji modelu do każdego repozytorium. Oznaczało to, że w każdym repozytorium trzymaliśmy ten sam kod definiujący model.

Było to rozwiązanie krótkoterminowe i bardzo uciążliwe, bo przy każdej zmianie modelu trzeba było zmienić kod w kilku miejscach, przez co marnowaliśmy czas. Poza tym łatwo było o błędy aplikacji w przypadku, kiedy zapomnieliśmy zmienić kod wszędzie lub popełniliśmy pomyłkę przy kopiowaniu kodu.

### .NET solutions i projects

W początkowym rozwiązaniu ze zduplikowanym kodem, każdy element architektury tworzył osobne *.NET solution*. W .NET Framework, *.NET solution* jest kontenerem na jeden lub więcej *projekt .NET* i zawiera instrukcje kompilacji projektów .NET. *Projekt .NET* za to jest instancją zawierającą wszystkie pliki, które są kompilowane do pliku wykonywalnego, biblioteki lub strony internetowej.

Trzymanie kodu każdego elementu architektury jako osobne .NET solution uniemożliwiało współdzielenie kodu między projektami .NET. Dlatego, aby umożliwić współdzielenie kodu, zmieniliśmy strukturę kodu aplikacji tak, aby kod każdego elementu nie był .NET solution, a jedynie projektem .NET, oraz stworzyliśmy jedno główne .NET solution, które zawierało wszystkie te projekty. Następnie odseparowaliśmy kod definicji modelu bazy danych do nowego projektu .NET i dodaliśmy go do list zależności w pozostałych projektach, aby mogły się do niego odwoływać.

### Git submodules

Zmieniając strukturę projektu, zdecydowaliśmy się stworzyć nowe, główne repozytorium, w którym, jako podrepozytoria, znajdują się wcześniejsze repozytoria. Było to możliwe dzięki usłudze Git submodules, która pozwala na wygodne trzymanie repozytoriów wewnątrz innych repozytoriów i oddzielenie procesu zarządzania repozytorium zewnętrznym od zarządzania repozytoriami wewnętrznymi. Z poziomu wewnętrznego repozytorium, zarządzanie niczym nie różni się od sytuacji, w której jest ono jedynym repozytorium. Zaś z poziomu zewnętrznego repozytorium, każde podrepozytorium jest trzymane jako link do commitu.

Dzięki użyciu usługi Git submodules, zachowaliśmy historię wszystkich repozytoriów oraz mogliśmy kontynuować poprzedni flow pracy, skupiając się na jednym repozytorium naraz, bez wnikania w to jak są ze sobą połączone.

## Rozdział 8

# Efekty końcowe

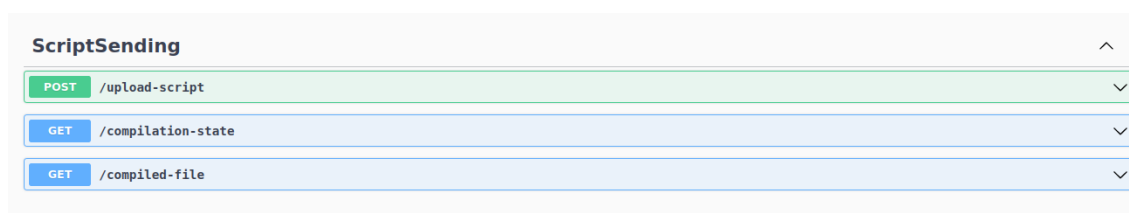
Efektem naszej pracy był gotowy serwer do kompilacji wizualnych skryptów do plików DLL. Powstał on jako element narzędzia do modyfikowania gier, nad którym pracuje firma ReadyCode i które ma na celu udostępnienie modyfikowania gier nawet mniej doświadczonym deweloperom.

Zaprojektowaliśmy architekturę mikroserwisów tak, aby kompilacja mogła odbywać się potokowo. Zaimplementowaliśmy oraz wystawiliśmy kilka komunikujących się ze sobą serwerów. Zadbaliśmy o to, by przebieg potokowej kompilacji odbywał się w odizolowanych kontenerach, dzięki czemu w przypadku złośliwego ataku, szkoda będzie dotyczyć jednego kontenera.

