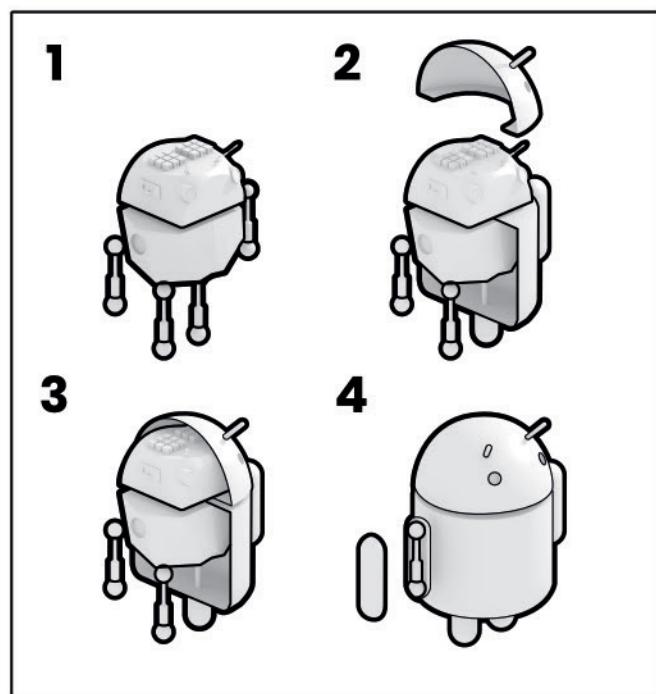
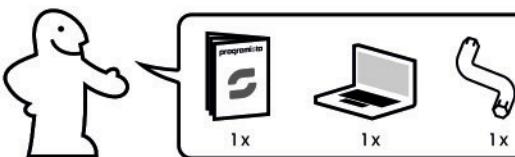
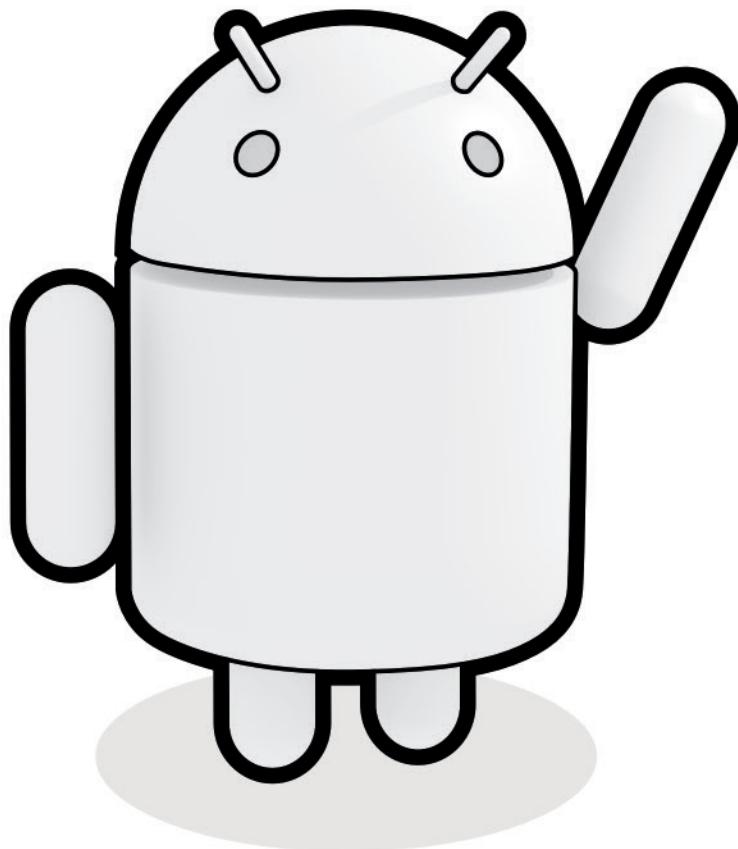


# ANDROID

## Jak stworzyć własną wersję systemu



ISSN 2084-9400



0.1

9 772084 940800

POWERED BY:

HUUUGE

TRANSAKCJE ROZPROSZONE  
W SPRING BOOT

NOWY STANDARD  
JĘZYKA C

ALGORYTMY  
EWOLUCYJNE

KONTENERYZACJA  
Z RANCHEREM



Join our team of Makers!

[career.cybercom.com](http://career.cybercom.com)

**Change  
tomorrow  
with us**

[www.cybercom.pl](http://www.cybercom.pl)



Cybercom Poland Sp. z o.o.

ul. Hrubieszowska 2, 01-209 Warszawa / ul. Składowa 35, 90-127 Łódź / ul. Unii Lubelskiej 4c, 85-059 Bydgoszcz

## Gdy coraz jaśniej za oknem...

Zbliżająca się wiosna jest okresem zmian. Wiele osób wykorzystuje ten całkowicie arbitralny moment, aby poczynić pewne postanowienia. Warto rozważyć wpisanie na tę listę takich pozycji jak „nauczę się czegoś nowego o informatyce”, „poczytam trochę o bezpieczeństwie” czy też „popracuję nad swoim aparatem matematycznym”. Jeszcze lepiej, jeśli będziecie robić to wspólnie z magazynem „Programista”, który jak co miesiąc stara się dostarczać przekrój artykułów na temat wszystkiego, co da się zaprogramować.

Lekturę proponujemy rozpocząć od zapoznania się z głównym tematem numeru, jakim jest samodzielne tworzenie systemu opartego o Androida, omówione w artykule „Własna wersja Androida – nie tak trudno jak myślisz” autorstwa Piotra Jastrzębskiego. Po jego lekturze można się dowiedzieć, jak samodzielnie skompilować i uruchomić Androida. W artykule zawarto również przykład prostej modyfikacji systemu.

Dla tych, którzy planują utworzyć wieloplatformową aplikację przeznaczoną na urządzenia mobilne, przydatnym okazać się może artykuł na temat Xamarina autorstwa Dawida Boryckiego.

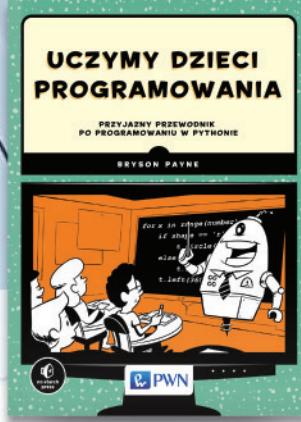
Polecamy również artykuł Pawła "KrzaQ" Zajączkowskiego, który uważnie śledzi rozwój języków C i C++. Tym razem można przeczytać o dobrzejsztwach planowanych w ramach nadchodzącego standardu C18.

Mamy nadzieję, że każdy znajdzie interesujący dla siebie tekst w tym wydaniu. Życzymy miłej lektury!

Mariusz „maryush” Witkowski,  
Michał Leszczyński

PS. Autorem projektu okładki do tego numeru jest Sebastian Rosik.

Polecamy!



## JĘZYKI PROGRAMOWANIA

### Co nowego w świecie języka C

Paweł "KrzaQ" Zajączkowski

### Transakcje rozproszone w Spring Boot

Paweł Kotula

## PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

### Własna wersja Androida – nie tak trudno jak myślisz

Piotr Jastrzębski

### Xamarin. Tworzenie wieloplatformowych aplikacji mobilnych

Dawid Borycki

## PROGRAMOWANIE SYSTEMOWE

### Wykrywanie zmian w systemach plików UNIX/Linux za pomocą skryptów

### BASH/Korn Shell i sum sprawdzających

Wojciech Bartosz Potentas

4

8

16

20

30

30

48

48

## PROGRAMOWANIE APLIKACJI WEBOWYCH

### Programowanie interaktywnych diagramów z wykorzystaniem GoJS

Tomasz Świdlik

## INŻYNIERIA OPROGRAMOWANIA

### Algorytmy ewolucyjne: wydajne metody rozwiązywania złożonych problemów optymalizacyjnych

Maciej Szymkowiak

40

48

## ZARZĄDZANIE PROJEKTAMI

### Konteneryzacja z Rancherem

Michał Pawlik

58

## LABORATORIUM TEINA

### Podglądaj i podsłuchuj, czyli jak testować oprogramowanie z użytkownikami

Katarzyna Małecka

68

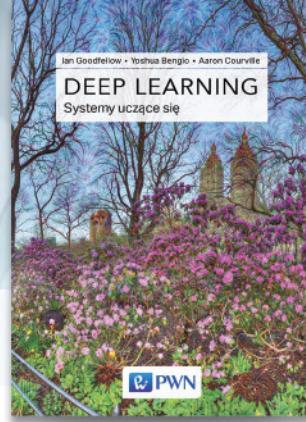
## KLUB DOBREJ KSIĄŻKI

### The Manga Guide. Mikroprocesory

Mariusz „maryush” Witkowski

70

reklama



ZAPISZ SIĘ NA NASZ NEWSLETTER I BĄDŹ NA BIEŻĄCO ➔

Odwiedź nas na:  
 **IT.PWN.PL**

 KSIEGARNIA.PWN.PL

# Co nowego w świecie języka C

C18 – tak naprawdę będzie brzmiała nazwa nowego standardu języka C. Aby tak się stało, komisja standaryzacyjna musi zakończyć nad nim prace na tyle prędko, aby organizacja ISO mogła go przyjąć jeszcze w tym roku. W tym artykule opisano wybrane zmiany spośród tych, które mają być wprowadzone.

**B**rak rozmów spowodowany jest tym, że jest to wydanie mające na celu wyłącznie poprawę błędów z poprzednich wersji. Większość modyfikacji zwiększa precyzję zapisów w samym dokumencie, cofa nieintencjonalne rozmijanie się ze standardem C++ w częściach wspólnych dla obu języków lub usuwa błędy.

## DOPRECZOWANIE INICJALIZACJI ZA POMOCĄ DESIGNATED INITIALIZERS

C99 wprowadziło możliwość inicjalizacji elementów struktur, unii i tablic w kolejności określonej przez programistę. Dla unii i struktur inicjalizacja odbywa się za pomocą zapisu:

» `.element = wartość_inicjalizująca`

a dla tablic:

» `[index] = wartość_inicjalizująca`

Ich przykładowe użycie znajduje się w Listingu 1.

**Listing 1. Przykładowe użycie designated initializers**

```
#include <assert.h>
#include <string.h>

typedef struct
{
    int num;
    double dbl;
    char const* str;
} foo;

int main(void)
{
    foo bar = { .str = "str!", .dbl = 42 };
    assert(bar.num == 0);
    assert(bar.dbl == 42);
    assert(strcmp(bar.str, "str!") == 0);

    int arr[3] = { [1] = 42 };
    assert(arr[0] == 0);
    assert(arr[1] == 42);
    assert(arr[2] == 0);
}
```

Przy takiej inicjalizacji wszystkie niewymienione jawnie elementy inicjalizowane są wartością zero. W bardziej zawiły przykładzie przedstawionym w Listingu 2 sytuacja nie jest już tak oczywista. Czy wartość `.t.k` to 42 (od inicjalizacji `.t = x`), czy może 0 (od niejawnnej inicjalizacji `.t` podczas inicjalizacji `.t.l`)?

**Listing 2. Przykład z raportu błędu (ang. Defect Report) [1]**

```
typedef struct {
    int k;
    int l;
```

```
    int a[2];
} T;

typedef struct {
    int i;
    T t;
} S;

T x = { .l = 43, .k = 42, .a[1] = 19, .a[0] = 18 };

void f(void)
{
    S l = { 1, .t = x, .t.l = 41, .t.a[1] = 17 };
}
```

Wedle komisji uważna lektura standardu pozwala wywnioskować, że jest to 42 – jawną inicjalizację jest ważniejsza od niejawnnej. Pomimo tego odpowiedni paragraf zostanie wzbogacony o przykład wyjaśniający tę wątpliwość.

## #ELIF Z NIEPOPRAWNYM WARUNKIEM

Zamierzeniem twórców języka była ekwiwalencja dyrektywy preprocesora `#elif` (Listing 3) z dyrektywą `#else` zawierającą wewnętrzną dyrektywę `#if` (Listing 4).

**Listing 3. Wewnętrzna dyrektywa #if**

```
#if 1
#else
#if EXPR
#endif
#endif
int main(void)
{}
```

**Listing 4. Dyrektywa #elif analogiczna do #if z Listingu 3**

```
#if 1
#elif EXPR
#endif
int main(void)
{}
```

Jeśli jednak EXPR będzie niepoprawnym wyrażeniem, zapis z Listingu 4 powinien spowodować błąd komilacji – co przedstawiono w Listingu 5.

**Listing 5. Program niepoprawny z powodu błędного warunku w pominiętej ścieżce [2]**

```
#if 1
#elif "
#endif
int main(void)
{}
```

# HUUUGE



## Join us!

Huuuge Games is waiting for you!

[huuugegames.com/work-with-us/](http://huuugegames.com/work-with-us/)



Od C18 oba warianty będą miały identyczne znaczenie – wyrażenie `#elif` będzie miało semantykę zdefiniowaną tak samo jak wyrażenie wewnętrznej dyrektywy `#if`.

## KONWERSJA LICZB ZESPOLONYCH NA \_BOOL

Zapisy poprzedniego standardu stały ze sobą w sprzeczności. Dla kodu z Listingu 6 jednocześnie aplikowalne są dwa zapisy standardu:

- » o konwersji do `_Bool` – wartość po konwersji jest równa 0, jeśli wartość konwertowana jest równa zero, w przeciwnym wypadku wartość po konwersji jest równa 1,
- » o konwersji liczb zespolonych do typów rzeczywistych – część urojona jest odrzucana, a część rzeczywista konwertowana do typu docelowego według normalnych reguł.

### Listing 6. Jaka powinna być wartość foo?

```
#include <complex.h>
#include <stdbool.h>

int main()
{
    _Bool foo = 0. + 1i;
}
```

Według pierwszego z wymienionych zapisów `foo` powinno mieć wartość 1, a według drugiego – 0. Autor nie był w stanie znaleźć wersji kompilatorów clang i gcc, które przejawiałyby drugie zachowanie. W C18 drugi z przytoczonych fragmentów o konwersji liczb zespolonych nie będzie dotyczył konwersji do `_Bool`, czyli `foo` będzie miało gwarantowaną wartość 1.

## PORZUCENIE ATOMIC\_VAR\_INIT

Wprowadzone w C11 obiekty `atomic` musiały być inicjalizowane za pomocą makra `ATOMIC_VAR_INIT` (Listing 7), inaczej ich wartość była traktowana jako niezdeterminowana<sup>1</sup>.

### Listing 7. Inicjalizacja atomic\_int z użyciem ATOMIC\_VAR\_INIT

```
#include <assert.h>
#include <stdatomic.h>
#include <threads.h>

int change(void* ptr)
{
    *(atomic_int*)ptr = 0;
    return 0;
}

int main(void)
{
    atomic_int x = ATOMIC_VAR_INIT(42);
    thrd_t thread;
    thrd_create(&thread, &change, &x);
    while(x);
    assert(x == 0);
    thrd_join(thread, NULL);
}
```

Ponieważ uniemożliwiało to użycie *designated initializers*, a popularne implementacje i tak implementowały to makro jako *no-op*<sup>2</sup>, zdecydowano się na porzucenie konieczności inicjalizacji w ten sposób. Od C18 legalna i poprawna będzie inicjalizacja przedstawiona w Listingu 8.

1. Nie dotyczy obiektów statycznych i globalnych.

2. Pusta instrukcja, brak jakichkolwiek działań. W tym przypadku oznacza to, że istniejące implementacje i tak implementowały makro `ATOMIC_VAR_INIT` jako ewaluujące się do swojego argumentu.

### Listing 8. Inicjalizacja atomic\_int jak zwykłej zmiennej int

```
#include <assert.h>
#include <stdatomic.h>
#include <threads.h>

int change(void* ptr)
{
    *(atomic_int*)ptr = 0;
    return 0;
}

int main(void)
{
    atomic_int x = 42;
    thrd_t thread;
    thrd_create(&thread, &change, &x);
    while(x);
    assert(x == 0);
    thrd_join(thread, NULL);
}
```

## ATOMOWY ŻART

Wbrew deklaracjom purystów językowych, w tym autora tego artykułu, C oraz C++ mają dużą część wspólną. Twórcy obu języków starają się też nie wprowadzać zbędnych rozbieżności między nimi. Dlatego też, jeśli jakieś wyrażenie pojawi się w opisie części wspólnej w jednym ze standardów, to z bardzo dużym prawdopodobieństwem zbliżone stwierdzenia można będzie również znaleźć w drugim.

Szkic standardu C++11, znany wtedy pod roboczą nazwą C++0x, zawierał, pośród innych, następujące gry słowne:

- » *Atomic objects are neither active nor radioactive* – obiekty `atomic` nie są ani aktywne, ani radioaktywne,
- » *Among other implications, atomic variables shall not decay* – pośród innych implikacji zmienne `atomic` nie powinny się rozpadać (ang. *decay*).

O ile pierwsze zdanie jest oczywistym żartem i nie zostało przeniesione do C11, to drugie może wyglądać jak typowe stwierdzenie, które można znaleźć w dokumencie standaryzacyjnym. Logicznie nie ma ono jednak sensu – *decay* oznacza w C niejawną konwersję typu tablicowego we wskaźnik na element. Pomimo tego znalazło się ono w C11. W C18 usunięto ten żart.

## PODSUMOWANIE

Choć w artykule przedstawiono wybrane najbardziej znaczące zmiany, to i tak można je traktować co najwyżej jako drobnostki. Gdyby nie kronikarskie zapędy autora, artykuł o C18 – o ile standard ten będzie ostatecznie nosił taką nazwę – mógłby nigdy nie powstać.

### Bibliografia:

[1]: <https://goo.gl/SVC9md>

[2]: <https://wandbox.org/permlink/gdTcEwoGxkJ8rsr>

### PAWEŁ "KRZAQ" ZAKRZEWSKI

<https://dev.krzaq.cc>

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Pracuje jako starszy programista C++ w firmie Huuge Games. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.



EXPO  
KRAKOW

28-29 MAY  
2018

## SPEAKERS:

- Peter Hedges
- Alex Mang
- Dror Helper
- Emil Wasilewski
- Dariusz Porowski
- Stefano Tempesta

DURING THE CONFERENCE

# Ask The Expert Zone

Powered by Strategic Sponsor:



Cloud DeveloperDays is a new conference prepared by the creators of .NET DeveloperDays.

You can expect the same seamless organization from an experienced team.

We are working hard on bringing you the best speakers in the industry.

# Transakcje rozproszone w Spring Boot

Coraz więcej organizacji, czy to bankowych czy przemysłowych, zaczyna rozumieć i doceniać korzyści, jakie niesie ze sobą wykorzystanie Spring Boot w procesie tworzenia oprogramowania. Jednocześnie chcąc iść z duchem czasu, nie mogą sobie pozwolić na utratę tak ważnych funkcjonalności, jakie do tej pory oferowały serwery aplikacyjne (ang. application servers) – mowa tutaj o transakcjach rozproszonych (ang. distributed transactions).

## TRANSAKCJE ROZPROSZONE – KRÓTKIE WPROWADZENIE

Zrozumienie, czym są transakcje rozproszone (ang. *distributed transactions*), należy poprzedzić przypomnieniem, czym w ogóle są transakcje. W ogólności są to jednostki przetwarzania, które charakteryzują się właściwościami określonymi mianem ACID:

- » Atomowość (ang. *atomicity*) – oznacza to, że wszystkie operacje wykonywane w ramach transakcji są wykonane albo w całości, albo też nie jest wykonana żadna z nich,
- » Spójność (ang. *consistency*) – zakończona transakcja pozostawia dane w spójnym stanie. Oznacza to, że jeżeli przed rozpoczęciem transakcji istniały pewne ograniczenia, to po zakończeniu te ograniczenia również są spełnione,
- » Izolacja (ang. *isolation*) – wykonywanie operacji w transakcji nie powinno wpływać na inne operacje wykonywane w innych transakcjach,
- » Trwałość (ang. *durability*) – zmiany wykonane na danych przez transakcję muszą być trwałe.