W dodatku w naszą architekturę wszczepiliśmy dodatkowe serwery, które mają za zadanie walidować plik na każdym etapie kompilacji. W celu walidacji sprawdzana jest poprawność formatu pliku. Kontrolowana jest długość niektórych napisów tak, by zapobiec buffer overflow attack, a także użycie znaków z ograniczonej puli. W końcu weryfikowane jest to, czy w wejściowym pliku używane są jedynie biblioteki spośród dozwolonych.

Nasze serwisy mają także połączenie z bazą danych tak, by przechowywać informację o aktualnym stanie pliku. Przechowywane są także daty modyfikacji plików, aby móc zmieniać stan pliku na błędny w przypadku zbyt długiego czasu oczekiwania na zakończenie kompilacji.

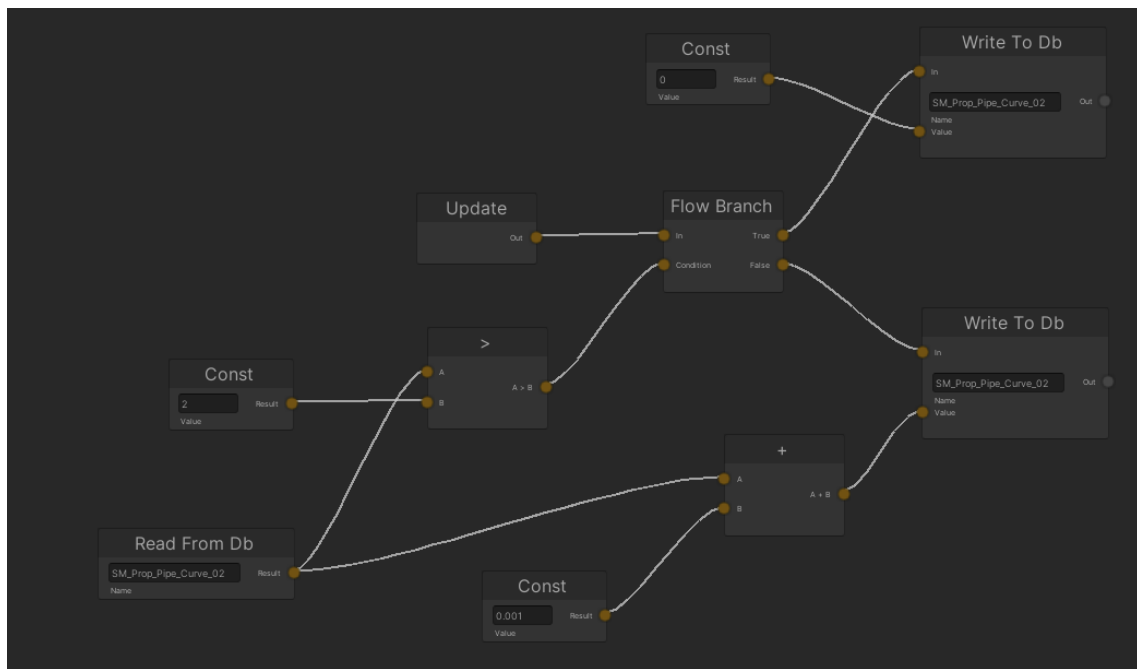
Klient ma możliwość przesłać plik YAML na wystawiony przez nas endpoint (rys. 8.1). Następnie może odpytywać requestami HTTPS o aktualny status pliku. Gdy podany input nie przeszedł któregoś z etapów walidacji, klient odczyta ze statusu informację, na którym etapie nastąpił błąd.



ScriptSending	
POST	/upload-script
GET	/compilation-state
GET	/compiled-file

Rysunek 8.1: Wystawione endpointy

Wejściowy plik YAML tworzony jest przy pomocy narzędzia ReadyCode do wizualnego edytowania skryptów (rys. 8.2). Ten plik przesyłany jest do zaimplementowanego przez nas serwera. Następnie kompilowany jest on do kodu w C# przy pomocy kompilatora utworzonego przez firmę ReadyCode. Następnie kod w C# kompilowany jest do plików DLL z użyciem kompilatora .NET.