Transakcje rozproszone są to transakcje, które muszą spełniać reguły ACID oraz działają na wielu zasobach transakcyjnych. Warto zwrócić uwagę, że zazwyczaj o transakcjach mówi się w kontekście baz danych, natomiast takim zasobem, równie dobrze, może być broker JMS czy inny zasób implementujący javax.transaction.xa.XAResource. Samo omówienie działania koordynatora globalnego (ang. *global coordinator*), koordynatora lokalnego (ang. *local coordinator*), protokołu zatwierdzania dwufazowego (ang. *Two-Phase Commit*) i występujących w nim faz wykracza poza ramy tego artykułu. Skupimy się na wykorzystaniu gotowych rozwiązań do zarządzania transakcjami rozproszonymi (w naszym przypadku będzie to Atomikos) do stworzenia aplikacji przyjmowania zgłoszeń reklamacyjnych.

## CEL BIZNESOWY

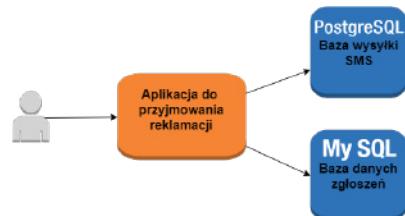
Wyobraźmy sobie sytuację, w której zostaliśmy poproszeni o stworzenie aplikacji przyjmującej zgłoszenia reklamacyjne przez formularz WWW. Aplikacja ta ma wysyłać powiadomienia SMS do użytkowników w chwili przyjęcia zgłoszenia oraz zapisywać inne potrzebne dane na potrzeby rozpatrzenia reklamacji. Okazało się, że funkcjonalność do wysyłki SMS jest już w naszej organizacji zaimplementowana i z powodzeniem wykorzystywana przez inne zespoły. Aby z niej skorzystać, wymagane jest jedynie dodanie re-

kordu do bazy danych PostgreSQL – reszta zostanie automatycznie przetworzona przez wewnętrzne mechanizmy, które są nieistotne z naszego punktu widzenia. Natomiast wszystkie dane potrzebne do przetworzenia reklamacji muszą znajdować się w bazie MySQL, ponieważ klient zakupił oprogramowanie generujące raporty, które wspiera jedynie ten silnik bazodanowy. Oczywiście niedopuszczalna jest sytuacja, w której uda się wysłać SMS, natomiast dane reklamacji nie zostaną umieszczone w bazie. Również zapisanie danych reklamacji bez powiadomienia klienta SMSem nie powinno się zdarzyć.

## CEL TECHNOLOGICZNY

Po zapoznaniu się z wymaganiami biznesowymi spróbujmy przełożyć je na język „techniczny”. Z technicznego punktu widzenia aplikacja powinna składać się z interfejsu użytkownika dostępnego przez stronę WWW oraz wspierać transakcyjny zapis do dwóch różnych źródeł danych (ang. *data source*). Intencją artykułu jest integracja Spring Boot oraz menedżera transakcji rozproszonych, zatem na te zagadnienia położymy największy nacisk. Sam formularz reklamacyjny dostępny na stronie internetowej nie wpływa na aspekty związane z działaniem transakcji, dlatego stworzymy jedyne prosty serwis RESTowy, który umożliwi nam odebranie żądania HTTP, a my skupimy się na jego transakcyjnym przetworzeniu i zapisaniu danych.

Architektura aplikacji do obsługi reklamacji pokazana jest na Rysunku 1.



Rysunek 1. Architektura aplikacji do przyjmowania reklamacji

## PRZYGOTOWANIE ŚRODOWISKA

Rozpoczęcie pracy nad aplikacją wymagać będzie od nas skonfigurowania dwóch różnych baz danych, w tym celu wykorzystamy narzędzie Docker (<http://docker.com>). Pozwala on na uruchomienie aplikacji (w naszym przypadku będą to dwie bazy danych) w wydzielonym kontenerze (ang. *container*) bez narzutu związane-

go z emulacją systemu operacyjnego (co czyni np. VirtualBox), tym samym zwiększa to ilość możliwych jednocześnie uruchomionych kontenerów na fizycznym sprzęcie. Uruchomione kontenery działają niezależnie od siebie, dzięki czemu w dowolnej chwili możemy dodawać/usuwać bądź modyfikować aktualnie potrzebne środowiska/aplikacje. Sama instalacja Dockera wybiega poza ramy tego artykułu – zakładamy, że mamy środowisko z dostępnym Dockerem, i skupiamy się na jego użyciu.

Na początek skorzystamy z publicznie dostępnych obrazów kontenerów PostgreSQL oraz MySQL do uruchomienia na środowisku developerskim.

Uruchomienie obrazu PostgreSQL:

#### Listing 1. Uruchomienie bazy PostgreSQL

```
$ docker run --name postgres-sms -e POSTGRES_PASSWORD=pass -p 5432:5432 -d postgres
```

Powyzsze polecenie uruchamia instancję postgresa z nazwą `postgres-sms` oraz hasłem `pass`, który jest dostępny z maszyny hosta na porcie 5432.

Następnie uruchamiamy obraz MySQL:

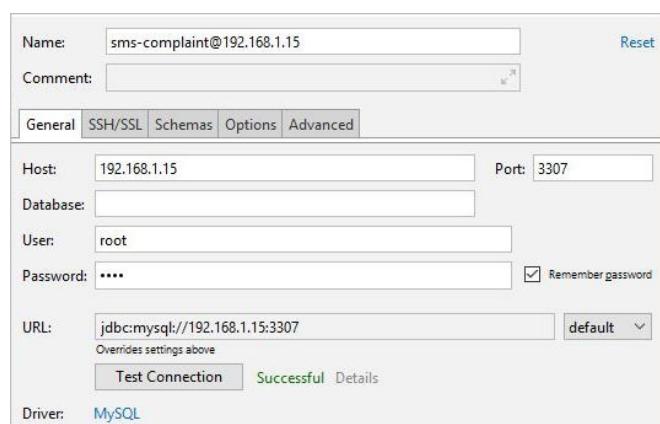
#### Listing 2. Uruchomienie bazy MySQL

```
$ docker run --name mysql-complaint -e MYSQL_ROOT_PASSWORD=pass -p 3307:3306 -d mysql:5.5.59
```

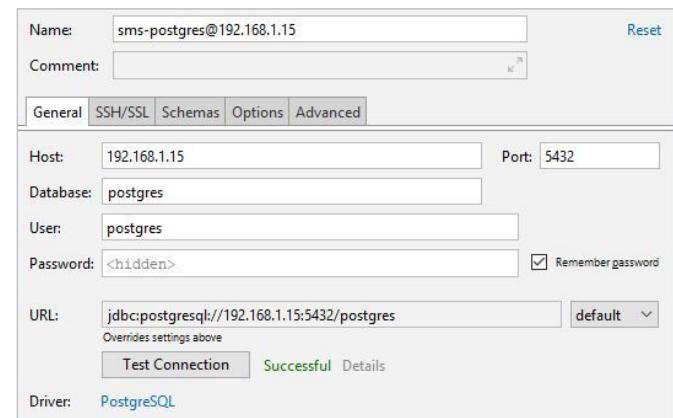
Powyzsze polecenie uruchamia instancję MySQL z nazwą `mysql-complaint` oraz hasłem `pass`, który jest dostępny z maszyny hosta na porcie 3307.

Sprawdźmy, czy nasze kontenery wystartowały prawidłowo, w tym celu wydajemy polecenie zawarte w Listingu 3.

Jak widać, kontenery bazodanowe wystartowały, spróbujmy się teraz podłączyć do uruchomionych baz danych zewnętrznym narzędziem:



Rysunek 2. Połączenie do bazy PostgreSQL



Rysunek 3. Połączenie do bazy MySQL

Same bazy danych zostały pomyślnie uruchomione, możemy przejść do stworzenia odpowiednich struktur tabel, które będą odpowiedzialne za przechowanie danych SMS oraz danych reklamacji:

#### Listing 4. Struktura bazy SMS – PostgreSQL

```
CREATE TABLE public.sms
(
    id SERIAL PRIMARY KEY NOT NULL,
    sms_text VARCHAR(160) NOT NULL,
    phone_number VARCHAR(9) NOT NULL,
    insert_date TIMESTAMP NOT NULL
);
COMMENT ON COLUMN public.sms.id IS 'Primary key';
COMMENT ON COLUMN public.sms.sms_text IS 'SMS Text limited to 160 chars';
COMMENT ON COLUMN public.sms.phone_number IS 'Phone number where SMS should be send';
COMMENT ON COLUMN public.sms.insert_date IS 'When record was created';
COMMENT ON TABLE public.sms IS 'SMS Table';
```

#### Listing 5. Struktura bazy reklamacji – MySQL

```
CREATE TABLE complaint
(
    id INT PRIMARY KEY COMMENT 'Primary key' AUTO_INCREMENT,
    firstname VARCHAR(50) NOT NULL COMMENT 'First name of the user',
    lastname VARCHAR(50) NOT NULL COMMENT 'Last name of the user',
    complaint TEXT NOT NULL COMMENT 'Complaint data'
);
ALTER TABLE complaint COMMENT = 'Table that store complaint data from users';
```

## IMPLEMENTACJA

Rozpoczęcie pracy nad nową aplikacją często stawia przed autorami pytania, jakich narzędzi i frameworków użyć, aby szybko i wygodnie rozwijać aplikację oraz dodawać nowe funkcjonalności w miarę zmieniających się wymagań. Jeżeli chodzi o narzędzia automatyzujące budowanie kodu źródłowego do wynikowej postaci (jar czy war), warto rozważyć zastosowanie Gradle ([www.gradle.org](http://www.gradle.org)). Daje

#### Listing 3. Wystartowane kontenery bazodanowe

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
abe7c6ac2ef8 mysql:5.5.59 "docker-entrypoint..." 22 seconds ago Up 21 seconds 0.0.0.0:3307->3306/tcp mysql-complaint
68d6703bd969 postgres "docker-entrypoint..." 27 minutes ago Up 27 minutes 0.0.0.0:5432->5432/tcp postgres-sms
```

on możliwość tworzenia własnych zadań (ang. *tasks*) oraz rozbudowy o nowe funkcjonalności przy wykorzystaniu języka Groovy ([www.groovy-lang.org](http://www.groovy-lang.org)). Sama instalacja Gradle sprowadza się do pobrania odpowiedniej paczki (ang. *package*), rozpakowaniu jej do wybranego folderu oraz ustawienia odpowiednich zmiennych środowiskowych. Po wykonaniu tych kroków sprawdzamy, czy instalacja przebiegła poprawnie, wykonując odpowiednie polecenie w konsoli:

#### Listing 6. Sprawdzenie poprawności instalacji Gradle

```
$>gradle -version
-----
Gradle 4.4.1
-----
Build time: 2017-12-20 15:45:23 UTC
Revision: 10ed9dc355dc39f6307cc98fdb8cea314bdd381c
...
...
```

Po poprawnym wykonaniu powyższego polecenia (Listing 6) przechodzimy do stworzenia właściwego szkieletu aplikacji, wydając polecenie:

#### Listing 7. Stworzenie szkieletu aplikacji

```
$>gradle init --type java-application
BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Po wykonaniu polecenia w folderze pojawią się następujące pliki oraz foldery:

#### Listing 8. Struktura folderów projektu

```
$>dir /Os
...
2018-01-14 10:49    <DIR>        .gradle
2018-01-14 10:49    <DIR>        gradle
2018-01-14 10:49    <DIR>        src
2018-01-14 10:49            579 settings.gradle
2018-01-14 10:49            1 027 build.gradle
2018-01-14 10:49            2 260 gradlew.bat
2018-01-14 10:49            5 296 gradlew
2018-01-14 10:49            4 File(s)          9 162 bytes
...
...
```

Najistotniejszy folder to **src** – gdzie znajdują się pliki źródłowe aplikacji – tutaj zagnieźdzona struktura jest bliźniaczo podobna jak w przypadku skorzystania z narzędzia Apache Maven. Sam plik **build.gradle** pełni podobną rolę co plik **build.xml** w Maven – zawiera definicję zadań, pluginów, zależności i innych informacji wymaganych do automatyzacji procesu budowania. Kolejnymi plikami, które pojawiły się podczas generowania szkieletu aplikacji, są **gradlew.bat** oraz **gradlew** – są to pliki, które pozwalają na zbudowanie aplikacji nawet przez osoby (czy inne narzędzia do automatyzacji), które nie mają zainstalowanego Gradle na swoich środowiskach.<sup>1</sup>

Kolejnym krokiem jest dodanie pluginu Spring Boot do pliku **build.gradle** – ułatwi to pracę, pozwalając na budowanie wykonywalnej paczki \*.jar oraz uruchamianie aplikacji bezpośrednio

z IDE (ang. *integrated development environment*). W tym celu na początku pliku **build.gradle** dodajemy wpis:

#### Listing 9. Dodanie pluginu Spring Boot do konfiguracji

```
plugins {
    id 'org.springframework.boot' version '1.5.9.RELEASE'
}
```

Dodatkowo plugin ten automatycznie rozszerza aplikację do zarządzania zależnościami (ang. *Dependency Management Plugin*) i konfiguruje go, aby importował **spring-boot-starter-parent bom**, przez co automatycznie będzie zarządzał zależnościami podobnie jak w przypadku Mavena. Dla przykładu pozwala on pominać numer wersji przy deklaracji zależności. Wszystkie wersje użytych bibliotek będą dostosowane do wersji podanej w Listingu 9. Zdefiniujmy wymagane zależności, dodając do pliku **build.gradle** następujące wpisy w sekcji zależności (ang. *dependencies*):

#### Listing 10. Lista zależności wykorzystywanych w projekcie

```
...
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web') //1
    compile('org.springframework.boot:spring-boot-starter-jta-atomikos') //2
    compile('org.springframework:spring-jdbc') //3
    compile group: 'mysql', name: 'mysql-connector-java', version: '5.1.45' //4
    compile group: 'org.postgresql', name: 'postgresql', version: '42.2.0' //5
}
...
» //1 zawiera zależności potrzebne do stworzenia prostego serwisu RESTowego oraz wystartowania kontenera webowego, który będzie odpowiedzialny za przyjmowanie żądań do naszej aplikacji.
» //2 dodanie zależności od menedżera transakcji rozproszonych, w tym przypadku jest to Atomikos – Spring Boot wspiera również inne rozwiązania, np. Bitronix, Narayana.
» //3 zawiera zależności, które będą wykorzystywane przy dostępie do baz danych.
» //4 i //5 dodanie zależności sterowników bazodanowych – warto zwrócić uwagę na możliwe różne formy podawania danych po słowie kluczowym compile inne niż w trzech pierwszych przypadkach.
```

Po dodaniu wymaganych zależności możemy przejść do stworzenia kontrolera do obsługi żądań przychodzących przez API. W tym celu tworzymy nowy pakiet (ang. *package*) w ścieżce **src/main/java** o nazwie **com.complaint.controller** i klasę o nazwie **ComplaintController** (kod pokazany w Listingu 11).

#### Listing 11. Obsługa przychodzących żądań

```
package com.complaint.controller;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
class ComplaintController {
    private static final Logger logger = LoggerFactory.getLogger(ComplaintController.class);
}

```

1. Powstały dwa pliki służą odpowiednio do uruchomienia na środowiskach Windows oraz Unix/OS

```

@RequestMapping("/complaint")
public String createComplaint() {
    Logger.info("Complaint creation entered");
    return "OK";
}
}

```

Kolejnym krokiem jest przeniesienie automatycznie wygenerowanej przez Gradle klasy App do odpowiedniego pakietu (com.complaint), dodanie wymaganych przez Spring Boot adnotacji oraz wywołania metody run w metodzie głównej (ang. *main method*) – kod klasy przedstawiony w Listingu 12:

#### Listing 12. Klasa i metoda główna programu

```

package com.complaint;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}

```

Tak przygotowany projekt możemy uruchomić z konsoli, wydając polecenie:

#### Listing 13. Uruchomienie aplikacji

```
$> ./gradlew bootRun
```

W wyniku wykonania polecenia z Listingu 13 na konsoli pojawi się domyślny baner Spring Boota oraz logi załączone w Listingu 14:

#### Listing 14. Prawidłowo skompilowana i uruchomiona aplikacja

```

...
INFO 10048 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/complaint]}" onto public java.lang.String com.complaint.controller.ComplaintController.createComplaint()
...
INFO 10048 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
...

```

Przetestujmy, czy nasza aplikacja rzeczywiście odpowiada na żądania na porcie 8080, w tym celu użyjemy wtyczki do przeglądarki Chrome o nazwie Postman – i wyślemy żądanie na adres http://localhost:8080/complaint – po wysłaniu w logach aplikacji pojawia się wpis:

#### Listing 15. Wywołanie kontrolera przez API skutkuje odpowiednim wpisem w logach

```

...
INFO 2184 --- [nio-8080-exec-1] c.c.controller.ComplaintController : Complaint creation entered
...

```

Co udowadnia, że aplikacja działa prawidłowo oraz jest gotowa do dalszej rozbudowy. W tym celu stworzymy serwis odpowiedzialny za dodawanie danych do odpowiednich baz danych. Tworzymy interfejs ComplaintService oraz klasę ComplaintServiceImpl implementującą dany interfejs w pakiecie com.complaint.service

#### Listing 16. Interfejs serwisu do zapisu reklamacji

```

package com.complaint.service;
public interface ComplaintService {
    /**
     * Create new complaint
     * @param firstName - First Name
     * @param lastName - Last Name
     * @param complaint - Complaint Text
     * @param phoneNumber - Phone number where SMS will be send
     */
    void newComplaint(String firstName, String lastName, String complaint, String phoneNumber);
}

```

#### Listing 17. Implementacja serwisu do zapisu reklamacji

```

package com.complaint.service;
...
@Service
class ComplaintServiceImpl implements ComplaintService {
    private static final Logger logger = LoggerFactory.getLogger(ComplaintServiceImpl.class);

    private final ComplaintRepository complaintRepository;//1
    private final SmsRepository smsRepository;//2
    private final String smsText;//3

    //4
    ComplaintServiceImpl(ComplaintRepository complaintRepository,
    SmsRepository smsRepository, @Value("${${smsText}}") String smsText) {
        this.complaintRepository = complaintRepository;
        this.smsRepository = smsRepository;
        this.smsText = smsText;
    }

    //5
    @Override
    @Transactional
    public void newComplaint(String firstName, String lastName,
    String complaint, String phoneNumber) {
        logger.info("New complaint start persisting.");
        complaintRepository.storeData(firstName, lastName,
        complaint);
        smsRepository.storeSMS(phoneNumber, smsText);
        logger.info("New complaint end persisting.");
    }
}

```

- » //1 – interfejs do zapisu danych reklamacji.
- » //2 – interfejs do zapisu danych sms.
- » //3 – tekst wiadomości, której wartość pobierana jest z pary klucz=wartość o takiej samej nazwie klucza jak nazwa zmiennej z pliku application.properties znajdującego się w folderze src/main/resources.
- » //4 – wstrzyknięcie opisanych powyżej interfejsów i tekstu odbywa się przez konstruktor. Dzięki temu, że klasa posiada jeden konstruktor, nie jest konieczne używanie adnotacji @Autowired i Spring automatycznie wstrzyknie odpowiednie dane do finalnych pól klasy.
- » //5 – implementacja metody, na którą składa się zapis informacji w logach oraz wywołanie wstrzykniętych obiektów oraz danych z punktów //1, //2 i //3.