Rysunek 8.2: Przykładowy mod poruszający rurą w wizualnym edytorze ReadyCode

```

1 using YooL.Logger;
2 using Ready.Helpers.Static;
3 using Ready.ECS.Runtime;
4 using Ready.Utils.Common;
5 using System.Numerics;
6 using Ready.Inject.Runtime;
7 using Ready.Inject.Build;
8 using Ready.Legacy.Core.App;
9
10 public class VisualScript : IApp
11 {
12     private IEntityDatabase _entityDatabase;
13     private void _EntryPointMethod0_Generated()
14     {
15         Chunk chunk0;
16         ChunkOffset chunkOffset1;
17         _entityDatabase.GetSecondaryIndex(new FieldTag<System.String>(new ArchetypeTag("sceneObject"), "sceneObjectName")).GetReader().GetChunk("SM Prop Pipe Curve 02", out chunk0, out chunkOffset1);
18         var property2 = _entityDatabase.GetProperty(new FieldTag<PositionRotation>(new ArchetypeTag("sceneObject"), "sceneObjectPosition")).GetRandomReadWriter();
19         var result3 = property2(chunk0, chunkOffset1).position.Y;
20         if ((2F) < (result3))
21         {
22             Chunk chunk4;
23             ChunkOffset chunkOffset5;
24             _entityDatabase.GetSecondaryIndex(new FieldTag<System.String>(new ArchetypeTag("sceneObject"), "sceneObjectName")).GetReader().GetChunk("SM Prop Pipe Curve 02", out chunk4, out chunkOffset5);
25             var property6 = _entityDatabase.GetProperty(new FieldTag<PositionRotation>(new ArchetypeTag("sceneObject"), "sceneObjectPosition")).GetRandomReadWriter();
26             var pos7 = property6(chunk4, chunkOffset5).position;
27             var rot8 = property6(chunk4, chunkOffset5).rotation;
28             property6(chunk4, chunkOffset5) = (new PositionRotation(new System.Numerics.Vector3(pos7.X, 0F, pos7.Z), property6(chunk4, chunkOffset5).rotation));
29             return;
30         }
31         else
32         {
33             Chunk chunk9;
34             ChunkOffset chunkOffset10;
35             _entityDatabase.GetSecondaryIndex(new FieldTag<System.String>(new ArchetypeTag("sceneObject"), "sceneObjectName")).GetReader().GetChunk("SM Prop Pipe Curve 02", out chunk9, out chunkOffset10);
36             var property11 = _entityDatabase.GetProperty(new FieldTag<PositionRotation>(new ArchetypeTag("sceneObject"), "sceneObjectPosition")).GetRandomReadWriter();
37             var result12 = property11(chunk9, chunkOffset10).position.Y;
38             Chunk chunk13;
39             ChunkOffset chunkOffset14;
40             _entityDatabase.GetSecondaryIndex(new FieldTag<System.String>(new ArchetypeTag("sceneObject"), "sceneObjectName")).GetReader().GetChunk("SM Prop Pipe Curve 02", out chunk13, out chunkOffset14);
41             var property15 = _entityDatabase.GetProperty(new FieldTag<PositionRotation>(new ArchetypeTag("sceneObject"), "sceneObjectPosition")).GetRandomReadWriter();
42             var pos16 = property15(chunk13, chunkOffset14).position;
43             var rot17 = property15(chunk13, chunkOffset14).rotation;
44             property15(chunk13, chunkOffset14) = (new PositionRotation(new System.Numerics.Vector3(pos16.X, (result12) + (0.001F), pos16.Z), property15(chunk13, chunkOffset14).rotation));
45             return;
46         }
47     }
48
49     public void OnEnter(IInjectionDomain runDom)
50     {
51         _entityDatabase = (ReadyApp.current.Resolve<IEntityDatabase>());
52     }
53
54     public void OnUpdate()
55     {
56         _EntryPointMethod0_Generated();
57     }
58
59     public void OnExit()
60     {
61     }
62
63     public void OnBuild(IInjectionDomain buildDom, IInjectionDomainBuilder runScope)
64     {
65     }
66 }

```

Rysunek 8.3: Przykładowy mod poruszający rurą w postaci kodu C# po pierwszym etapie kompilacji

Gdy status pliku wskazuje na to, że jest on w wynikowej postaci oraz że został zwalido-

wany z pozytywnym rezultatem, klient może pobrać ten plik. Później klientowi pozostaje już załadowanie pliku DLL do danej gry i korzystanie ze zmodyfikowanej przez siebie gry.

Aby przetestować oraz zwizualizować działanie zaimplementowanego przez nasz narzędzia, postanowiliśmy skompilować prosty mod do niezaawansowanej gry, używając wystawionego przez nas serwera. Taka gra została nam dostarczona przez firmę ReadyCode wraz z przykładowym wizualnym skryptem. Gra została stworzona jedynie, aby móc zaprezentować i przetestować wprowadzanie modyfikacji, dlatego jest bardzo podstawowa. W grze w przestrzeni 3D widzimy różne obiekty, które się nie poruszają. Natomiast, kiedy wgramy skompilowany mod, to poszczególne obiekty poruszają się. Wypróbowaliśmy mod, który powoduje poruszanie się rury, a także mod, który powoduje poruszanie się skrzynki.

Poniżej przedstawiamy zrzuty ekranu z przebiegu kompilacji wizualnego skryptu wraz z wyświetleniem logów dla niektórych kontenerów.



Rysunek 8.4: Żądanie HTTP wysyłające skrypt przedstawione w Swaggerze

>	i	2023-06-12 14:27:25.285 CEST	Request starting HTTP/1.1 POST http://34.116.231.70/upload-script multipart/form-data;+boundary=-----
>	i	2023-06-12 14:27:25.286 CEST	[40m[32minfo[39m[22m[49m: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
>	i	2023-06-12 14:27:25.286 CEST	Executing endpoint 'WebApi.Controllers.ScriptSendingController.UploadScript (WebApi)'
>	i	2023-06-12 14:27:25.287 CEST	[40m[32minfo[39m[22m[49m: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[3]
>	i	2023-06-12 14:27:25.287 CEST	Route matched with {action = "UploadScript", controller = "ScriptSending"}. Executing controller action wi
>	i	2023-06-12 14:27:26.257 CEST	[40m[32minfo[39m[22m[49m: WebApi.Controllers.ScriptSendingController[0]
>	i	2023-06-12 14:27:26.257 CEST	Uploaded visual script (b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.yaml) to bucket
>	!	2023-06-12 14:27:26.268 CEST	2023/06/12 12:27:26 New connection for "script-compilation-service:europe-central2:quickstart-instance"
>	i	2023-06-12 14:27:26.294 CEST	[40m[32minfo[39m[22m[49m: Microsoft.EntityFrameworkCore.Database.Command[20101]
>	i	2023-06-12 14:27:26.294 CEST	Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
>	i	2023-06-12 14:27:26.294 CEST	SELECT NOW()
>	i	2023-06-12 14:27:26.303 CEST	[40m[32minfo[39m[22m[49m: Microsoft.EntityFrameworkCore.Database.Command[20101]
>	i	2023-06-12 14:27:26.303 CEST	Executed DbCommand (4ms) [Parameters=[@p0='?' (DbType = Guid), @p1='?' (DbType = DateTime), @p2='?' (DbType
>	i	2023-06-12 14:27:26.303 CEST	INSERT INTO "Scripts" ("ScriptId", "CreatedAt", "Fail", "Form", "LastUpdatedAt")
>	i	2023-06-12 14:27:26.303 CEST	VALUES (@p0, @p1, @p2, @p3, @p4);
>	i	2023-06-12 14:27:26.548 CEST	[40m[32minfo[39m[22m[49m: Microsoft.AspNetCore.Mvc.Infrastructure.ObjectResultExecutor[1]
>	i	2023-06-12 14:27:26.548 CEST	Executing OkObjectResult, writing value of type 'WebApi.Models.UploadScriptResponse'.
>	i	2023-06-12 14:27:26.548 CEST	[40m[32minfo[39m[22m[49m: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[2]

Rysunek 8.5: Fragment logów w kontenerze serwera API (wgranie skryptu)