Same interfejsy wstrzyknięte do klasy ComplaintServiceImpl, czyli SmsRepository oraz ComplaintRepository, pokazane są w Listingach 18-21:

#### Listing 18. Interfejs ComplaintRepository

```

package com.complaint.repository;
public interface ComplaintRepository{

```

```
void storeData(String firstName, String lastName, String
complaint);
}
```

**Listing 19. Interfejs SmsRepository**

```
package com.complaint.repository;
public interface SmsRepository {
    void storeSMS(String phoneNumber, String smsText);
}
```

Natomiast sama implementacja tychże znajduje się poniżej:

**Listing 20. Implementacja interfejsu ComplaintRepository**

```
package com.complaint.repository;
...
@Repository
class ComplaintRepositoryImpl implements ComplaintRepository {
    private static final Logger logger = LoggerFactory.getLogger(ComplaintRepositoryImpl.class);

    private final NamedParameterJdbcOperations complaintJdbcTemplate;

    ComplaintRepositoryImpl(NamedParameterJdbcOperations complaintJdbcTemplate) {
        this.complaintJdbcTemplate = complaintJdbcTemplate;
    }

    @Override
    public void storeData(String firstName, String lastName,
    String complaint) {
        logger.info("Storing Complaint Data, First Name: {}, Last
        Name: {}, Complaint {}", firstName, lastName, complaint);
        final String sql = "insert into complaint(firstname,
        lastname, complaint) values (:firstName, :lastName,
        :complaint)";
        MapSqlParameterSource paramSource = new MapSqlParameterSource(
            "firstName", firstName);
        paramSource.addValue("lastName", lastName);
        paramSource.addValue("complaint", complaint);
        complaintJdbcTemplate.update(sql, paramSource);
    }
}
```

**Listing 21. Implementacja interfejsu SmsRepository**

```
package com.complaint.repository;
...
@Repository
class SmsRepositoryImpl implements SmsRepository {
    private static final Logger logger = LoggerFactory.getLogger(SmsRepositoryImpl.class);

    private final NamedParameterJdbcOperations smsJdbcTemplate;

    SmsRepositoryImpl(NamedParameterJdbcOperations smsJdbcTemplate) {
        this.smsJdbcTemplate = smsJdbcTemplate;
    }

    @Override
    public void storeSMS(String phoneNumber, String smsText) {
        logger.info("Storing SMS Data, Phone: {}, Text: {}",
            phoneNumber, smsText);
        final String sql = "insert into sms(sms_text, phone_number,
        insert_date) values (:smsText, :phoneNumber, :insertDate)";
        MapSqlParameterSource paramSource = new MapSqlParameterSource(
            "smsText", smsText);
        paramSource.addValue("phoneNumber", phoneNumber);
        paramSource.addValue("insertDate", new Date());
        smsJdbcTemplate.update(sql, paramSource);
    }
}
```

Klasy te są bliźniaczo podobne – obie wstrzykują przez konstruktor interfejs NamedParameterJdbcOperations, który służy do za-

pisywania danych do odpowiednich baz danych, natomiast sama implementacja metod opiera się na wywoaniu metody update z wstrzykniętego interfejsu z instrukcją SQL oraz mapą parametrów, które są bindowane do odpowiednich wartości w zapytaniu SQL.

Ostatnie ziarna (ang. beans), jakie musimy zdefiniować, to te, które wstrzykujemy do klas SmsRepositoryImpl oraz ComplaintRepositoryImpl, tutaj samo ich stworzenie polega na wykorzystaniu gotowych klas dostępnych w Springu – należy jedynie skonfigurować je, aby były widoczne jako ziarna – w tym celu tworzymy pakiet com.complaint.config, a w nim klasę AppConfig przedstawioną w Listingu 22.

**Listing 22. Konfiguracja źródeł dostępu do danych**

```
package com.complaint.config;
...
@Configuration
@EnableTransactionManagement
class AppConfig {

    @Bean
    @Primary
    @ConfigurationProperties("sms.datasource")
    DataSource createPostgresSqlDataSource() {
        return new AtomikosDataSourceBean();
    }
    @Bean
    @ConfigurationProperties("complaint.datasource")
    DataSource createMySqlDataSource() {
        return new AtomikosDataSourceBean();
    }
    @Bean
    NamedParameterJdbcOperations smsJdbcTemplate() {
        return new NamedParameterJdbcTemplate(createPostgresSqlDataSource());
    }
    @Bean
    NamedParameterJdbcOperations complaintJdbcTemplate() {
        return new NamedParameterJdbcTemplate(createMySqlDataSource());
    }
}
```

Adnotacja @Configuration powoduje, że klasa będzie przetwarzana przez kontener Springa jako źródło definicji ziaren, a obiekty zwracane z metod znajdującej się w tej klasie oznaczone adnotacjami @Bean będą rejestrowane w kontekście aplikacji Spring – zwróciły uwagę, że zarówno klasa, jak i metody deklarujące ziara nie muszą być publiczne.

Adnotacja @EnableTransactionManagement sprawia, że możemy używać adnotacji do zarządzania transakcjami – jest to odpowiednik przestrzeni nazw (ang. namespace) <tx:> w konfiguracji XML – ponieważ tutaj posiadamy konfigurację w postaci pliku java, konieczne jest dodanie tej adnotacji.

Adnotacja @Primary służy do wskazania, które ziarno ma być użyte w przypadku, gdy znajduje się więcej niż jeden kandydat do wstrzyknięcia danego typu, ponieważ definiujemy dwa ziarna, które zwracają jako rezultat DataSource. Innym sposobem uniknięcia problemów z wieloma ziarnami bez stosowania adnotacji @Primary jest dodanie wykluczenia (ang. exclude), aby nasze źródła danych nie były automatycznie konfigurowane przez Spring Boot, zmieniając w klasie App adnotację:

**Listing 23. Wyłączenie autokonfiguracji na źródłach danych**

```
...
@SpringBootApplication(exclude={DataSourceAutoConfiguration.
    class})
...
```

Adnotacja `@ConfigurationProperties` pozwala przenieść konfigurację do zewnętrznego pliku – w naszym przypadku jest to `application.properties`. Parametr w adnotacji mówi, z jakim prefiksem będą odczytywane parametry i wstrzykiwane do metody tworzącej ziarna `AtomikosDataSourceBean`.

Kolejnym plikiem w naszej aplikacji jest plik konfiguracyjny `application.properties`, konfiguracja źródeł danych pokazana jest w Listingu 24.

#### Listing 24. Konfiguracja źródeł danych

```
...
#XA Configuration SMS
sms.datasource.unique-resource-name=smsDatabase
sms.datasource.xa-data-source-class-name=org.postgresql.
xa.PGXADataSource
sms.datasource.xa-properties.user=postgres
sms.datasource.xa-properties.password=pass
sms.datasource.xa-properties.URL=jdbc:postgres
ql://192.168.1.15:5432/postgres
#XA Configuration Complaint
complaint.datasource.unique-resource-name=complaintDatabase
complaint.datasource.xa-data-source-class-name=com.mysql.jdbc.
jdbc2.optional.MysqlXADataSource
complaint.datasource.xa-properties.user=root
complaint.datasource.xa-properties.password=pass
complaint.datasource.xa-properties.URL=jdbc:mys
ql://192.168.1.15:3307/complaint?characterEncoding=UTF-8
```

Jak widać, konfiguracja poprzedzona jest odpowiednim prefiksem dla każdej z konfiguracji źródła danych (ang. `datasource`)

Same wartości odpowiadają za:

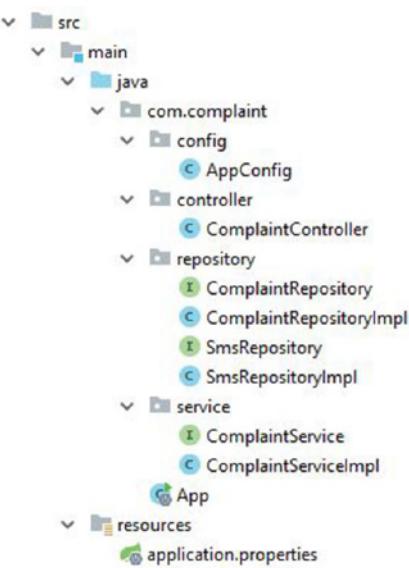
- » `<datasource_prefix>.unique-resource-name` – odpowiada za nadanie unikalnej nazwy każdemu źródłu danych, wymagane przez Atomikos w celu poprawnego działania.
- » `<datasource_prefix>.xa-data-source-class-name` – wskazuje klasę implementującą `javax.sql.XADatasource` – potrzebną, aby JTA mogło koordynować rozproszone transakcje.
- » `<datasource_prefix>.xa-properties.user` – użytkownik bazy danych.
- » `<datasource_prefix>.xa-properties.password` – hasło do bazy danych.
- » `<datasource_prefix>.xa-properties.URL` – URL połączenia do bazy danych, sterownik JDBC na tej podstawie wie, na jakim hoście szukać bazy danych, jak dana baza się nazywa oraz umożliwia przekazanie dodatkowych parametrów, które mogą się różnić w zależności od silników bazodanowych (ang. `database engine`).

Ostatnią modyfikacją, jaką należy wykonać, jest dodanie adnotacji `@Transactional` do serwisu `ComplaintServiceImpl`. Dokumentacja Spring zaleca dodawanie tej adnotacji na klasach implementujących interfejsy, ponieważ w języku Java adnotacje nie są dziedziczone z interfejsów, co w pewnych sytuacjach może powodować, że dany obiekt/metoda nie zostanie objęty transakcją.

#### Listing 24. Objęcie wywołania metody serwisu transakcją

```
...
@Override
@Transactional
public void newComplaint(String firstName, String lastName,
String complaint, String phoneNumber) {
...
```

Ostateczna struktura aplikacji pokazana jest na Rysunku 4.



Rysunek 4. Finalna struktura aplikacji

## TESTOWANIE APLIKACJI

Samo testowanie przeprowadzimy poprzez wywołanie odpowiedniej metody serwisu z parametrami przekazanymi w URL do naszego kontrolera.

W tym celu modyfikujemy klasę `ComplaintController`, dodając konstruktor wstrzykujący transakcyjny serwis obsługujący reklamacje (Listing 25).

#### Listing 25. Wstrzyknięcie klasy serwisu oraz dodanie parametrów do wywołania

```
...
@RestController
class ComplaintController {
    private static final Logger logger = LoggerFactory.
    getLogger(ComplaintController.class);
    private final ComplaintService complaintService;
    ComplaintController(ComplaintService complaintService) {
        this.complaintService = complaintService;
    }
    @RequestMapping("/complaint")
    public String createComplaint(@RequestParam String firstName,
        @RequestParam String lastName, @RequestParam String complaint,
        @RequestParam String phoneNumber) {
        logger.info("Complaint creation entered FirstName: {},",
        LastName: {}, Complaint: {}, PhoneNumber: {}", firstName,
        lastName, complaint, phoneNumber);
        complaintService.newComplaint(firstName, lastName,
        complaint, phoneNumber);
        return "OK";
    }
    ...
}
```

Ponownie uruchamiamy aplikację poleceniem z Listingu 13.

Następnie wchodzimy na adres URL z przeglądarki (czy narzędzia – np. POSTMAN) z odpowiednimi parametrami:

#### Listing 26. Wysłanie danych reklamacji do aplikacji

```
http://localhost:8080/complaint?firstName=Jan&lastName=Nowak&co
mplaint=Testowe Zgłoszenie&phoneNumber=123123123
```

Uwaga! Należy zwrócić uwagę, że wywołanie kontrolera z parametrami domyślną metodą GET ma na celu jedynie uproszczenie procesu testowania na potrzeby artykułu. Odpowiednią do tego celu metodą stosowaną w produkcyjnych aplikacjach powinna być metoda POST. Pominięte są tutaj również inne aspekty jak zabezpieczenie przed atakami, podsłuchem itp.

Niestety w odpowiedzi nasza aplikacja zamiast zwrócić pożądany rezultat „OK” zwraca:

#### Listing 27. Nieprawidłowy rezultat testowego zapytania

```
{  
    "status": 500,  
    "error": "Internal Server Error",  
    "exception": "org.springframework.transaction.  
UnexpectedRollbackException",  
    "message": "JTA transaction unexpectedly rolled back (maybe  
due to a timeout); nested exception is javax.transaction.  
RollbackException: Prepare: NO vote",  
    "path": "/complaint"  
}
```

W logach aplikacji (Listing 28) pojawia się informacja, co spowodowało błąd. Okazuje się, że winna jest konfiguracja bazy danych PostgreSQL.

#### Listing 28. Błąd konfiguracji bazy danych PostgreSQL

```
...  
Caused by: org.postgresql.util.PSQLException: ERROR: prepared  
transactions are disabled  
Hint: Set max_prepared_transactions to a nonzero value.  
...
```

Aby temu zaradzić, modyfikujemy nasz obraz Dockera sms-postgres, dodając wymagany parametr z podpowiedzi (ang. *hint*) błędu, i startujemy na nowo nasz obraz polecienniem z Listingu 29:

#### Listing 30. Zatwierdzenie transakcji na obu źródłach danych

```
c.a.datasource.xa.XAResourceTransaction : XAResource.commit ( 3139322E3136382E35362E312E746D30303035303030130:3139322E3136  
382E35362E312E746D38 , false ) on resource complaintDatabase represented by XAResource instance com.mysql.jdbc.jdbc2.optional.  
JDBC4MysqlXAConnection@dfbcd90  
c.a.datasource.xa.XAResourceTransaction : XAResource.commit ( 3139322E3136382E35362E312E746D30303035303030130:3139322E313  
6382E35362E312E746D39 , false ) on resource smsDatabase represented by XAResource instance org.postgresql.xa.PGXAConnection@  
ccdf836
```

#### Listing 31. Rollback na obu źródłach danych

```
c.a.datasource.xa.XAResourceTransaction : XAResource.rollback ( 3139322E3136382E35362E312E746D303030313030303131:3139322  
E3136382E35362E312E746D31 ) on resource complaintDatabase represented by XAResource instance com.mysql.jdbc.jdbc2.optional.  
JDBC4MysqlXAConnection@43ab3685  
c.a.datasource.xa.XAResourceTransaction : XAResource.rollback ( 3139322E3136382E35362E312E746D303030313030303131:3139322E313  
6382E35362E312E746D32 ) on resource smsDatabase represented by XAResource instance org.postgresql.xa.PGXAConnection@4767b78d
```



#### PAWEŁ KOŁUDA

pkoluda@gmail.com

Na co dzień zajmuje się tworzeniem oprogramowania na JVM. Fan stosowania dobrych praktyk i czystego kodu. Od wielu lat związany z aplikacjami dla sektora bankowego. Obecnie zatrudniony w GFT na stanowisku Senior Software Engineer, gdzie koordynuje prace zespołu migrującego aplikacje na Spring Boot. Poza pracą pasjonat jazdy na rowerze, kolekcjoner zegarków i modeli kamienic.

#### Listing 29. Dodanie wymaganych parametrów do wsparcia transakcji rozproszonej

```
docker run --name postgres-sms -e POSTGRES_PASSWORD=pass -p  
5432:5432 -d postgres -c max_prepared_transactions=100
```

Kolejna próba wywołania adresu URL kończy się powodzeniem, co można zaobserwować w logach aplikacji (Listing 30).

Sprawdźmy, czy wywołanie adresu URL z nieprawidłowymi danymi spowoduje, że aplikacja zachowa się w oczekiwany sposób i dane nie pojawią się w żadnej z baz danych. W tym celu podajemy telefon dłuższy niż może przechować kolumna phone\_number w tabeli sms. Kolumna ta została zdefiniowana jako varchar(9), przez co ma określona maksymalną długość na 9 znaków. Wywołajmy URL, podając parametr phoneNumber jako 123-123-123 (Listing 31).

Jak widać, dane zostały prawidłowo wycofane z obu baz danych, co dowodzi, że aplikacja wykorzystuje transakcje rozproszone.

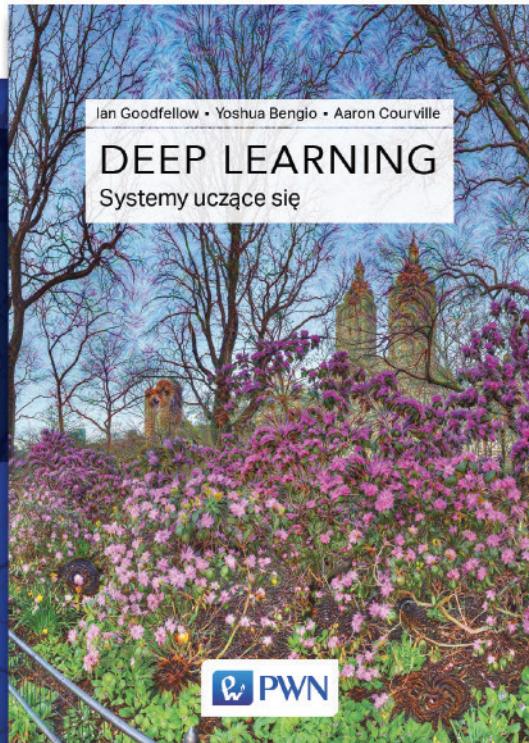
## PODSUMOWANIE

Samo stworzenie aplikacji, która wspiera transakcje rozproszone przy wykorzystaniu Spring Boot, jest zadaniem relatywnie prostym. Istotną rzeczą, na którą należy zwrócić uwagę, jest jedynie odpowiednia konfiguracja. Częstym błędem jest tworzenie źródeł danych, które zapisują dane prawidłowo, ale nie biorą udziału w transakcji rozproszonej. Przy braku odpowiednich testów oraz wdrożenia błędnie działającej aplikacji, gdy dochodzi do sytuacji cofnięcia transakcji, prowadzi to do niespójności w różnych źródłach danych i trudnego w analizie dochodzenia i poszukiwania błędu.

#### W sieci:

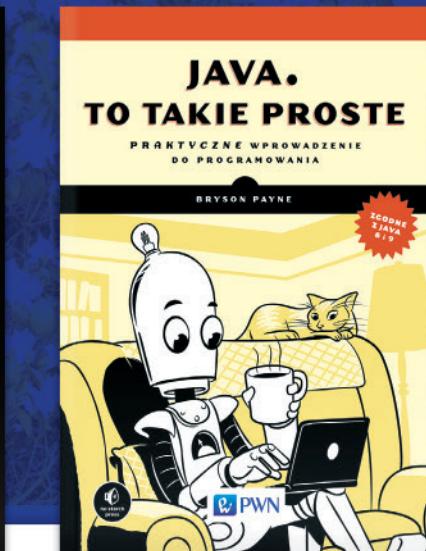
Kod źródłowy projektu: <https://goo.gl/hvRPt5>

# Najlepsze od



## JEDYNA TAKA KSIĄŻKA!

Polecamy:



Odwiedź nas na:  
**IT.PWN.PL**

Książki dostępne na: [www.ksiegarnia.pwn.pl](http://www.ksiegarnia.pwn.pl)

# Własna wersja Androida – nie tak trudno jak myślisz

Android to system operacyjny oparty na jądrze Linuksa, którego pierwotnym celem była obsługa smartfonów. Jego główną zaletą jest elastyczność pozwalającą uruchamiać go na różnych urządzeniach oraz dostosowywać jego wygląd i zachowanie do swoich potrzeb. Ta zdolność do adaptacji sprawiła, że Android jest dziś używany nie tylko na telefonach komórkowych, ale również na tabletach, telewizorach, zegarkach oraz w samochodach. Odgrywa też ważną rolę w branży Internetu Rzeczy (ang. Internet of Things). Według firmy Google, w maju 2017 na świecie było ponad 2 miliardy aktywnych urządzeń kontrolowanych przez system Android [1].

Ten artykuł ma na celu pokazanie, jak prostym procesem jest budowa własnej wersji Androida. Najpierw jednak niezbędna jest krótka dygresja na temat tego, czym jest Android. Android jest tak naprawdę rodziną systemów operacyjnych opartych na tej samej platformie. Platforma ta nazywa się Android Open Source Project (AOSP) i jej kod jest otwarty na licencji Apache 2.0. W oparciu o AOSP firma Google co roku wydaje swoją wersję Androida. Kod źródłowy tej wersji nie jest już publiczny, ale jest udostępniany producentom sprzętu (np. smartfonów) w celu adaptacji do sprzedawanych przez nich urządzeń. Producenci mają możliwość dokonania zmian w wyglądzie i zachowaniu systemu przez stworzenie własnej wersji opartej o Androida wydanego przez firmę Google. Na Rysunku 1 przedstawiono przykładowe zależności między różnymi wersjami Androida.