> i	2023-06-12 14:27:27.517 CEST	info: VisualScriptSanitizationStage.Workers.VisualScriptSanitizer[0]
> i	2023-06-12 14:27:27.517 CEST	Message ID: 8369604860416829 File name: b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.yaml
> i	2023-06-12 14:27:27.743 CEST	info: VisualScriptSanitizationStage.Workers.VisualScriptSanitizer[0]
> i	2023-06-12 14:27:27.743 CEST	Downloaded file to sanitize
> i	2023-06-12 14:27:27.943 CEST	fail: VisualScriptSanitizationStage.Workers.VisualScriptSanitizer[0]
> i	2023-06-12 14:27:27.943 CEST	Yamllint error for script b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3 :
> i	2023-06-12 14:27:27.943 CEST	fail: VisualScriptSanitizationStage.Workers.VisualScriptSanitizer[0]
> i	2023-06-12 14:27:27.943 CEST	Yamllint output for script b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3 :
> i	2023-06-12 14:27:28.202 CEST	info: VisualScriptSanitizationStage.Workers.VisualScriptSanitizer[0]
> i	2023-06-12 14:27:28.202 CEST	Published message 8369602834212464
> i	2023-06-12 14:27:28.448 CEST	info: VisualScriptSanitizationStage.Workers.VisualScriptSanitizer[0]
> i	2023-06-12 14:27:28.448 CEST	Published message 8369604132273703

Rysunek 8.6: Fragment logów w kontenerze sanityzacji skryptów wizualnych

> i	2023-06-12 14:27:29.106 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:29.106 CEST	to-csharp-worker	Message ID: 8369602834212464 File name: b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.yaml
> i	2023-06-12 14:27:29.323 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:29.323 CEST	to-csharp-worker	Downloaded file to local-b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.yaml
> i	2023-06-12 14:27:29.324 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:29.324 CEST	to-csharp-worker	Compiling file local-b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.yaml
> i	2023-06-12 14:27:30.820 CEST	to-csharp-worker	fail: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:30.820 CEST	to-csharp-worker	Compilation error:
> i	2023-06-12 14:27:30.821 CEST	to-csharp-worker	fail: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:30.821 CEST	to-csharp-worker	Compilation output:Input path is: `local-b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.yaml`
> i	2023-06-12 14:27:30.821 CEST	to-csharp-worker	Output path is: `/App/local-b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.cs`
> i	2023-06-12 14:27:30.821 CEST	to-csharp-worker	
> i	2023-06-12 14:27:31.725 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:31.725 CEST	to-csharp-worker	Uploaded b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.cs
> i	2023-06-12 14:27:32.031 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:32.031 CEST	to-csharp-worker	Deleted unmodified .vs file
> i	2023-06-12 14:27:32.250 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:32.250 CEST	to-csharp-worker	Published message 8369602799231935
> i	2023-06-12 14:27:32.321 CEST	to-csharp-worker	info: ToCSharpWorker.Workers.ToCSharpWorker[0]
> i	2023-06-12 14:27:32.321 CEST	to-csharp-worker	Published message 8369607780393674

Rysunek 8.7: Fragment logów w kontenerze kompilacji skryptów wizualnych

> i	2023-06-12 14:28:02.421 CEST	csharp-file-sanitization-stage...	info: CSharpFileSanitizationStage.Workers.CSharpFileSanitizer[0]
> i	2023-06-12 14:28:02.422 CEST	csharp-file-sanitization-stage...	Message ID: 8369602799231935 File name: b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.cs
> i	2023-06-12 14:28:02.660 CEST	csharp-file-sanitization-stage...	info: CSharpFileSanitizationStage.Workers.CSharpFileSanitizer[0]
> i	2023-06-12 14:28:02.660 CEST	csharp-file-sanitization-stage...	Downloaded file to sanitize
> i	2023-06-12 14:28:02.666 CEST	csharp-file-sanitization-stage...	info: CSharpFileSanitizationStage.Workers.CSharpFileSanitizer[0]
> i	2023-06-12 14:28:02.666 CEST	csharp-file-sanitization-stage...	File content:
> i	2023-06-12 14:28:02.670 CEST	csharp-file-sanitization-stage...	info: CSharpFileSanitizationStage.Workers.CSharpFileSanitizer[0]
> i	2023-06-12 14:28:02.670 CEST	csharp-file-sanitization-stage...	using Yooni.Logger;
> i	2023-06-12 14:28:02.670 CEST	csharp-file-sanitization-stage...	using Ready.Helpers.Static;
> i	2023-06-12 14:28:02.670 CEST	csharp-file-sanitization-stage...	using Ready.ECS.Runtime;
> i	2023-06-12 14:28:02.670 CEST	csharp-file-sanitization-stage...	using Ready.Utils.Common;
> i	2023-06-12 14:28:02.670 CEST	csharp-file-sanitization-stage...	using System.Numerics;
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	using Ready.Inject.Runtime;
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	using Ready.Inject.Build;
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	using Ready.Legacy.Core.App;
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	public class VisualScript : IApp
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	{
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	private IEntityDatabase __entityDatabase;
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	private void __EntryPointMethod0__Generated()
> i	2023-06-12 14:28:02.673 CEST	csharp-file-sanitization-stage...	{
> i	2023-06-12 14:28:02.676 CEST	csharp-file-sanitization-stage...	Chunk chunk0;
> i	2023-06-12 14:28:02.676 CEST	csharp-file-sanitization-stage...	ChunkOffset chunkOffset1;
> i	2023-06-12 14:28:02.676 CEST	csharp-file-sanitization-stage...	__entityDatabase.GetSecondaryIndex(new FieldTag<System.String>(new ArchetypeTag(