Rysunek 1. Przykładowe zależności między różnymi wersjami Androida

Proces budowy własnej wersji Androida, opisany w tym artykule, opiera się na platformie Android Open Source Project. Systemy operacyjne rekomendowane do budowania systemu Android to Ubuntu 14.04 (Trusty) i macOS v10.10 (Yosemite) lub późniejszy (z Xcode 4.5.2 i Command Line Tools). W tym artykule zilustrowano tworzenie własnej wersji w systemie macOS.

## STRUKTURA SYSTEMU

Koncepcyjnie system Android można podzielić na 6 części. Na Rysunku 2, który został zaczerpnięty z oficjalnej dokumentacji Androida [2], przedstawiono podział systemu na komponenty.

Podstawą całego systemu jest zmodyfikowane jądro Linuksa. Ciekawym rozwiązaniem jest wydzielenie warstwy abstrakcji sprzętu (Hardware Abstraction Layer, w skrócie HAL), które sprawia, że system nie polega na żadnym konkretnym sprzęcie i może być łatwo portowany na dowolne urządzenie. Dodatkowo implementacja HAL dla konkretnego hardware'u jest często własnością jego



Rysunek 2. Podział Androida na warstwy i komponenty

producenta i najczęściej ma zamknięty kod źródłowy. Kolejnym elementem układanki jest wsparcie dla programów napisanych w C i C++. Duża część platformy zaimplementowana jest w tych językach, więc wsparcie dla nich jest niezbędne. Następnym komponentem jest Android Runtime, na którym uruchamiane są aplikacje napisane w języku Java lub Kotlin. Jest to wirtualna maszyna wy-

konującą pliki DEX, które zawierają bajtkod opracowany specjalnie na potrzeby Androida. W ten sposób dochodzimy do środowiska zarządzającego aplikcjami (Java API Framework). Jest ono odpowiedzialne za kontrolowanie cyklu życia wszystkich aplikacji oraz stanowi dla nich jedyny punkt dostępu do reszty systemu, sprzętu oraz innych zasobów. Tutaj też znajduje się biblioteka graficzna odpowiedzialna za interfejs użytkownika. Na końcu znajdują się aplikacje napisane w języku Java lub Kotlin, które uruchamiane są przez środowisko zarządzające aplikcjami i dostarczają użytkownikowi konkretnych funkcjonalności typu zegarek czy klient email.

## TWORZENIE OBRAZU SYSTEMU

Przed rozpoczęciem budowania własnej wersji systemu Android należy upewnić się, że maszyna, którą będziemy do tego wykorzystywać, posiada:

1. 64 bitowy procesor,
2. przynajmniej 100 GB wolnego miejsca na dysku,
3. przynajmniej 16 GB RAM-u.

Domyślny system plików w systemie macOS ignoruje wielkość liter w nazwach plików i katalogów, co powoduje problemy podczas pracy z kodem źródłowym Androida. Z tego powodu rekomendowane jest stworzenie obrazu dysku, który nie ignoruje wielkości liter, zamontowanie go i praca na nim. Taki obraz można stworzyć komendą:

```
hdiutil create -type SPARSE -fs ,Case-sensitive Journaled HFS+' -size 100g ~/android.dmg
```

Ten obraz dysku można montować i odmontowywać następującymi komendami:

```
hdiutil attach ~/android.dmg.sparseimage -mountpoint /Volumes/android i hdiutil detach /Volumes/android
```

Przy okazji należy wspomnieć, że domyślny limit liczby otwartych plików jest zbyt mały i trzeba go podnieść za pomocą komendy ulimit -S -n 1024.

## Instalacja niezbędnego oprogramowania

Do skompilowania źródeł systemu Android potrzebny jest Xcode, który można zainstalować z App Store, oraz Command Line Tools, do instalacji którego służy komenda xcode-select --install. W instalacji reszty wymaganego oprogramowania pomoże nam program MacPort. Instrukcje do jego instalacji znajdują się na stronie [www.macports.org/install.php](http://www.macports.org/install.php). Ważną rzeczą jest, by upewnić się, że /opt/local/bin znajduje się w zmiennej środowiskowej PATH przed /usr/bin. Gdy mamy już dostępny na komputerze program MacPort, możemy przystąpić do instalacji właściwego oprogramowania, od którego zależy system Android. Służy do tego komenda:

```
POSIXLY_CORRECT=1 sudo port install gmake libsdl git gnupg1.
```

## Pobranie źródeł AOSP

Kod źródłowy systemu Android jest podzielony na wiele repozytoriów programu GIT. Do zarządzania tym zbiorem repozytoriów słu-

ży narzędzie o nazwie repo, które należy zainstalować, wykonując następujące komendy:

```
mkdir ~/bin
export PATH=~/bin:$PATH
curl https://storage.googleapis.com/git-repo-downloads/repo >
~/bin/repo
chmod a+x ~/bin/repo
```

Aby pobrać kod źródłowy dla ostatniej wersji Androida, należy wykonać następujące komendy:

```
cd /Volumes/android/
repo init -u https://android.googlesource.com/platform/manifest
repo sync
```

Ostatnia komenda może zająć chwilę, gdyż powoduje ona ściągnięcie około 50 GB danych. W moim przypadku zajęło to ponad 50 minut.

## Pobranie sterowników o zamkniętych źródłach

Większość sterowników implementujących warstwę abstrakcji sprzętu ma zamknięte źródła i nie jest częścią AOSP. Z tego powodu trzeba je ściągnąć oddziennie. Sterowniki do wszystkich telefonów z serii Nexus i Pixel są dostępne na stronie [developers.google.com/android/drivers](https://developers.google.com/android/drivers). Pobrane sterowniki należy rozpakować w repozytorium kodu źródłowego systemu Android i uruchomić skrypt extract-<nazwa sterownika>.sh dla każdego z nich. W tym artykule będziemy budować obraz systemu dla generycznego emulatora dostępnego wraz z ze źródłami Androida. Opcja ta nie wymaga ściągania żadnych dodatkowych sterowników.

## Budowanie obrazu systemu

By zbudować obraz systemu, należy najpierw wykonać komendę source build/envsetup.sh. Następnie trzeba wykonać polecenie lunch. Komenda ta posiada jeden argument, którym w naszym przypadku będzie aosp\_x86\_64-eng. Argument ten jest zależny od urządzenia, na które budujemy obraz, oraz od typu obrazu, jaki chcemy uzyskać. Argument ma formę <kod urządzenia>-<rodzaj obrazu>. Kody najpopularniejszych urządzeń przedstawiono w Tabeli 1.

Nazwa urządzenia	Kod urządzenia
Pixel XL	aosp_marlin
Pixel	aosp_sailfish
Nexus 6P	aosp_angler
Nexus 5X	aosp_bullhead
Nexus 6	aosp_shamu
Nexus Player	aosp_fugu
Nexus 9	aosp_flounder
Nexus 5	aosp_hammerhead

Tabela 1. Kody wykorzystywane w poleceniu lunch dla różnych urządzeń

Dostępne są 3 rodzaje obrazów:

1. **user** – zoptymalizowany obraz z ograniczonym dostępem, który jest gotowy na produkcję,
2. **userdebug** – podobny do user, ale z dostępem root'a oraz z możliwością debugowania,
3. **eng** – inżynierijny, nie zoptymalizowany obraz z dodatkowymi możliwościami do debugowania.

Na zakończenie należy wywołać make -jX z X zastąpionym przez dwukrotność dostępnych na komputerze wątków sprzętowych. Ta operacja jest bardzo czasochłonna, dlatego warto odpalić ją na noc. Na moim 4-korowym Intel Core i7 2,8 GHz komenda make -j8 zajęła prawie 3 godziny. Wynikiem tej operacji było 51 GB plików binarnych. Na Rysunku 3 przedstawiono screenshot podsumowanie komendy make.

```
#### build completed successfully (02:52:04 (hh:mm:ss)) ####
```

```
real    172m3.838s
user    1144m50.805s
sys     128m3.724s
```

Rysunek 3. Podsumowanie komendy make

## URUCHAMIANIE OBRAZU SYSTEMU

Po zbudowaniu obrazu Androida mamy dostęp do komendy emulator, która uruchamia zbudowany przez nas system na emulatorze. Jest to dość wygodna opcja, która jednak wiąże się z nieco

dłuższym oczekiwaniem, ponieważ emulator jest zazwyczaj wolniejszy niż fizyczne urządzenia. Na Rysunku 4 przedstawiono emulator uruchomiony ze zbudowaną przez nas wersją Androida.

Alternatywnie możemy wgrać obraz systemu na fizyczne urządzenie. Aby to zrobić, musimy przede wszystkim zbudować obraz dla tego konkretnego sprzętu. Do wyboru docelowego urządzenia służy komenda lunch opisana w poprzednim rozdziale. Następnie trzeba zbudować pomocnicze narzędzia przy użyciu polecenia make fastboot adb.

Urządzenie pozwala na wgranie systemu tylko jeśli jego bootloader jest odblokowany. Domyślnie bootloader jest zablokowany praktycznie na każdym sprzęcie. Można go odblokować, ale wiąże się to z utratą wszystkich danych użytkownika na urządzeniu. Aby odblokować sprzęt z serii Nexus i Pixel, należy wykonać następujące kroki:

1. W ustawieniach urządzenia wybrać „About phone” i kliknąć w „Build number” siedem razy.
2. Wybrać „Developer options” i włączyć „OEM unlocking” i „USB debugging”.
3. Zrestartuj urządzenie w trybie bootloadera (jak to zrobić, piękę poniżej) i wykonaj:
  - a. fastboot flashing unlock dla urządzeń wyprodukowanych w 2015 i później.
  - b. fastboot oem unlock dla urządzeń wyprodukowanych przed 2015.

Gdy sprzęt jest już odblokowany, można wgrać na niego obraz systemu po uruchomieniu go w trybie bootloadera, wykonując komendy adb reboot bootloader i fastboot flashall -w.

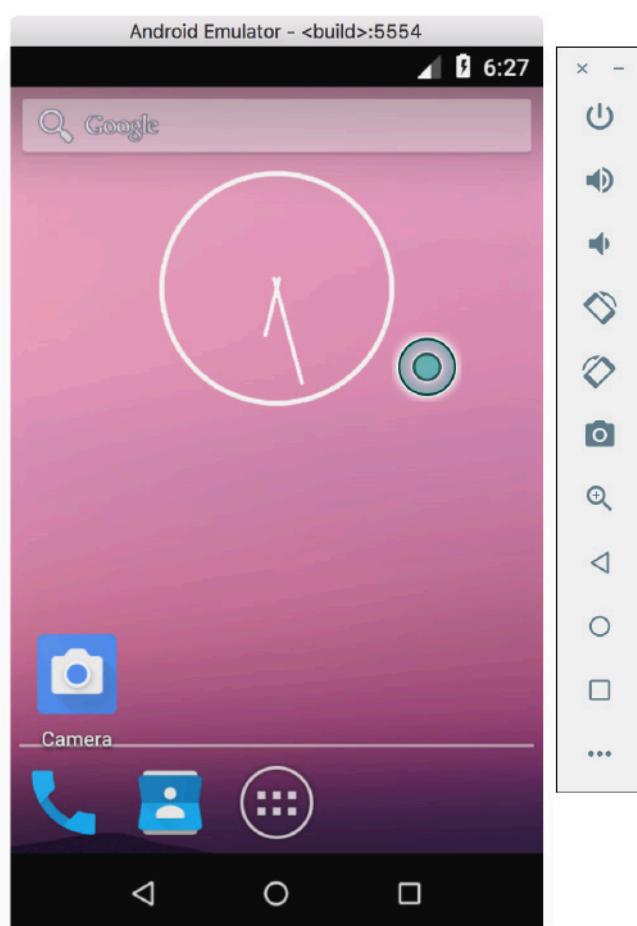
Urządzenia uruchamia się w trybie bootloadera poprzez przytrzymanie odpowiedniej kombinacji przycisków podczas włączania sprzętu. W Tabeli 2 opisano kombinacje powodujące uruchomienie w trybie bootloadera wybranych urządzeń z serii Nexus i Pixel.

Model	Kombinacja klawiszy
Pixel XL	Wcisnąć i przytrzymać „volume down” i kliknąć „power”
Pixel	Wcisnąć i przytrzymać „volume down” i kliknąć „power”
Nexus 6P	Wcisnąć i przytrzymać „volume down” i kliknąć „power”
Nexus 5X	Wcisnąć i przytrzymać „volume down” i kliknąć „power”
Nexus 6	Wcisnąć i przytrzymać „volume down” i kliknąć „power”
Nexus Player	Wcisnąć i przytrzymać „power”
Nexus 9	Wcisnąć i przytrzymać „volume down” i kliknąć „power”
Nexus 5	Wcisnąć i przytrzymać naraz „volume up” i „volume down”. Potem kliknąć „power”

Tabela 2. Kombinacje klawiszy powodujące uruchomienie wybranych urządzeń w trybie bootloadera

## PRZYKŁADOWA MODYFIKACJA

Umiejętność budowy własnej wersji Androida sprawia, że możemy dokonać zmian zachowania systemu. Na przykład możemy spra-



Rysunek 4. Emulator telefonu z własną wersją Androida

wić, żeby kliknięcie w element interfejsu użytkownika nie miało żadnego efektu poza wyświetleniem informacji „Na pewno? Jeśli tak, kliknij dwa razy”. Dopiero szybkie, dwukrotne kliknięcie spowoduje wykonanie docelowej akcji. Taka zmiana wymaga modyfikacji klasy android.view.View, której źródła znajdują się w pliku frameworks/base/core/java/android/view/View.java. Każdy element interfejsu użytkownika dziedziczy z tej klasy. Znajduje się w niej metoda performClick, która odpowiada za obsługę kliknięć. W Listingu 1 zawarto implementację tej metody.

**Listing 1. Metoda performClick obsługująca kliknięcia w interfejsie użytkownika**

```
/**  
 * Call this view's OnClickListener, if it is defined.  
 * Performs all normal actions associated with clicking:  
 * reporting accessibility event, playing a sound, etc.  
 *  
 * @return True there was an assigned OnClickListener  
 * that was called, false otherwise is returned.  
 */  
public boolean performClick() {  
    final boolean result;  
    final ListenerInfo li = mListenerInfo;  
    if (li != null && li.mOnClickListener != null) {  
        playSoundEffect(SoundEffectConstants.CLICK);  
        li.mOnClickListener.onClick(this);  
        result = true;  
    } else {  
        result = false;  
    }  
  
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);  
    notifyEnterOrExitForAutoFillIfNeeded(true);  
  
    return result;  
}
```

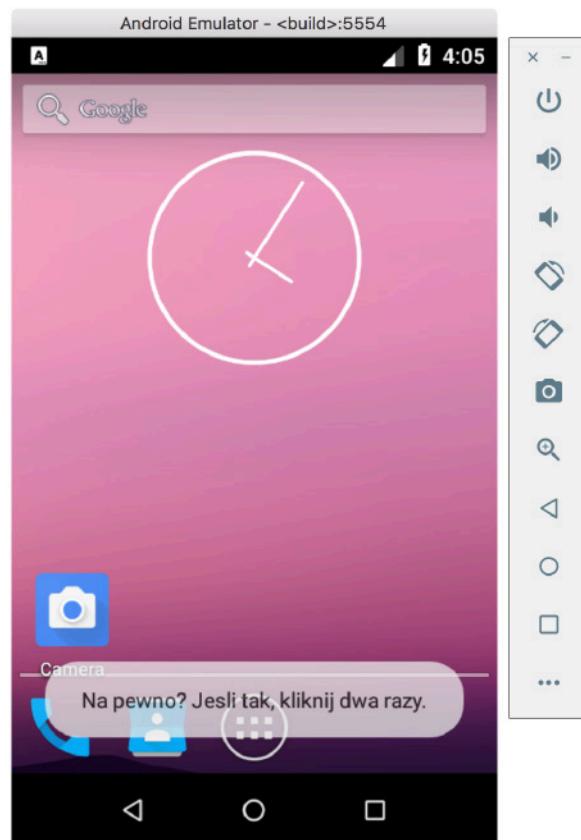
By osiągnąć oczekiwany efekt, wystarczy dodać do klasy View pole private long mLastClickTime, które będzie przechowywać czas ostatniego kliknięcia w dany element interfejsu. Następnie należy zmodyfikować metodę performClick, tak by korzystała z pola mLastClickTime. W Listingu 2 przedstawiono tę modyfikację.

**Listing 2. Modyfikacja metody performClick**

```
public boolean performClick() {  
    final boolean result;  
    final ListenerInfo li = mListenerInfo;  
    long millisInSecond = java.util.concurrent.TimeUnit.SECONDS.  
toMillis(1);  
    long currentTime = SystemClock.uptimeMillis();  
    boolean quickDoubleClick = currentTime - mLastClickTime <  
millisInSecond;  
    mLastClickTime = currentTime;  
    if (li != null && li.mOnClickListener != null) {  
        if (quickDoubleClick) {  
            playSoundEffect(SoundEffectConstants.CLICK);  
            li.mOnClickListener.onClick(this);  
            result = true;  
        } else {  
            android.widget.Toast.makeText(mContext.  
getApplicationContext(),  
            "Na pewno? Jesli tak, kliknij dwa razy.",  
            android.widget.Toast.LENGTH_LONG).show();  
            result = false;  
        }  
    } else {  
        result = false;  
    }  
}
```

```
sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);  
notifyEnterOrExitForAutoFillIfNeeded(true);  
return result;  
}
```

Zmodyfikowaną wersję Androida można zbudować i uruchomić w ten sam sposób co wersję bazową. Na Rysunku 5 przedstawiono emulator uruchomiony z systemem Android zawierającym naszą zmianę. Widzimy na nim efekt pojedynczego kliknięcia w jakiś element interfejsu użytkownika.



Rysunek 5. Emulator uruchomiony ze zmodyfikowanym systemem Android

## PODSUMOWANIE

Mam nadzieję, że powyższy artykuł udowodnił, że budowa systemu Android nie jest rzeczą trudną i tak naprawdę każdy może się tego podjąć w celach edukacyjnych i biznesowych.

## Referencje

- [1] <https://goo.gl/1LmjST>
- [2] <https://developer.android.com/guide/platform/index.html>

## PIOTR JASTRZĘBSKI

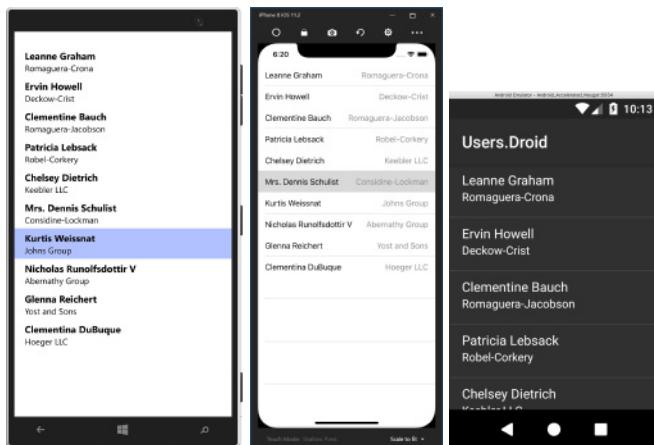
Inżynier oprogramowania z dziesięcioletnim stażem. Aktualnie pracuje nad rozproszoną bazą danych NoSQL – ScyllaDB. Wcześniej rozwijał real-time trading systemem w londyńskim City oraz pracował nad systemem Android w firmie Google. Absolwent informatyki na Uniwersytecie Warszawskim.

# Xamarin. Tworzenie wieloplatformowych aplikacji mobilnych

Istnieje bardzo wiele metod tworzenia wieloplatformowych aplikacji mobilnych. Począwszy od rozwiązań ściśle natywnych, dostarczanych przez twórców poszczególnych platform, poprzez rozwiązań wieloplatformowe, unifikujące języki i narzędzia programowania, aż do rozwiązań hybrydowych, w których aplikacje mobilne powstają w oparciu o techniki webowe i w pełni zunifikowane interfejsy programistyczne. W tym artykule chciałbym pokazać, w jaki sposób tworzyć natywne aplikacje mobilne przy użyciu narzędzi Xamarin oraz języka C#.

## WSTĘP

Firma Xamarin, będąca częścią firmy Microsoft, od 2011 roku dostarcza coraz to większą liczbę narzędzi programistycznych, których głównym celem jest uproszczenie procesu tworzenia aplikacji mobilnych. Główne hasło reklamowe z witryny Xamarin brzmi: „Everything you need to deliver great mobile apps” i doskonale tłumaczy misję firmy.



Rysunek 1. Aplikacja Xamarin-Users uruchomiona w emulatorach UWP, iOS oraz Android

Jednym z głównych narzędzi dostarczanych przez Xamarin są biblioteki Xamarin.iOS oraz Xamarin.Android. Opakowują one funkcjonalność dostarczaną przez natywne interfejsy programistyczne Apple (Xamarin.iOS) oraz Google (Xamarin.Android) w ramach dodatkowej cienkiej warstwy, utworzonej w oparciu o platformę .NET. W efekcie programista uzyskuje możliwość tworzenia aplikacji dla iOS i Androida z wykorzystaniem języków platformy .NET, a w szczególności C#. Rozwiązań to jest wyjątkowo wygodne dla programistów znających już platformę .NET i język C#. W efekcie końcowym mamy bowiem możliwość tworzenia aplikacji dla najbardziej popularnych platform mobilnych (iOS, Android oraz Uniwersalna Platforma Windows – UWP) przy użyciu jednego języka programowania oraz wspólnego kodu, gdyż .NET i C# są natywnymi narzędziami dla UWP.

W tym artykule pokażę, w jaki sposób zaimplementować aplikacje mobilne, przedstawione na Rysunku 1. Celem tych aplikacji jest pobranie listy użytkowników z serwisu REST, a następnie

prezentacja wybranych danych w widoku listy. W tym przypadku wspólna funkcjonalność tych aplikacji to oczywiście komunikacja z serwisem REST. Jak za chwilę pokażę, nie trzeba jej implementować niezależnie dla każdej z platform. Zamiast tego kod wspólny utworzymy w osobnej bibliotece z wykorzystaniem .NET Standard.

Narzędzia Xamarin były już opisywane na łamach „Programisty”. Warto tu przytoczyć artykuły:

- » Daniela Krzyczkowskiego (<https://goo.gl/GnfAJ5>),
- » Sylwestra Wieczorkowskiego (<https://goo.gl/MwFkQr>),
- » Rafała Świerkota (<https://goo.gl/38RbuU>),
- » oraz artykuły mojego autorstwa:
  - » <https://goo.gl/LrBMZS>,
  - » <https://goo.gl/d3zXFt>,
  - » <https://goo.gl/ijvPd3>.

Literatura ta traktowała głównie o Xamarin.Forms, nie pokazując jawnie, w jaki sposób korzystać z Xamarin.iOS, Xamarin.Android w połączeniu z UWP. Ten artykuł ma na celu uzupełnienie tej luki.

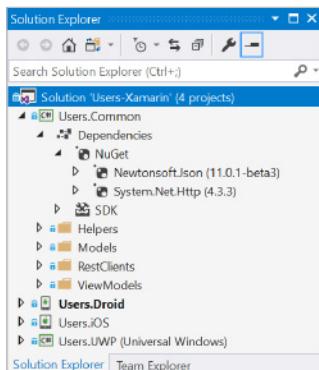
## UWAGI WSTĘPNE

Na Rysunku 2 zaprezentowana jest zawartość rozwiązania, które będziemy tworzyć w tym artykule (pełen kod jest dostępny pod adresem: <https://goo.gl/gJJeQg>). Mamy tam cztery projekty:

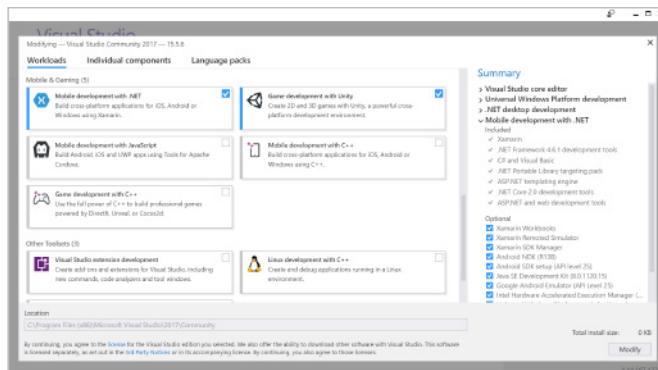
- » Users.Common – współdzielona biblioteka, która zawiera implementację części wspólnych,
- » Users.Droid – aplikacja dedykowana systemowi Android,
- » Users.iOS – projekt aplikacji dla iOS,
- » Users.UWP – aplikacja dla Uniwersalnej Platformy Windows.

Należy podkreślić, że mamy trzy niezależne projekty, dedykowane konkretnym platformom. Implementują one interfejs użytkownika oraz elementy specyficzne dla danej platformy. Część wspólna (jak np. dostęp do serwisu REST) jest taka sama dla każdej z platform i znajduje się w projekcie Common.

Jeśli chodzi o narzędzia developerskie, to Xamarin można zainstalować niezależnie lub skorzystać z instalatora Visual Studio, w którym trzeba zaznaczyć elementy z grupy Mobile development with .NET (Rysunek 3). W jej skład wchodzą narzędzia zarówno dla iOS, jak i Android. Z kolei dla UWP instalujemy standardowo pakiet Universal Windows Platform development. Jako środowisko programistyczne wybrałem darmowe Visual Studio 2017 Community.



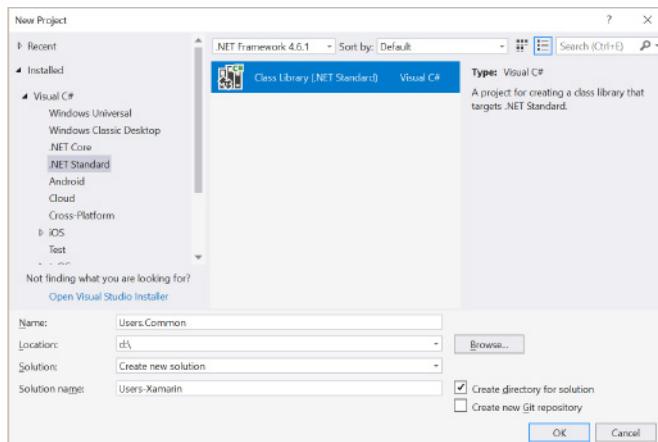
Rysunek 2. Struktura projektu Users-Xamarin



Rysunek 3. Instalacja składników Visual Studio

## IMPLEMENTACJA WSPÓŁDZIELONEJ FUNKCJONALNOŚCI

Tworzenie aplikacji rozpoczynamy od zaimplementowania części wspólnej, która będzie wykorzystywana w pozostałych projektach. W tym celu korzystamy z kreatora tworzenia nowego projektu Visual Studio (Rysunek 4), gdzie z listy dostępnych szablonów wybieramy *Class Library (.NET Standard)*, zmieniamy nazwę projektu na *Users.Common*, a w polu *Solution name* wpisujemy *Xamarin-Users*.

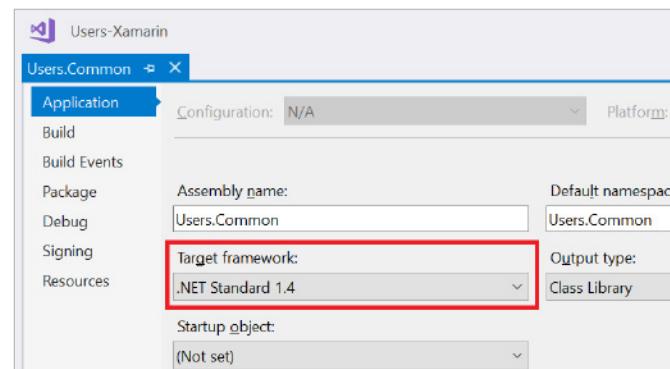


Rysunek 4. Tworzenie współdzielonego projektu według szablonu Class Library (.NET Standard)

Projekt utworzony w ten sposób składa się z jednej biblioteki .NET Standard, która będzie ładowana w pozostałych aplikacjach dla UWP, iOS i Android. Biblioteki .NET Standard były już opisywane na łamach „Programisty” (<https://goo.gl/WAkW1s>). W tym miejscu przypomnę jedynie, że .NET Standard stanowi formalną specyfiką

języka interfejsów programistycznych platformy .NET, która unifikuje przenośne biblioteki (ang. *Portable Class Libraries*, PCL) dla takich platform jak: .NET Core, Xamarin, ASP.NET, ASP.NET Core, UWP, Windows Phone itd. Dokładna lista wspieranych platform w zależności od wersji .NET Standard jest dostępna w formie tabeli pod adresem: <https://goo.gl/AmWFXv>. Zasada jest taka, że im niższa wersja, tym więcej wspieranych platform. Z kolei im wyższa wersja, tym więcej dostępnych funkcjonalności.

Na podstawie specyfikacji .NET Standard zdecydowałem, aby moja współdzielona biblioteka korzystała z .NET Standard w wersji 1.4. Tak aby wspierać UWP od wersji 10.0, Xamarin.iOS od wersji 10.0 i Xamarin.Android od wersji 7.0. W celu skonfigurowania wersji .NET Standard wystarczy wybrać odpowiedni element w liście rozwijanej *Target framework* we właściwościach projektu *Users.Common* (Rysunek 5).



Rysunek 5. Konfiguracja wersji .NET Standard

Po skonfigurowaniu projektu instalujemy dwa pakiety NuGet, które przyspieszą obsługę wysyłania zapytań i przetwarzania odpowiedzi z serwisu REST. Tymi pakietami są (zob. węzeł *Dependencies* na Rysunku 2):

- » *System.Net.Http*,
- » *Newtonsoft.Json*.

Mając te zależności, można przystąpić do właściwej implementacji, co rozpoczynamy od utworzenia klienta serwisu REST.

## Serwis REST i tworzenie klas C# na podstawie obiektów JSON

W tym artykule będę korzystał z zasobu users, dostarczanego przez witrynę JSONPlaceholder (<https://jsonplaceholder.typicode.com/>). Dla uproszczenia w dalszej części artykułu zasób users będę nazywał serwisem. Serwis users nie udostępnia pełnej funkcjonalności, bo nie aktualizuje listy użytkowników (żądanie PUT) ani nie dodaje nowych elementów do listy (żądanie POST), a jedynie potwierdza poprawne obsłużenie żądania (kod statusu HTTP zawsze jest 200). Pomimo tego serwis users w zupełności wystarcza do zaimplementowania aplikacji pokazanych wcześniej na Rysunku 1, gdyż prezentują one listę użytkowników. Aby pobrać listę tych użytkowników, wystarczy wysłać zapytanie typu GET. W jego wyniku uzyskuje się listę dziesięciu obiektów typu JSON. Każdy z nich ma strukturę przedstawioną w Listingu 1. Obiekt ten składa się z ośmiu właściwości: *id*, *name*, *username*, *email*, *address*, *phone*, *website* oraz *company*. *Address* oraz *company* są reprezentowane przez obiekty złożone. Pozostałe właściwości z wyjątkiem *id* (typ liczb całkowitych) są typu tekstowego.

**Listing 1. Struktura obiektów zwracanych przez serwis users z JSONPlaceholder**

```
{
  "id": 2,
  "name": "Ervin Howell",
  "username": "Antonette",
  "email": "Shanna@melissa.tv",
  "address": {
    "street": "Victor Plains",
    "suite": "Suite 879",
    "city": "Wisokyburgh",
    "zipcode": "90566-7771",
    "geo": {
      "lat": "-43.9509",
      "lng": "-34.4618"
    }
  },
  "phone": "010-692-6593 x09125",
  "website": "anastasia.net",
  "company": {
    "name": "Deckow-Crist",
    "catchPhrase": "Proactive didactic contingency",
    "bs": "synergize scalable supply-chains"
  }
}
```

Mając definicję z Listingu 1, można wygenerować odpowiednie klasy C#, które posłużą do niemal automatycznego parsowania odpowiedzi z serwisu users. Klasy C# można utworzyć ręcznie lub automatycznie z wykorzystaniem wybranego narzędzia. Ja skorzystałem z drugiej opcji i posłużyłem się witryną JSONUtils (<https://jsonutils.com/>), gdzie w pole tekstowe *JSON Text or URL* wkleiłem dane z Listingu 1. Następnie w polu *Class Name* wpisałem User i kliknąłem przycisk *Submit*. W efekcie serwis JSONUtils wygenerował cztery klasy: Users, Address, Company oraz Geo. Zapisałem je w osobnych plikach w podfolderze *Models* projektu Users. Common.

Wygenerowane obiekty nie wymagają dodatkowego komentowania, gdyż, jak to pokazano w Listingu 2, klasy te są złożone jedynie z automatycznie zaimplementowanych właściwości.

**Listing 2. Definicja klasy User, będąca odpowiednikiem obiektu JSON z Listingu 1**

```
public class User
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Username { get; set; }
  public string Email { get; set; }
  public Address Address { get; set; }
  public string Phone { get; set; }
  public string Website { get; set; }
  public Company Company { get; set; }
}
```

W kolejnym kroku przejdziemy do omówienia implementacji klasy klienta serwisu users.

## Klient REST

Klient REST serwisu users jest zaimplementowany w klasie *UsersClient* (*Users.Common/RestClients/UsersClient.cs*). Klasa ta posiada jedno prywatne pole typu *HttpClient*. Jest ono instancjonowane w ramach konstruktora klasy (Listing 3). Ta inicjalizacja ogranicza się do skonfigurowania adresu bazowego, w taki sposób, aby wskazywał na zasób users serwisu JSONPlaceholder, czyli następujący URL: <http://jsonplaceholder.typicode.com/users/>.

**Listing 3. Prywatne pole oraz konstruktor klasy *UsersClient***

```
private HttpClient httpClient;
public UsersClient()
{
  httpClient = new HttpClient();
  BaseAddress = new Uri(
    "http://jsonplaceholder.typicode.com/users/");
}
```

Klasa *UsersClient* posiada jedną publiczną metodę *GetAll* (Listing 4). Metoda ta wysyła do serwisu users zapytanie typu GET z wykorzystaniem metody *GetAsync* instancji klasy *HttpClient*. Wynikiem działania metody *GetAsync* jest instancja klasy *System.Net.Http.HttpResponseMessage*, która reprezentuje odpowiedź z serwisu REST.

**Listing 4. Publiczna metoda klasy *UsersClient***

```
public async Task<IEnumerable<User>> GetAll()
{
  var response = await httpClient.GetAsync(string.Empty);
  CheckStatusCode(response.StatusCode);
  return await DeserializeResponse<IEnumerable<User>>(response);
}
```

Klasa *HttpResponseMessage* posiada właściwość *StatusCode*, która przechowuje kod statusu HTTP. W tym przykładzie sprawdzam, czy ten status ma wartość 200 (*System.Net.HttpStatusCode.OK*). Do tego celu wykorzystuję funkcję pomocniczą *CheckStatusCode* (Listing 5). Jeśli ta funkcja nie zgłosi wyjątku, to przystępuję do deserializacji zawartości wiadomości HTTP z wykorzystaniem pomocniczej funkcji *DeserializeResponse* (Listing 5). Ta ostatnia działa w ten sposób, że najpierw odczytuje treść wiadomości w postaci łańcucha tekstowego JSON (*Content.ReadAsStringAsyncAsync*), a następnie wywołuje statyczną funkcję *DeserializeObject* klasy *JsonConvert*. *DeserializeObject* jest odpowiedzialna za przekształcenie łańcucha JSON do kolekcji obiektów typu *User* (Listing 2).

**Listing 5. Metody pomocnicze z klasy *UsersClient***

```
private static void CheckStatusCode(HttpStatusCode statusCode)
{
  if (statusCode != HttpStatusCode.OK)
  {
    throw new Exception(
      $"Unexpected status code: {statusCode}");
  }
}

private async Task<T> DeserializeResponse<T>(
  HttpResponseMessage response)
{
  var jsonString = await response.Content.ReadAsStringAsync();
  return JsonConvert.DeserializeObject<T>(jsonString);
}
```

W dalszej części artykułu klasa *UsersClient* (a faktycznie jej metoda *GetAll*) posłuży nam do pobierania listy elementów z serwisu users. Jednak zanim do tego przejdziemy, omówimy pozostałe składniki biblioteki *Users.Common*. Pierwszym z nich jest statyczna klasa *Check* (*Users.Common/Helpers/Check.cs*). Implementuje ona statyczną metodę *IsNull*, która służy do walidacji argumentów i sprawdza, czy dany obiekt został zainicjowany (Listing 6)

lub w dosłownym tłumaczeniu „czy odwołanie do obiektu zostało ustawione na wystąpienie obiektu”.

#### Listing 6. Definicja klasy Check

```
public static class Check
{
    public static void IsNull(object obj)
    {
        if(obj == null)
        {
            throw new ArgumentNullException();
        }
    }
}
```

Ostatnim składnikiem biblioteki `Users.Common` jest klasa `UsersViewModel` (Listing 7). Implementuje ona prostą klasę, która pełni funkcję view modelu. Mianowicie klasa `UsersViewModel` posiada właściwość `Users`. Właściwość ta jest typu `INotifyCollectionChanged<User>`, czyli jest to kolekcja obiektów typu `User`, implementująca interfejsy `INotifyCollectionChanged` oraz `IPropertyChanged`. Dzięki temu każdorazowe dodanie, usunięcie lub modyfikacja elementu kolekcji powoduje zgłoszenie powiadomienia, które może być wykorzystane przez widoki na potrzeby ich aktualizacji (mechanizm data binding). W rezultacie każdorazowa zmiana kolekcji `Users` może niemal automatycznie znajdować swoje odzwierciedlenie w UI aplikacji.

#### Listing 7. Definicja klasy UsersViewModel

```
public class UsersViewModel
{
    public ObservableCollection<User> Users { get; private set; }
    = new ObservableCollection<User>();

    private UsersClient usersClient = new UsersClient();

    public async Task RetrieveUsers()
    {
        var users = await usersClient.GetAll();

        Users.Clear();

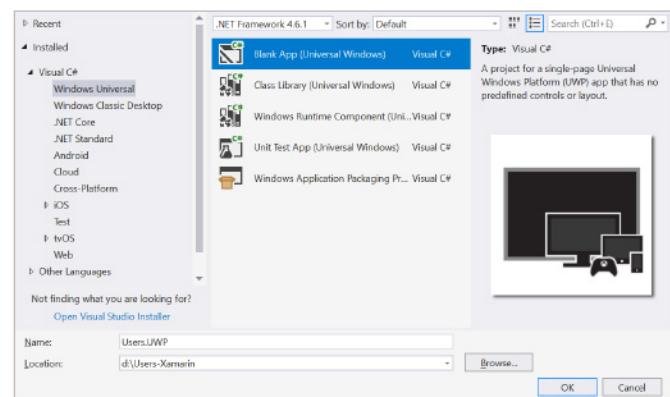
        foreach (User user in users)
        {
            Users.Add(user);
        }
    }
}
```

Po przygotowaniu części wspólnej przejdziemy do utworzenia aplikacji dla poszczególnych platform. Rozpoczniemy od Uniwersalnej Platformy Windows (UWP).

## APLIKACJA UWP

W celu utworzenia aplikacji UWP korzystamy z szablonu `Blank App (Universal Windows)` dla Visual C# (Rysunek 6). Zmieniamy nazwę projektu na `Users.UWP`. Po utworzeniu projektu musimy jedynie ustawić referencję do projektu `Users.Common`. Realizujemy to standardowo, czyli za pomocą menedżera referencji (Rysunek 7), w którym najpierw klikamy zakładkę `Projects`, a następnie z listy projektów wybieramy `Users.Common`.