Rysunek 8.8: Fragment logów w kontenerze sanityzacji pliku C#

> i	2023-06-12 14:28:04.059 CEST	to-dll-worker	info: ToDLLWorker.Workers.ToDLLWorker[0]
> i	2023-06-12 14:28:04.059 CEST	to-dll-worker	Message ID: 8369603835298810 ScriptId: b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3
> i	2023-06-12 14:28:04.305 CEST	to-dll-worker	info: ToDLLWorker.Workers.ToDLLWorker[0]
> i	2023-06-12 14:28:04.305 CEST	to-dll-worker	Downloaded file to modify
> i	2023-06-12 14:28:04.305 CEST	to-dll-worker	info: ToDLLWorker.Workers.ToDLLWorker[0]
> i	2023-06-12 14:28:04.305 CEST	to-dll-worker	File content:
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	info: ToDLLWorker.Workers.ToDLLWorker[0]
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Yooni.Logger;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Ready.Helpers.Static;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Ready.ECS.Runtime;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Ready.Utils.Common;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using System.Numerics;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Ready.Inject.Runtime;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Ready.Inject.Build;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	using Ready.Legacy.Core.App;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	public class VisualScript : IApp
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	{
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	private IEntityDatabase __entityDatabase;
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	private void __EntryPointMethod0__Generated()
> i	2023-06-12 14:28:04.306 CEST	to-dll-worker	{

Rysunek 8.9: Fragment logów w kontenerze kompilacji pliku C#



>	i	2023-06-12 14:28:11.315 CEST	dll-file-sanitization-stage	info: DllFileSanitizationStage.Workers.DllFileSanitizer[0]
>	i	2023-06-12 14:28:11.315 CEST	dll-file-sanitization-stage	Message ID: 8369603574574086 File name: b9a8f03f-a06d-4ab1-b0a8-d3a7585c8fc3.dll
>	i	2023-06-12 14:28:11.560 CEST	dll-file-sanitization-stage	info: DllFileSanitizationStage.Workers.DllFileSanitizer[0]
>	i	2023-06-12 14:28:11.560 CEST	dll-file-sanitization-stage	Downloaded file to sanitize
>	i	2023-06-12 14:28:11.562 CEST	dll-file-sanitization-stage	info: DllFileSanitizationStage.Workers.DllFileSanitizer[0]
>	i	2023-06-12 14:28:11.562 CEST	dll-file-sanitization-stage	Modules assembly references sanitization has given positive result. Script: b9a8
>	i	2023-06-12 14:28:12.412 CEST	dll-file-sanitization-stage	info: DllFileSanitizationStage.Workers.DllFileSanitizer[0]
>	i	2023-06-12 14:28:12.412 CEST	dll-file-sanitization-stage	Uploaded sanitized file
>	i	2023-06-12 14:28:12.688 CEST	dll-file-sanitization-stage	info: DllFileSanitizationStage.Workers.DllFileSanitizer[0]
>	i	2023-06-12 14:28:12.688 CEST	dll-file-sanitization-stage	Deleted dll file
>	i	2023-06-12 14:28:12.762 CEST	dll-file-sanitization-stage	info: DllFileSanitizationStage.Workers.DllFileSanitizer[0]
>	i	2023-06-12 14:28:12.762 CEST	dll-file-sanitization-stage	Published message 8369604590636460