Przystępujemy teraz do zdefiniowania głównego widoku aplikacji poprzez edycję pliku `MainPage.xaml`. Modyfikujemy go w taki sposób, aby zawierał jedną kontrolkę typu `ListView` (Listing 8). Jej zadaniem jest prezentacja wybranych właściwości obiektów typu `User`. Tymi właściwościami są: nazwa użytkownika (`Name`) oraz na-



Rysunek 6. Tworzenie projektu aplikacji `Users.UWP`



Rysunek 7. Konfigurowanie referencji projektu `Users.UWP`

zwa jego firmy (`Company.Name`). Sama kolekcja użytkowników jest pobierana z właściwości `Users` instancji klasy `UsersViewModel`.

Kod z Listingu 8 pokazuje, że stworzenie listy i powiązanie jej ze źródłem danych sprowadza się do skonfigurowania dwóch atrybutów klasy `ListView`:

- » `ItemsSource` – wskazuje źródło danych dla listy. W tym przypadku źródłem jest `usersViewModel.Users` (zob. Listing 7),
- » `ItemTemplate` – determinuje sposób wyświetlania poszczególnych elementów listy. W tym przykładzie sposób prezentacji jest określony za pomocą klasy `DataTemplate` (zob. zasoby widoku `Users.UWP.MainPage` z Listingiem 8). Klasa `DataTemplate` pozwala nam dowolnie ustalić sposób prezentacji elementów. Tutaj ograniczyłem się jedynie do wyświetlania dwóch pól tekstowych (dwie kontrolki typu `TextBlock`). Właściwości `Text` tych kontrolek są powiązane z właściwościami `Name` oraz `Company.Name` instancji klasy `User` w oparciu o mechanizm data binding. Jest on, podobnie jak atrybut `ItemsSource` kontrolki `ListView`, konfigurowany za pomocą rozszerzenia znaczników XAML o nazwie `{x:Bind}`. Warto zwrócić uwagę na atrybut `x:DataType` klasy `DataTemplate`. Używam go tutaj po to, aby móc jawnie zdefiniować typ danych prezentowany przez elementy listy (czyli instancje klasy `User`). Aby móc wskazać typ danych, uzupełniłem jeszcze deklarację klasy `Page` o następujący atrybut: `xmlns:models="using:Users.Common.Models"`. Wskazuje on przestrzeń nazw, w której zdefiniowałem typ `User`.

#### Listing 8. Deklaracja głównego widoku aplikacji `Users.UWP`

```
<Page
    x:Class="Users.UWP.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:models="using:Users.Common.Models">

    <Page.Resources>
        <DataTemplate x:Key="UserTemplate"
                      x:DataType="models:User">
            <StackPanel Orientation="Vertical">
```

```

        Margin="5">
    <TextBlock Text="{x:Bind Name}"
        FontSize="16"
        FontWeight="Bold"/>
    <TextBlock Text="{x:Bind Company.Name}"
        FontSize="14"/>
</StackPanel>
</DataTemplate>
</Page.Resources>

<ListView ItemsSource="{x:Bind usersViewModel.Users}"
    ItemTemplate="{StaticResource UserTemplate}"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"/>
</Page>

```

Po zdefiniowaniu widoku w pliku *code behind* wystarczy utworzyć instancję klasy *UsersViewModel* oraz wywołać jej metodę *RetrieveUsers*. Można to wygodnie zrealizować poprzez edycję klasy *MainPage* z pliku *MainPage.xaml.cs* (Listing 9).

**Listing 9. Definicja klasy MainPage**

```

public sealed partial class MainPage : Page
{
    private UsersViewModel usersViewModel = new UsersViewModel();

    public MainPage()
    {
        InitializeComponent();
    }

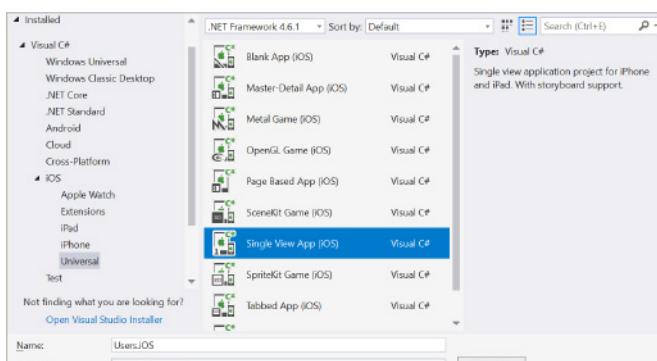
    protected async override void OnNavigatedTo(NavigationEventArgs e)
    {
        base.OnNavigatedTo(e);
        await usersViewModel.RetrieveUsers();
    }
}

```

Możemy teraz uruchomić aplikację, aby uzyskać efekt pokazany wcześniej z lewej strony Rysunku 1.

## APLIKACJA IOS

Przejdzmy teraz do aplikacji iOS. W tym przypadku korzystamy z szablonu Single View App (iOS) dla Visual C# (Rysunek 8). Z koleją nazwą projektu zmieniamy na *Users.iOS*. Po utworzeniu projektu dodajemy referencję do biblioteki *Users.Common*, po czym możemy przystąpić do właściwej implementacji. Jednakże komplikacja aplikacji iOS z poziomu Visual Studio, uruchomionego w środowisku Windows, wymaga połączenia ze zdalnym agentem Xamarin. Ten ostatni musi być uruchomiony na komputerze Mac z zainstalowanymi bibliotekami *Xamarin.iOS*, Xcode wraz z narzędziami developerorskimi dla iOS. To wszystko można wygodnie zainstalować za pomocą instalatora Visual Studio for Mac, co opisałem w poprzednim artykule (<https://goo.gl/LrBMZS>).



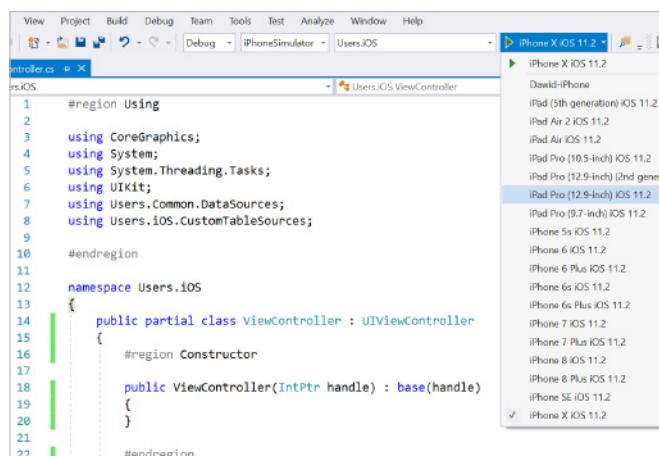
Rysunek 8. Tworzenie projektu aplikacji *Users.iOS*

Proces konfiguracji oraz połączenia z Xamarin Agent nie jest skomplikowany. W zasadzie podczas pierwszej próby komplikacji wyświetcone zostanie okno dialogowe, które prowadzi nas przez proces konfiguracji. Ponadto mechanizm ten jest dobrze udokumentowany (<https://goo.gl/4Hai45>). Z poziomu Visual Studio wybór konkretnego serwera zdalnej komplikacji poprzez Xamarin Agent realizuje się bardzo wygodnie z pomocą odpowiedniego okna dialogowego. Przykładową postać tego okna przedstawiłem na Rysunku 9. W moim przypadku w sieci lokalnej mam dostęp do jednej maszyny iMac.



Rysunek 9. Nawiązywanie połączenia z Xamarin Agent

Po nawiązaniu połączenia ze zdalnym agentem komplikacji możemy komplikować aplikację iOS na maszynie Windows. Co ciekawe, możemy również lokalnie testować aplikację z wykorzystaniem dowolnego emulatorka iOS zainstalowanego na zdalnej maszynie. Narzędzia Xamarin automatycznie odnajdują listę zainstalowanych emulatorów oraz trybów komplikacji. Są one prezentowane w odpowiednich listach rozwijanych na pasku narzędzi Visual Studio (zob. Rysunek 10).



Rysunek 10. Uruchamianie aplikacji w symulatorach iOS

W efekcie, chociaż fizycznie pracujemy na maszynie Windows, to mamy wygodne narzędzie do szybkiego i efektywnego tworzenia aplikacji iOS. Przejdzmy zatem do implementacji.

Aby utworzyć widok listy w iOS, korzysta się z kontrolki *UIKit*. *UITableView*. Można ją utworzyć statycznie poprzez edycję scenariusza (plik *Main.storyboard*) lub dynamicznie z wykorzystaniem metody *UIViewController.Add*. Stanowi ona alias dla funkcji *UIKit.UIView.AddSubview*. Ja będę korzystał z drugiej możliwości. Jednak niezależnie od wybranej metody tworzenia kontrolki *UITableView* należy wcześniej zaimplementować źródło danych dla listy. Wymaga to utworzenia klasy, która dziedziczy po *UIKit.UITableViewSource*. W tym przypadku zaimplementowałem taką klasę w pliku *Users.iOS/CustomTableSources/UsersTableViewSource.cs* (Listing 10).

**Listing 10. Pełna definicja klasy UsersTableSource implementującej źródło danych dla kontrolki UITableView**

```
public class UsersTableSource : UITableViewSource
{
    private IEnumerable<User> users;
    private const string cellId = "UserCell";
    public UsersTableSource(IEnumerable<User> users)
    {
        Check.IsNotNull(users);
        this.users = users;
    }
    public override UITableViewCell GetCell(
        UITableView tableView, NSIndexPath indexPath)
    {
        // Get item to display
        var user = users.ElementAt(indexPath.Row);

        // Try to reuse a cell before creating the new one
        var cell = tableView.DequeueReusableCell(cellId)
            ? new UITableViewCell(UITableViewCellStyle.Value1, cellId);

        // Configure cell properties
        cellTextLabel.Text = user.Name;
        cell.DetailTextLabel.Text = user.Company.Name;

        return cell;
    }
    public override nint RowsInSection(
        UITableView tableView, nint section)
    {
        return users.Count();
    }
}
```

UITableViewSource definiuje dwie metody, które muszą zostać zaimplementowane w klasie pochodnej (tutaj jest nią UsersTableSource). Tymi metodami są:

- » `GetCell` – metoda ta jest wywoływana każdorazowo, gdy zachodzi konieczność wyświetlenia lub odświeżenia wiersza kontrolki UITableView. Z tego powodu metoda `GetCell` zwraca instancję klasy UIKit.UITableViewCell. Ta ostatnia reprezentuje element listy. Szablon komórki można stworzyć w scenariuszu lub z poziomu kodu. SDK iOS dostarcza też kilka wbudowanych szablonów komórek. Wszystkie one są reprezentowane przez wartości typu wyliczeniowego UITableViewCellStyle. W Listingu 10 pokazałem przykładową implementację metody `GetCell`, która pobiera dane użytkownika z zapisanej wcześniej kolekcji `users`, a następnie tworzy komórkę tabeli według szablonu UITableViewCellStyle.Value1. Po utworzeniu komórki wypełniam ją danymi z instancji klasy User.
- » `RowsInSection` – metoda ta zwraca liczbę elementów (wierszy) tabeli. W tym przypadku liczba wierszy odpowiada całkowitej liczbie użytkowników zapisanych w polu `users` klasy UsersTableSource.

Oprócz wymaganych metod klasa UsersTableSource implementuje jeszcze konstruktor, który służy do uzyskania kolekcji użytkowników i zapisania jej w polu `users`. Warto zwrócić uwagę, jak łatwo można wykorzystać współdzieloną funkcjonalność w projektach dedykowanych konkretnym platformom. Tutaj bez najmniejszego problemu korzystam z klasy `Check`, aby zweryfikować poprawność argumentu przekazywanego do konstruktora klasy UsersTableSource.

Aplikacje iOS, podobnie jak Android, wspierają natywnie wzorzec Model-View-Controller (MVC). Z kolei UWP wspiera natywnie paradygmat Model-View-ViewModel (MVVM). Dlatego w przypad-

ku aplikacji UWP wiązanie źródła danych z UI przebiegło niemal automatycznie. Dla iOS musimy samodzielnie powiązać źródło danych (czyli kolekcję użytkowników pobraną z serwisu REST) z kontrolką UITableView poprzez zaimplementowaną wcześniej klasę UsersTableSource. Zrealizujemy to poprzez edycję pliku `ViewController.cs` z projektu `Users.iOS`.

Rozpoczynamy od zaimplementowania trzech metod pomocniczych (Listing 11):

- » `CreateUsersTableSource` – asynchroniczna metoda, która tworzy instancję klasy `UsersViewModel` i za jej pomocą pobiera listę użytkowników z serwisu REST. W ostatnim kroku ta lista jest wykorzystywana do utworzenia instancji klasy `UsersTableSource`. Alternatywnie metoda `CreateUsersTable`, zamiast korzystać z `UsersViewModel`, mogłaby uzyskać listę użytkowników bezpośrednio w oparciu o metodę `GetAll` klasy `UsersClient`. Jednak rozwiązanie oparte o klasę `view model` może, po zastosowaniu dodatkowych bibliotek (np. `MvvmCross`), posłużyć do zaimplementowania mechanizmu data binding w zgodzie z MVVM.
- » `CreateUsersTableView` – tworzy kontrolkę `UITableView`, a następnie konfiguruje jej właściwość `Source` w taki sposób, aby wskazywała na instancję klasy `UsersTableSource`. Dzięki temu utworzona tabela będzie prezentowała listę użytkowników.
- » `GetFrameWithVerticalMargin` – zwraca strukturę typu `CGRect`, określającą pozycję tabeli w widoku. Metoda `GetFrameWithVerticalMargin` przyjmuje jeden argument, który określa górny margines, liczony od górnej krawędzi ekranu. Wynik działania metody `GetFrameWithVerticalMargin` jest wykorzystywany do pozycjonowania kontrolki `UITableView`, aby elementy listy nie przykrywały paska statusu iOS.

**Listing 11. Metody pomocnicze klasy ViewController**

```
private async Task<UsersTableSource> CreateUsersTableSource()
{
    var usersViewModel = new UsersViewModel();
    await usersViewModel.RetrieveUsers();
    return new UsersTableSource(usersViewModel.Users);
}

private void CreateUsersTableView(UsersTableSource usersTableSource)
{
    // Create and display table view
    var table = new UITableView(GetFrameWithVerticalMargin(20));
    {
        Source = usersTableSource;
    };
    Add(table);
}

private CGRect GetFrameWithVerticalMargin(nfloat offset)
{
    var rect = View.Frame;
    rect.Y = offset;
    rect.Height -= offset;
    return rect;
}
```

Po przygotowaniu metod pomocniczych wykorzystujemy je do zaimplementowania metody zdarzeniowej `ViewDidLoad` klasy `ViewController` (Listing 12). Mianowicie najpierw wywołujemy metodę `CreateUsersTableSource`, a następnie wynik jej dzia-

łania (czyli instancej klasy `UsersTableSource`) przekazujemy do `CreateUsersTableView`. W efekcie końcowym, po uruchomieniu aplikacji, nastąpi wyświetlenie listy użytkowników (zob. środkowy zrzut ekranu z Rysunku 1).

**Listing 12. Tworzenie źródła danych i prezentacja listy użytkowników**

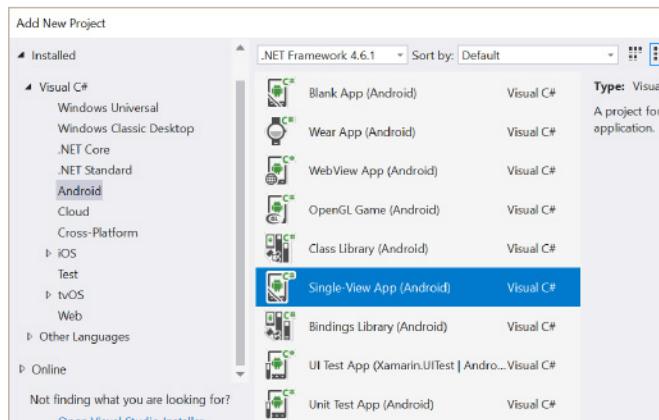
```
public async override void ViewDidLoad()
{
    base.ViewDidLoad();

    // Create users repository and use it to instantiate table
    // source
    var usersTableSource = await CreateUsersTableSource();

    // ... and use it to populate the table view
    CreateUsersTableView(usersTableSource);
}
```

## APLIKACJA ANDROID

Na koniec utworzymy aplikację dla Androida. W tym przypadku korzystamy z szablonu Single-View App (Android) dla Visual C# (Rysunek 11). Analogicznie jak poprzednio ustalamy nazwę projektu na `Users.Droid` i dodajemy referencję do `Users.Common`.



Rysunek 11. Tworzenie projektu aplikacji `Users.Droid`

W przypadku Androida proces implementacji widoku listy przebiega w pewnym sensie analogicznie jak w przypadku iOS. Mianowicie rozpoczęnamy pracę od zaimplementowania klasy, która pełni funkcję adaptera źródła danych. Zadaniem tego adaptera jest połączenie danych z kontrolkami. Innymi słowy, adapter determinuje sposób prezentacji elementów listy, czyli działa podobnie do klasy `UsersTableSource` z projektu `Users.iOS`.

**Listing 13. Definicja klasy `UsersListAdapter`**

```
public class UsersListAdapter : BaseAdapter<User>
{
    public override User this[int position]
        => users.ElementAt(position);
    public override int Count => users.Count();

    private IEnumerable<User> users;
    private Activity context;

    public UsersListAdapter(Activity context,
        IEnumerable<User> users) : base()
    {
        Check.IsNotNull(context);
        this.context = context;

        Check.IsNotNull(users);
        this.users = users;
    }

    public override View GetView(int position,
```

```
        View convertView, ViewGroup parent)
    {
        // Get item to display
        var user = this[position];

        // Try to reuse view before creating the new one
        var view = convertView;
        if (view == null)
        {
            view = context.LayoutInflater.Inflate(
                Android.Resource.Layout.SimpleListItem2, null);
        }

        // Configure cell properties
        view.FindViewById<TextView>(
            Android.Resource.Id.Text1).Text = user.Name;
        view.FindViewById<TextView>(
            Android.Resource.Id.Text2).Text = user.Company.Name;

        return view;
    }

    public override long GetItemId(int position)
    {
        return this[position].Id;
    }
}
```

Adapter listy użytkowników zaimplementowałem w klasie `UsersListAdapter` z pliku `Users.Droid/CustomAdapters/UsersListAdapter.cs`. `UsersListAdapter` dziedziczy po generycznej klasie `Android.Widget.BaseAdapter` (Listing 13). Ta ostatnia, podobnie jak `UITableViewDataSource`, wymaga, aby klasa pochodna (tutaj `UsersListAdapter`) implementowała następujące elementy:

- » `GetView` – metoda wywoływana w celu utworzenia elementu listy. Podobnie jak w przypadku iOS, również i Android udostępnia gotowe szablony elementów (jednak jest ich znacznie więcej niż w iOS). Te szablony są reprezentowane przez statyczne właściwości klasy `Layout`. W tym przykładzie korzystam z szablonu `SimpleListItem2`, co umożliwia mi wyświetlenie nazwy użytkownika oraz nazwy jego firmy. Warto zwrócić uwagę, że struktura metody `GetView` wygląda analogicznie jak `GetCell` w iOS. Mianowicie najpierw tworzymy lub próbujemy wykorzystać istniejący już widok elementu listy, a następnie konfigurujemy jego właściwości, aby prezentował wybrane dane z klasy `User`.
- » `GetItemId` – metoda, która zwraca identyfikator obiektu listy. W tym przypadku identyfikator jest ustalany poprzez odczytanie właściwości `Id` wskazanego obiektu `User`. Ten ostatni jest odczytywany za pomocą indeksatora klasy `UsersListAdapter`.
- » `T this[int position]` – indeksator elementów prezentowanych w liście. Indeksator ten działa w ten sposób, iż wywołuje metodę `ElementAt` kolekcji `users`. Ta ostatnia jest używana za pomocą konstruktora klasy `UsersListAdapter`.
- » `Count` – właściwość określająca liczbę elementów listy.