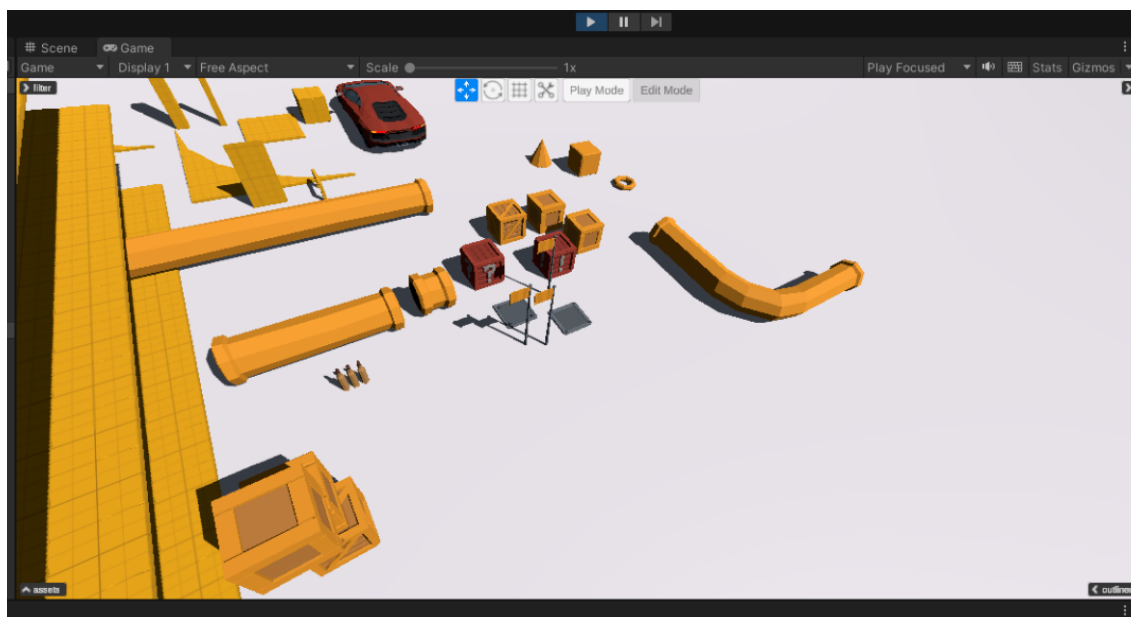
Rysunek 8.10: Fragment logów w kontenerze sanitzacji pliku DLL



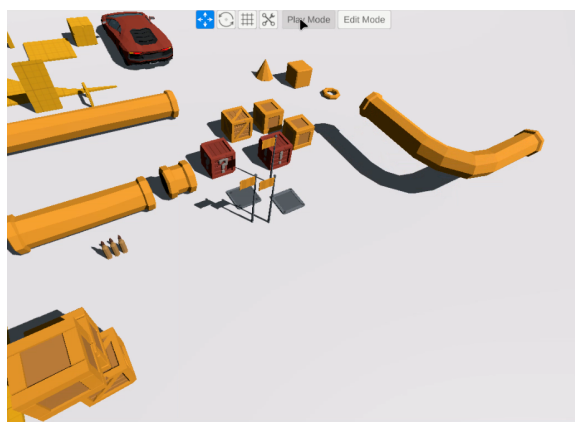
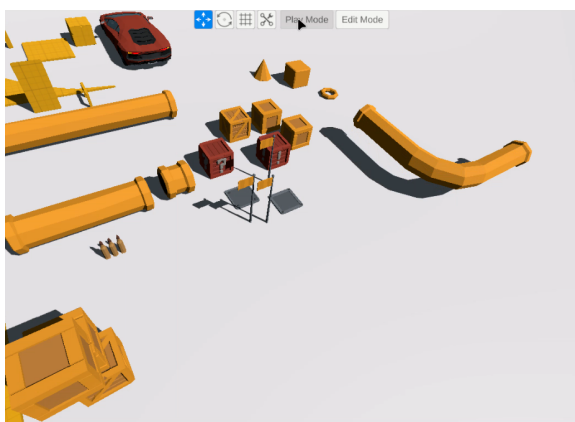
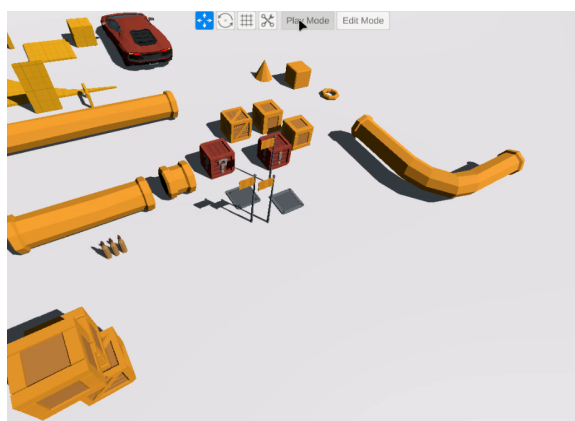
Rysunek 8.11: Żądanie HTTP pobierające skompilowany plik przedstawione w Swaggerze