Ponadto klasa `UsersListAdapter` posiada jeszcze dwa pola: `users` oraz `context`. Pierwsze przechowuje kolekcję użytkowników, uzyskaną wcześniej z serwisu REST. Drugie pole, `context`, przechowuje referencję do aktywności, w ramach której tworzona jest instancja klasy `UsersListAdapter`. Pole `context` służy do utworzenia szablonu komórki listy (zob. metoda `GetView` z Listingiem 13).

Ostatnim już składnikiem klasy `UsersListAdapter` jest jej konstruktor. Nie wymaga on jednak dłuższego komentarza, gdyż jego działanie ogranicza się do pobrania obiektów, które są następnie zapisywane w polach `users` i `context`.

Mając gotową klasę adaptera, wykorzystujemy ją do zaimplementowania klasy `MainActivity` (Listing 14). W pierwszym kroku

zmieniamy jej deklarację w taki sposób, aby dziedziczyła po klasie `Android.App.ListActivity`. Dzięki temu tworzona aktywność będzie implementowała widok listy elementów. Następnie, w ramach metody `OnCreate` klasy `MainActivity`, instancjonujemy obiekt `UsersListAdapter`. Działa ona podobnie jak metoda `CreateUsersTableSource` z Listingu 14. Mianowicie najpierw tworzona jest instancja klasy `UsersViewModel`, po czym następuje wywołanie metody `RetrieveUsers` w celu pobrania danych z serwisu REST. Lista użytkowników jest odczytywana z właściwości `Users` i przekazywana, razem z bieżącą instancją klasy `MainActivity`, do konstruktora klasy `UsersListAdapter`. W efekcie nastąpi wyświetlenie listy użytkowników, jak to prezentuje ostatni zrzut ekranu z Rysunku 1.

#### Listing 14. Definicja klasy `MainActivity` z projektu `Users.Droid`

```
[Activity(Label = "Users.Droid",
    MainLauncher = true, Icon = "@mipmap/icon")]
public class MainActivity : ListActivity
{
    protected async override void OnCreate(
        Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);

        ListAdapter = await CreateUsersListAdapter();
    }

    private async Task<UsersListAdapter> CreateUsersListAdapter()
    {
        var usersViewModel = new UsersViewModel();

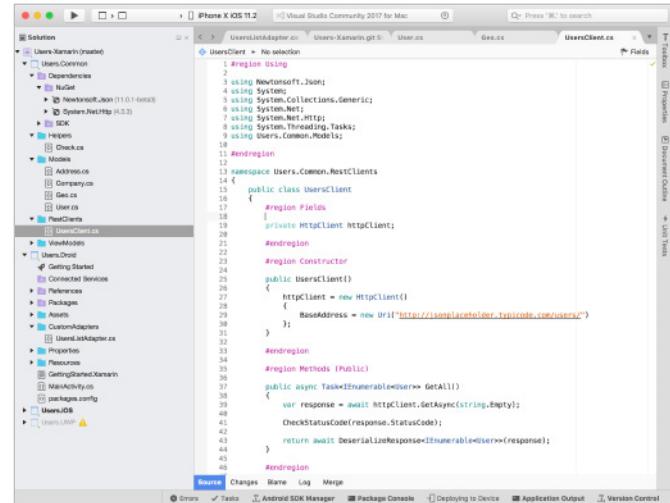
        await usersViewModel.RetrieveUsers();

        return new UsersListAdapter(this, usersViewModel.Users);
    }
}
```

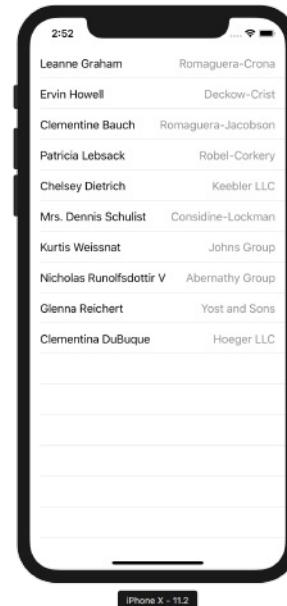
## DYSKUSJA

Zanim przejdziemy do podsumowania tego artykułu, należy podkreślić, że projekt Xamarin-Users można, bez jakichkolwiek modyfikacji, otworzyć w Visual Studio for Mac (Rysunek 12). Bez najmniejszego problemu możemy wówczas uruchamiać aplikacje Android i iOS w emulatorach (zob. prawy zrzut ekranu z Rysunku 1 oraz Rysunek 13) lub na rzeczywistych urządzeniach po wcześniejszej konfiguracji certyfikatów. Jednak tracimy wówczas dostęp do podprojektu `Users.UWP`. Nie może on być otwarty, ponieważ wymaga narzędzi developerskich Windows 10 SDK, których nie można zainstalować w macOS Sierra. Jeśli zależy nam na tworzeniu aplikacji UWP na komputerze Mac, to musimy korzystać z wirtualizacji. Ciekawym rozwiązaniem jest chociażby aplikacja Parallels Desktop, która umożliwia uruchamianie aplikacji Windows w środowisku macOS w taki sposób, że wydają się one działać natywnie. Mając zainstalowany Parallels Desktop, możemy korzystać z Visual Studio dla Windows i komplikować aplikacje iOS poprzez Xamarin Agent. Reasumując, aby tworzyć aplikacje dla trzech platform jednocześnie (UWP, Android i iOS), musimy dysponować maszynami z Windows 10 i macOS. Bardzo wygodnym rozwiązaniem wydaje się być jednak praca na maszynach Windows 10 i komplikacja aplikacji dla platform Apple poprzez Xamarin Agent. Wystarczy wówczas jedna maszyna Mac dla wszystkich członków zespołu developerskiego. Widzimy, że platformy iOS i UWP wzajemnie się wykluczają. Jedynie Android jest wyjątkowy, bo narzędzia developerskie są kompatybilne zarówno z Windows, jak i z macOS. Z kolei iOS i UWP wymaga od programisty odpowiedniego środowiska developerskiego.

W tym miejscu, jak pokazałem w tym artykule, z dużą pomocą przychodzą narzędzia Xamarin. Bardzo dobrze unifikują platformy Windows i macOS za pomocą Xamarin Agent.



Rysunek 12. Projekt `Users-Xamarin` w Visual Studio for Mac



Rysunek 13. Aplikacja `Users.iOS` uruchomiona w symulatorze iPhone X w systemie macOS Sierra

## Porównanie z innymi podejściami

Na koniec porównajmy informacje zebrane w tym artykule z moimi poprzednimi opowieściami o narzędziach do tworzenia aplikacji wieloplatformowych: Embarcadero RAD Studio, Qt Creator, Qt Mobile, Ruby Motion, Intel XDK, Ionic, Electron, React.Native i Xamarin.Forms. Wszystkie te narzędzia można podzielić na trzy grupy (Tabela 1):

- » Aplikacje natywne bez zunifikowanego dostępu do UI – są to aplikacje, które implementujemy z wykorzystaniem jednego środowiska i języka programowania. Część wspólna jest zazwyczaj zawarta w osobnej bibliotece i nie implementuje jakichkolwiek mechanizmów tworzenia UI. Ta biblioteka jest wykorzystywana w projektach dla konkretnych platform, w ramach których musimy samodzielnie zadbać o tworzenie UI dla danej platformy.

- » Aplikacje natywne ze zunifikowanym dostępem do UI – uogólniają poprzedni typ aplikacji w taki sposób, że dostarczają dodatkową zunifikowaną warstwę API. Ta warstwa dostarcza obiekty i metody, które pozwalają tworzyć UI. Obiekty UI mogą być takie same dla wszystkich platform lub być mapowane na natywne kontrolki konkretnej platformy w trakcie komplikacji lub działania aplikacji.
- » Aplikacje hybrydowe – tworzone z wykorzystaniem technologii webowych. Dostarczają spójnego dostępu zarówno do API platformy, jak i UI.

Rodzaj aplikacji	Przykładowe narzędzia	Poziom złożoności	Integracja z platformą
Natywne bez zunifikowanego dostępu do UI	Xamarin Ruby Motion	Wysoki	Duża
Natywne ze zunifikowanym dostępem do UI	Xamarin.Forms Embarcadero RAD Studio Qt Creator React.Native	Średni	Średnia
Hybrydowe	Intel XDK Ionic Electron	Niski	Mała

Tabela 1. Porównanie narzędzi tworzenia aplikacji mobilnych

Jak dokonać optymalnego wyboru pomiędzy powyższymi możliwościami? Odpowiedź na to pytanie zależy głównie od funkcjonalności aplikacji oraz poziomu jej integracji z systemem kontrolującym daną platformę. Dlatego w Tabeli 1 użyłem dwóch kryteriów: poziomu złożoności oraz integracji z platformą.

Mechanizm integracji z platformą łatwo wyjaśnić na przykładzie źródła danych dla tabeli w systemie iOS. Domyslnie wszystkie wbudowane aplikacje iOS, które korzystają z widoku listy, implementują również tak zwaną funkcję *swipe to delete*. Umożliwia ona usuwanie lub modyfikację elementów listy po wykonaniu gestu typu *swipe*. W efekcie pojawiają się dodatkowe przyciski, które pozwalają użytkownikowi wykonać zmiany w danym elemencie listy. W przypadku gdy korzystamy z natywnych narzędzi bez zunifikowanego dostępu do UI, możemy zaimplementować funkcję *swipe to delete* bez najmniejszego problemu. Mianowicie wystarczy uzupełnić definicję o metodę *EditActionsForRow*. Podejście, które

nie korzysta ze zunifikowanego dostępu do UI, wymaga jednak większych nakładów pracy i w ogólnym rozrachunku jest najbardziej skomplikowane, ale daje największe możliwości.

Rozwiązaniem o średnim poziomie złożoności jest tworzenie aplikacji natywnych ze zunifikowanym dostępem do UI. Uzyskujemy wówczas możliwość tworzenia aplikacji wieloplatformowych w oparciu o jedno narzędzie. Przyspiesza to tworzenie aplikacji kosztem ograniczonej integracji z systemem oraz dostępu do specyfiki platformowej.

Najłatwiejszym i zarazem najszybszym sposobem tworzenia wieloplatformowych aplikacji mobilnych jest podejście hybrydowe, w którym aplikacje powstają z wykorzystaniem technologii webowych. To rozwiązanie, podobnie jak w przypadku aplikacji natywnych ze zunifikowanym dostępem do UI, umożliwia nam tworzenie aplikacji dla wszystkich platform jednocześnie z wykorzystaniem dobrze znanych technologii, które dodatkowo są wspierane przez szereg narzędzi i bibliotek. Aplikacje hybrydowe doskonale sprawdzają się w przypadku, gdy nie zależy nam na integracji z systemem, a na uniwersalności i najkrótszym czasie wykonania aplikacji. Przeciętny użytkownik prawdopodobnie nie zauważa różnicy pomiędzy aplikacją hybrydową a natywną.

Czytając blogi programistyczne czy przeglądając różne wystąpienia konferencyjne, można spotkać bardzo wiele głosów krytykujących aplikacje hybrydowe. W praktyce okazuje się jednak, że bardzo wiele firm decyduje się na takie rozwiązanie. Ja sam na co dzień korzystam z kilku takich aplikacji na moim urządzeniu iOS (iPhone 7+). Dobrymi przykładami są aplikacja jednego z największych operatorów telekomunikacyjnych oraz aplikacja jednego z banków. Obie aplikacje sprawdzają się bardzo dobrze. Jedynie aplikacja operatora mobilnego nie umożliwia łatwego skopiowania numeru faktuury, aby móc go następnie wkleić w tytule przelwu podczas płatienia rachunku. Potwierdza to, iż faktycznie aplikacje hybrydowe realizują swoje zadanie, jednak czasem tracimy integrację z systemem.

## PODSUMOWANIE

W ramach tego artykułu pokazałem przykładowe wykorzystanie narzędzi Xamarin do utworzenia trzech aplikacji mobilnych dla najbardziej popularnych platform: UWP, iOS i Android. Podczas tego opisu pokazałem, jaka jest typowa struktura projektu, która umożliwia współdzielenie kodu, omówiłem mechanizmy tworzenia list w poszczególnych platformach i, ostatecznie, porównałem biblioteki Xamarin z innymi, opisanymi wcześniej narzędziami tworzenia aplikacji wieloplatformowych.



### DAWID BORYCKI

Naukowiec, programista, autor wielu książek, amerykańskich patentów, artykułów i videotutoriali o programowaniu w różnych technologiach. Pierwszy polski autor w Microsoft Press i MSDN Magazine. Z wykształcenia doktor fizyki teoretycznej. Obecnie pracuje w Polskiej Akademii Nauk, gdzie zajmuje się rozwojem nowoczesnych, nieinwazyjnych urządzeń do analizowania mikroprzeźłyków krwi w mózgu i obrazowania przez ośrodkie nieprzeczyste optycznie.



Największy wybór profesjonalnego oprogramowania w Polsce !

... w ofercie programy ponad 500 producentów ...



Więcej informacji:

📞 (22) 868 40 42



[sales@tts.com.pl](mailto:sales@tts.com.pl)

Sprzedaż



Dystrybucja



Import na zamówienie

TTS Company Sp. z o.o.

ul. Domaniewska 44A

02-672 Warszawa

[www.tts.com.pl](http://www.tts.com.pl)

# Wykrywanie zmian w systemach plików UNIX/Linux za pomocą skryptów BASH/Korn Shell i sum sprawdzających

Jesteś użytkownikiem Linux lub BSD? Lubisz oprogramowanie o otwartych źródłach? Niepokoją Cię potencjalnie niekontrolowane zmiany w systemie plików lub chcesz wiedzieć o nich więcej? W tym artykule opisano, jak śledzić zmiany w systemie plików pomiędzy wybranymi datami, oraz podano gotowe skrypty wraz z opisem działania i użycia. Międzylektury i niech wszystkie zmiany w Twoim systemie będą kontrolowane!

## PRZYCZAJONE ZŁO

Gdy zetknąłem się z zagadnieniem zabezpieczania systemów UNIX™ [1] lub GNU [2], usłyszałem opinię, że dobry admin usprawnienia bezpieczeństwa w jądrze pisze sam. Słyszałem tę opinię tylko raz. Poznałem wielu adminów, którzy nie pisali, lecz aplikowali istniejące łatwy usprawniające zabezpieczenia [3] na jądro, a całe oprogramowanie na swoich systemach instalowali ze źródeł. Ręcznie na podstawie projektu Linux From Scratch [4] lub automatycznie – w ramach FreeBSD albo Gentoo Linux [5]. Celem tych działań było ustrzeżenie się przez złośliwym kodem, zwłaszcza typu *rootkit* – umożliwiającym zdalny dostęp z uprawnieniami root'a włamywaczom. Kod ten mógłby zostać podstępnie dopisany do paczki binarnej. Ten artykuł traktuje o metodzie sprawdzania, czy pliki wykonywalne w systemie zmieniły się w nieoczekiwany sposób. Skrypty zaprezentowane w artykule powstały na NetBSD [6] z paczkami binarnymi i zostały także przetestowane na Gentoo Linux.

## VANILLA SECURITY

Poważne dystrybucje GNU i UNIX są tak zrobione, że *software* produkcyjny pracujący na nich oraz użytkownicy są bezpieczni. Utrzymanie bezpieczeństwa jest łatwiejsze na codziennie aktualizowanych stacjach roboczych, w sieci z wewnętrznymi adresami Internet Protocol (IP) nie dającymi użytkownikom prawa instalacji programów. Trudniejsze, gdy są to komputery o IP zewnętrznym, zwykle przydzielanym komputerom na krańcu sieci. Ciężar bezpieczeństwa zwykle spoczywa na administratorze. W sytuacji gdy użytkownik sam jest administratorem i użytkownikiem na swoim komputerze, który instaluje znalezione w Internecie paczki,owość bezpieczeństwa zdobyta wraz z systemem przestaje obowiązywać, a użytkownik nadal jest odpowiedzialny za bezpieczeństwo. Paczki dystrybucji, czy to źródłowe czy binarne, są oznaczane przez twórców skrótami sprawdzającymi stan – różnymi rodzajami sum kontrolnych oraz kryptograficznych podpisów, które pozwalały sprawdzić, czy paczka została niezmieniona w drodze pomiędzy serwerem a terminaliem. Zakładając, że dodatkowi użytkownicy komputera, usługi i połączenie sieciowe nie zmieniają oprogramowania, oraz ufając programistom, autorom paczek, serwerom,

nie instalując nieznanego oprogramowania lub paczek mających błędne oznaczenia stanów, oraz apletów webowych, uznajemy, że mamy zaufany *software*. Kiedy instalujemy nową paczkę, stan ten należy rozważyć od początku. Instalator aplikacji często pracuje z uprawnieniami root'a i może zmienić dowolną część systemu operacyjnego. Paczka, która kryje coś nieoczekiwanej, prawdopodobnie wprowadzi do systemu nieoczekiwany element. Zgodność oznaczenia dostarczonego przez producenta i tego wygenerowanego przy sprawdzaniu paczki jest tu kluczową informacją. Jeśli konsekwentnie pójść krok dalej w powyższym podejściu do zabezpieczeń, można uznać, że zaufany system zainstalowany na komputerze cały powinien być oznaczony sumami kontrolnymi, jak również w inny sposób, jeśli to możliwe. Gdy zainstalujemy nowe oprogramowanie, można ponownie oznaczyć system i porównując oba oznaczenia – sprawdzić, co się zmieniło. Sumy kontrolne pozwalają wygenerować ciąg znaków właściwy dla konkretnego pliku. Możliwe jest, że taka suma pasuje do innej zawartości, jednak moc sumy, inny skrót sprawdzający oraz zabezpieczenia opisane w artykule eliminują taką możliwość. Na potrzeby artykułu został użyty algorytm SHA256 [7].

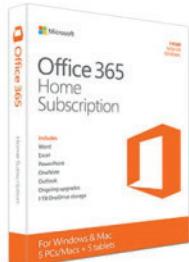
## PRIOR-ART

Oczywiście pierwsze, co należało zrobić, to sprawdzić, czy istnieje narzędzie, które wykona za nas to zadanie. Istnieje przynajmniej jeden gotowy program, którego opis sugeruje, że oznacza on stan plików w systemie i znajduje zmiany – Advanced Intrusion Detection Environment (AIDE) [8]. Jednak przy jego użytkowaniu może okazać się, że nie jest łatwo rozgryźć, jak działa, a modyfikowanie go to zadanie dla profesjonalnego programisty na cały etat, na co tylko niektórzy mogą sobie pozwolić.