Dostarczoną grę uruchamialiśmy w Unity i obserwowaliśmy efekt bez moda. Następnie przesyłaliśmy wizualny skrypt (plik YAML) na wystawiony przez nas serwer. Wówczas nasz serwer skompilował plik do postaci DLL. Pobrany z serwera plik DLL wkładaliśmy ręcznie do plików gry w odpowiednie miejsce, a następnie uruchamialiśmy zmodyfikowaną grę, czym osiągnęliśmy oczekiwany efekt.





Rysunek 8.12: Przykładowa gra, w wersji bez moda (wszystkie widoczne obiekty stoją w miejscu)



Rysunek 8.13: Przykładowa gra w wersji z modelem powodującym ruch rury

## Rozdział 9

# Organizacja pracy

W ramach organizacji pracy ustaliliśmy regularne spotkania z promotorem i firmą. W obu wypadkach cotygodniowe. W drugim semestrze do tego dołożyliśmy regularne spotkania wewnętrzne naszego zespołu, zależnie od możliwości zdalnie lub stacjonarnie. Do tego w ramach podziału zadań korzystaliśmy z platformy Asana do przydzielania zadań. Wzajemnie udostępnialiśmy i ocenialiśmy swój kod przez git.

Ostateczną wersję kodu udostępniliśmy w repozytorium o adresie:  
<https://gitlab.mimuw.edu.pl/jp418366/224755.git>

### **Karolina Laskowska**

- Research technologii deployowania
- Projektowanie architektury
- Sanityzacja (YAML, DLL)
- Wystawienie produktu na GCP z użyciem Kubernetesa
- Migracje bazy danych na produkcji
- Implementacja DbUpdate
- Połączenie serwisów z Pub/Sub

### **Izabela Ożdżeńska**

- Projektowanie architektury
- Restrukturyzacja i uwspólnienie kodu projektu
- Podłączenie bazy danych
- Konfiguracja Dockerfile'a kompilatora VS → C#
- Uruchomienie i integracja narzędzi ReadyCode (kompilatora VS → C#)

**Kamil Bugała**

- Sanityzacja pliku C#
- Uruchomienie endpointów API (wysyłanie danych przez plik)
- Podłączenie sanityzatorów
- Obsługa błędów - dodanie różnych statusów

**Jan Paszkowski**

- Konfiguracja Dockerfile'a kompilatora C# → DLL
- Podłączenie kompilatora C# → DLL
- Uruchomienie endpointów API (sprawdzanie stanu, pobieranie)

# Bibliografia

- [1] Autorzy Kubernetesa, *Performing a Rolling Update*,  
<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>,  
[6.06.2023].
- [2] Andrew Lock, *Running database migrations when deploying to Kubernetes*,  
<https://andrewlock.net/deploying-asp-net-core-applications-to-kubernetes-part-7-running-database-migrations/#using-kubernetes-jobs-and-init-containers>, [6.06.2023].
- [3] Microsoft, *.NET dependency injection*,  
<https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>,  
[6.06.2023].
- [4] Saptak Chaudhuri, *Architektura mikroserwisów*,  
<https://appmaster.io/pl/blog/architektura-mikroserwisow>, [6.06.2023].
- [5] Tim Butler, *What is a Dockerfile?*,  
<https://www.cloudbees.com/blog/what-is-a-dockerfile>, [6.06.2023].
- [6] Microsoft, *What are solutions and projects in Visual Studio?*,  
<https://learn.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2022>, [6.06.2023].
- [7] Autorzy Kubernetesa, *Kubernetes Components*,  
<https://kubernetes.io/pl/docs/concepts/overview/components/>, [6.06.2023].