## SKRYPTY

Parafrując tezę – dobry admin pisze swoje narzędzia zabezpieczające sam – zwłaszcza jeśli sprawdzają się do skryptu. Artykuł traktuje o skryptach, które oznaczają stan systemu za pomocą skrótów sprawdzających – metod opisania pliku, które z dużym prawdopodobieństwem pasują tylko do jednej zawartości. Skrypty

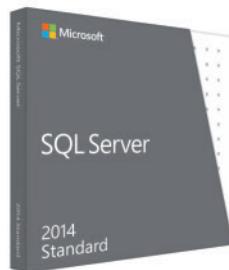
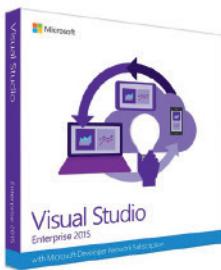
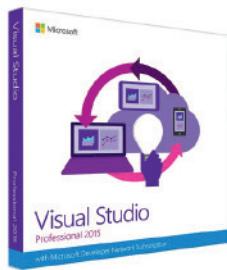
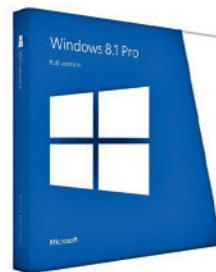
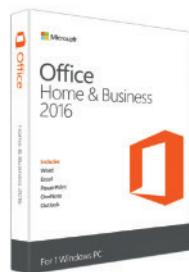
**TTS Company rekomenduje oprogramowanie Microsoft ®**



## Microsoft Partner



Gold Datacenter  
Gold Small and Midmarket Cloud Solutions  
Silver Data Platform  
Silver Data Analytics  
Silver Cloud Productivity



**www.OprogramowanieKomputerowe.pl**

Microsoft Azure

Office 365

OneDrive

Więcej informacji: ☎ (22) 868 40 42 ✉ sales@tts.com.pl

Sprzedaż



Dystrybucja



Import na zamówienie

ty oznaczają system za pomocą sum kontrolnych, a pliki wykonywalne dodatkowo opisują za pomocą linkera [9][10][11]. Skrypty tworzą log, który można porównywać z jego następcą oraz poprzednikiem. Skrypty istnieją w dwóch wariantach: „A” [12], gdzie plik konfiguracyjny zawiera listę katalogów do pominięcia, i „THE” [13], gdzie plik konfiguracyjny zawiera listę plików do przeprocesowania. Skrypty działają w powłokach Korn Shell [14] i Bourne Again Shell [15], używając zewnętrznego polecenia sed [16][17] i były testowane na NetBSD oraz Gentoo Linux. Skrypty są dostępne na GIT pod podanymi linkami na zasadach licencji LGPL wersja 3.

## WYKRYWANIE POTENCJALNEJ INFEKCJI ELF

Format plików ELF [18] umożliwia trzymanie wspólnych funkcji programów we współdzielonych bibliotekach. Programy i biblioteki korzystające z nich linkują się bezpośrednio do wspólnego kodu, dzięki czemu całość zajmuje mniej miejsca na dysku i przez co struktura wykonywalnego kodu przypomina graf wskazujący na różne zbiory instrukcji maszynowych.

### Filesystem

W języku branżowym użytkowników UNIX i GNU system plików to drzewo plików i katalogów, jak i moduł jądra utrzymujący strukturę danych w odpowiednim formacie. Root file system oznacza między innymi drzewo plików i katalogów zamontowane w /, ale może to być też ffs lub ext2, czyli formaty zapisu struktury plików na dysku i moduły jądra do jej obsługi.

Ciekawą formą istnienia złośliwego kodu jest wbudowywanie się w biblioteki działające w formacie ELF. Jeśli złośliwy kod, w tym rootkit, jest napisany sprytnie, może ukryć swoje elementy w różnych bibliotekach i być wywoływany w momencie, gdy bazujący na nich program lub kolejna biblioteka odwoła się do jego adresu. Aby wykryć podłożenie niechcianego kodu, w swojej propozycji skryptów zabezpieczających jako skrót sprawdzający wyborem *printout* z linkera *ld*, który listuje zależności sprawdzanego fragmentu kodu wynikowego. Użytkownik zabezpieczenia, po instalacji nowego oprogramowania, może przyjrzeć się, które zależności były zmieniane, używając komendy *diff* na logach *AllPaths\*<nazwa>\*<data>.log* ze skryptów. Tutaj przyda się wcześniej zdobyta wiedza o bibliotekach i docieklełość użytkownika. Różnice w logach pozwalają dowiedzieć się, które biblioteki się zmieniły. Do użytkownika należy ocena, czy biblioteka została zmieniona w sposób uzasadniony, czy nieoczekiwane. Tylko tyle i aż tyle.

## BACKUP

Dobrym pomysłem podczas zapisywania stanu systemu jest również wykonanie jego kopii zapasowej. Jeśli po modyfikacji oraz porównaniu stanów okaże się, że zaszły podejrzane zmiany, system można przywrócić. Samo podejrzenie, przy braku prostej metody powrotu, może powodować błędne decyzje – np. bagatelowizowanie zagrożenia w obliczu pojawiającej się konieczności reinstalacji systemu przez administratora lub nadmiarowe re-instalowanie systemu operacyjnego. Mówią tutaj zwłaszcza o komputerach używanych nieformalnie lub nie podlegających sztywnym procedurom, nie posiadających funkcjonalnych *mirrorów*. Mechanizmy backupu przedstawione tu mogą im się szczególnie przydać, jednak będą wymagały procedury. Skrypty te służą dobrze, gdy używając ich, będzie się wykonywało backup

przed, oraz oznaczenie przed i po modyfikacji systemu, a także po podejrzeniu niekontrolowanej modyfikacji. Kopię zapasową można wykonać poprzez obraz partycji, jak i używając skryptów opisanych w tym artykule. Skrypty do kopii zapasowej i oznaczania stanu mogą używać tego samego pliku konfiguracyjnego.

### Drzewo katalogów dla skryptów i kopii zapasowych

Drzewo katalogów związane z systemem oznaczania zostało utworzone na oddzialejnej partycji podmontowanej do katalogu /backup.

/backup – miejsce przechowywania kopii zapasowych i podkatalogów.

/backup/scriptnconf – miejsce przechowywania skryptów i konfiguracji w wariantce „A”, a także logów w oddzielnich katalogach o nazwie zależnej od czasu i identyfikatora nadanego logom. Logi w katalogu to:

- ▶ *backuploopa.sh*, *cutslash.sh*, *mkdirsafterbackupa.sh*, *searchstring.sh*, *sumfsa.sh*, *sumfsaconfig.cfg*

/backup/scriptnconfthe – miejsce przechowywania skryptów i konfiguracji w wariantce „THE”, a także logów w oddzielnich katalogach o nazwie zależnej od czasu i identyfikatora nadanego logom. Logi w katalogu to:

- ▶ *backuploopthe.sh*, *cutslash.sh*, *dirssumlist.cfg*, *mkdirsafterbackupthe.sh*, *sumthefs.sh*

Katalogi ze skryptami zawierają także podkatalogi z logami stanu systemu.

Główny log to *AllPaths\**. Dodatkowo *err\** to log z błędami wykonania.

## UŻYTKOWNIK I SHELL

Skrypty muszą móc dotrzeć do każdego miejsca w systemie. Z tego też powodu najlepiej działają uruchamiane przez użytkownika z uprawnieniami do czytania wszystkich plików. Jednakże służą one do sprawdzania stanu systemu po zainstalowaniu potencjalnie niebezpiecznego oprogramowania. Takie oprogramowanie mogło zmieścić dowolny program w systemie. Co, jeśli zmieniło właśnie shella?

### Kilka sposobów, aby uniknąć uruchomienia szkodliwego oprogramowania podczas sprawdzania systemu

1. Wykonanie obrazu partycji i sprawdzanie jego kopii na innym komputerze.
2. Za każdym razem po instalacji uruchomić system z płyty w trybie ratunkowym (ang. *rescue mode*), podmontować system plików do systemu plików załadowanego z płyty i oznaczać system, zapisując logi na pendrive.
3. Udostępnienie całego „/” poprzez NFS i oznaczanie go z innego komputera w sieci, zapisując logi na tamtym komputerze.
4. Wyjęcie twardego dysku i także oznaczanie na innym komputerze.

Są to metody informatyki śledczej [19] pracujące na danych z nośnika, nie z uruchomionego komputera. Ich zaletą jest nieangażowanie oprogramowania oznaczanego w sam proces. Przy ich zastosowaniu czas pozostania bądź powrotu komputera do pracy nie jest priorytetem.

Sposób oznaczenia systemu z użyciem jego oprogramowania minimalizujący ryzyko uruchomienia złośliwego kodu to instalacja ze źródeł programów, których używają skrypty, i ukrycie ich w jakimś nie rzucającym się w oczy katalogu, bez linkowania ELF. Na przykład /usr/bin/awkc. Program powłoki, sed'a i tar'a trzeba skompiliować osobiste ze źródeł, dostarczając programy stanowiące zależności i skupiając się, by nie linkowały się do bibliotek w formacie ELF. Informację, jak to zrobić, zawierają pliki README i INSTALL do-

starczone razem ze źródłami każdego z programów. Nazwa użytkownika skryptów powinna być tak wymyślona, aby nie zwracała na siebie uwagi. Na przykład „wojciec” z katalogiem domowym /home/wojciec. To, czy domyślna powłoka użytkownika powinna wskazywać miejsce standardowe – /bin/sh czy inne – na przykład /var/bin/awkc, zależy od przewidywanego skupienia użytkownika przy przystępowaniu do oznaczania systemu. Przekierowanie standardowej powłoki użytkownika do powłoki specjalnej – shella użytkownika wpisany jako /var/bin/awkc, minimalizuje szansę po-mylenia powłoki, ale zdradza jej obecność. Za to używanie shella ukrytego – brak wpisu /var/bin/awkc jako shella użytkownika – zwiększa szansę przypadkowego uruchomienia zwykłej powłoki i potencjalnie złośliwego kodu. Instalacja składa się z pobrania źródeł samej powłoki, programów tar i sed i wszystkich zależności, sprawdzenia sumy kontrolnej lub podpisu kryptograficznego paczek, rozpakowania archiwów, przeczytania, wybrania i wykonania poleceń z plików README oraz INSTALL z uwzględnieniem swoich wymagań – czyli nieużywania technologii ELF w danej komplikacji (–static). Komplikowanie należy przeprowadzić dla każdej zależności, rozpoczynając od „dołu” grafu, czyli od paczek, które zależności nie mają, a kończąc na właściwych programach i powłocie.

## DWA PODEJŚCIA

Skrypty istnieją w dwóch wersjach. Jedna pozwala na wykonanie kopii zapasowej oraz skrótu sprawdzającego stan wszystkich katalogów oprócz wypisanych w pliku konfiguracyjnym, druga przeciwnie – na podstawie listy katalogów do oznaczenia.

Podejście pierwsze jest dla niektórych bardziej wygodne, bo nie trzeba przepisywać listy katalogów w /, a jedynie zrobić listę katalogów do pominięcia, która będzie przenoszona na inne systemy plików w innych systemach operacyjnych. Skrypt ten jednak działa wolniej.

## PODSTAWOWE FUNKCJE

Oba skrypty korzystają z funkcji cutslash umieszczonej w dedykowanym skrypcie cutslash.sh. Za pomocą narzędzia sed i komendy y – transformuj – funkcja zmienia znaki „/” na “\_”. Robi to po to, aby można było użyć daty w nazwie plików z logami stanu systemu i kopiami zapasowymi. Funkcja ta uporczywie szuka wzorca w tekście – nie przestanie, póki nie zamieni wszystkich jego wystąpień. Standardowo w dacie „/” występuje trzy razy – Listing 1.

**Listing 1. Skrypt custlash.sh**

```
#!/bin/ksh
echo $1 | sed y=/=_=
```

Pierwszy parametr skryptu w potoku przedostaje się pod władzę programu sed, który komendą y zmienia znak slash na underscore. Znakiem separującym dla sed'a jest znak równości, który pomyślnie zastępuje backslash w tej funkcji, co jest ważne w tym kontekście użycia sed'a.

Drugą podstawową funkcją jest searchstring w skrypcie searchstring.sh – Listing 2.

**Listing 2. Skrypt searchstring.sh**

```
#!/bin/ksh
# First parameter $1 - string in which to search
# Second parameter $2 - string to search

set -e
```

```
CUTMATCHED1=`echo $1 |sed s=$2.*=$2=g`
CUTMATCHED2=`echo ${CUTMATCHED1} |sed s=$2==g`
```

Pierwszy parametr funkcji poprzez potok zostaje oddany pod władzę sed, który zastępuje cały ciąg znaków zaczynający się od drugiego parametru – poszukiwanego ciągu, tylko szukanym ciągiem. Wynik jest zapamiętywany w zmiennej CUTMATCHED1, która potokiem przechodzi do kolejnej substytucji. Jeśli jest w niej tylko poszukiwany ciąg znaków, zostanie on wymazany, a wartość substytucji zostaje zapisana do zmiennej CUTMATCHED2. Na wyjściu znajduje się w niej dowolny ciąg znaków lub brak wartości. W bieżącym if [20] brak wartości jest oznaczany jako -z.

## PĘTLA ELIMINACJI

**Listing 3. Pętla eliminacji w wariantce „A”**

```
# Second parameter $2 - list of dirs NOT to be backed up
for i in `ls -1 /` do
  i="/$i
  MATCHWRONG=0
  for j in `cat $2` do
    TOMATCH= `./searchstring.sh $i $j` 
    if [ -z $TOMATCH ]; then
      MATCHWRONG=1
      break
    fi
  done
  if [ $MATCHWRONG -eq 0 ]; then
    DIR="/$i
    echo $DIR
    tar -rvf $FILENAME $DIR
  else
    echo $i
    echo "Out of scope"
  fi
done
```

W Listingu 3 pokazano podstawową pętlę skryptu wykonującą kopię zapasową umieszczoną także w skrypcie oznaczającym stan. Wczytuje ona do zmiennej \$i kolejne katalogi i pliki wylistowane z katalogu głównego /, dla każdego katalogu i pliku sprawdza, czy znalazły się on na liście eliminacji w pliku konfiguracyjnym poprzez wczytanie wierszy pliku do zmiennej \$j i sprawdzenie, czy \$j występuje w \$i za pomocą funkcji ./searchstring.sh \$i \$j (.sh – funkcja jest w oddzielnym skrypcie). \$MATCHWRONG oznacza, czy znaleziono choć jedno dopasowanie. \$TOMATCH bez wartości (if [ -z \$TOMATCH]) oznacza znalezienie dopasowania i oznaczenie \$MATCHWRONG. \$MATCHWRONG jest zerowane dla każdego katalogu i pliku z /. Jeśli otrzymało się w tej zmiennej wartość zero, na końcu pętli jest wykonywany kawałek backupu – Listing 4. Dla skryptu oznaczającego stan systemu, w tych okolicznościach, jest wywoływane oznaczenie stanu dla danego węzła, co widać w Listingu 5.

**Listing 4. Dodanie kopii zapasowej zawartości ścieżki**

```
if [ $MATCHWRONG -eq 0 ]; then
  DIR="/$i
  echo $DIR
  tar -rvf $FILENAME $DIR
else
```

**Listing 5. Wybranie pliku do oznaczenia**

```
if [ $MATCHWRONG -eq 0 ]; then
  if [ -d $i ]; then
    echo "Dir"
```

```

# search $LOGFILENAME $i
else
  checkthefile $LOGFILENAME $i
fi
else

```

Jeśli MATCHWRONG było większe niż zero, na wyjściu standardowym wyświetlany jest komunikat „Out of scope” – Listing 6.

#### **Listing 6. Komunikat o elemencie wyeliminowanym – fragment pętli eliminacji**

```

else
  echo $i
  echo "Out of scope"
fi

```

Użycie funkcji cutslash znajduje się na początku wykonania skryptów wykonujących kopię zapasową oraz oznaczanie stanu – Listing 7.

#### **Listing 7. Konwersja daty – funkcja cutslash.sh**

```

RUNDATE=`date +%x`
RUNDATE=`./cutslash.sh $RUNDATE`
```

Konwersja daty w Listingu 7 polega na skopiowaniu daty do zmiennej \$RUNDATE oraz wpisaniu do tej samej zmiennej wyjścia z funkcji cutslash, której parametrem była pierwotna wartość \$RUNDATE.

#### **Listing 8. Inicjacja kopii zapasowej**

```

FILENAME="/backup/backup_\"$RUNDATE\"_$1\".tar"
DIR=""
tar -cvf $FILENAME /backup/scriptnconf*
```

Skrypt kopii zapasowej tworzy archiwum – Listing 8 – a potem daje do niego niewyeliminowane katalogi i pliki przy użyciu komendy:

```
tar -rvf $FILENAME $DIR
```

Skrypt tworzy na końcu skompresowaną wersję archiwum komendą:

```
gzip $FILENAME
```

## **PIERWSZY ZE SKRYPTÓW**

W Listingu 9 przedstawiono cały skrypt kopii zapasowej, z eliminacją ścieżek zaczynających się od wierszy z pliku konfiguracyjnego.

#### **Listing 9. Skrypt backuploopa.sh**

```

#!/bin/ksh
# First parameter $1 - archive name
# Second parameter $2 - list of dirs NOT to be backed up

set -e

RUNDATE=`date +%x`
RUNDATE=`./cutslash.sh $RUNDATE`
echo $RUNDATE
FILENAME="/backup/backup_\"$RUNDATE\"_$1\".tar"
DIR=""
tar -cvf $FILENAME /backup/scriptnconf*
for i in `ls -1 /` 
do
  i="/"$i
  MATCHWRONG=0
```

```

for j in `cat $2` 
do
  TOMATCH=`./searchstring.sh $i $j`
  if [ -z $TOMATCH ]; then
    MATCHWRONG=1
    break
  fi
done
if [ $MATCHWRONG -eq 0 ]; then
  DIR="$i"
  echo $DIR
  tar -rvf $FILENAME $DIR
else
  echo $i
  echo "Out of scope"
fi
done
echo ""
gzip $FILENAME
```

W pierwszym wierszu skryptu `#!/bin/ksh` oznacza ścieżkę do powłoki, która ma wykonać skrypt. W systemach Linux zalecane jest użycie `#!/bin/bash`. A jeśli użytkownik zainstalował wybraną powłokę sam, tutaj jest miejsce, by wpisać ścieżkę do niej. Skrypt może przyjmować parametry w formie wyrazów po jego nazwie w linii komend, np.

```
./backuploopa.sh FirstBackup dirslist.cfg
```

## **Użyte funkcjonalności KSH i BASH**

Kolejne wyrazy to parametry \$1, \$2 – przyjmują one wartości wyrazów wpisanych po nazwie funkcji lub skrypty przy ich uruchamianiu. Tutaj są to – nazwa logu oraz plik konfiguracyjny. W niektórych innych funkcjach jest jeszcze parametr \$3 pozwalający zdecydować, czy ma być użyty linker. Komenda echo wypisuje komunikaty na wyjście standardowe i pozwala nam na śledzenie stanu wykonania. Zmienna jest wymieniana w skrypcie z \$, kiedy chcemy odczytać z niej wartość. Nazwa funkcji bez znaku \$ jest używana w przypisaniu wartości. Ilość spacji pomiędzy zmienną a wartością w wyrażeniu A=a i ma znaczenie, podobnie w `if [-z ]`; Znaczenie ma też obecność i miejsce umieszczenia znaku ;. Blok kodu w ksh/bash oznaczamy znakami ` . Instrukcja set -e powoduje, że linijka skryptu może być uruchomiona dopiero po poprawnym zakończeniu poprzedniej. Instrukcja jest użyta dlatego, że nieprawidłowe wywołanie niektórych funkcji pozwalałoby na dalsze wykonanie skryptu, jednak działającego niepoprawnie, czego wynikiem mogłoby być nadpisanie starszych logów lub gromadzenie niepełnych albo niepoprawnych informacji.

## **OZNACZANIE STANU**

Skrypty oznaczania stanu zaczynają się od wybrania powłoki, pobrania parametrów, ustwienia zmiennych globalnych, sprawdzenia istnienia linkera oraz definicji funkcji. Po definicjach funkcji następuje transformacja daty oraz operacje na niektórych zmiennych i przejście do pętli eliminacji.

Parametr pierwszy to ścieżka do logu, który ma powstać. Skrypt zgłosi błąd i przerwie uruchamianie, jeśli log już istnieje. Parametr drugi to nazwa pliku konfiguracyjnego z listą katalogów i plików do pominięcia.

Początek skryptu oznaczania stanu z eliminacją – Listing 10.

#### **Listing 10. Sprawdzanie obecności linkera**

```

#!/bin/ksh
# SUMTHEFS
# This script maps the filesystem tree from / node
# and writes down cryptographic checksums for files
# and also linker info for executables
#
# First parameter $1 - sum name
# Second parameter $2 - config file
```

```

set -e
RUNDATE=`date +%x`
SUPERSTART=`pwd`
CONFIG=$SUPERSTART"/"$2

if [ -x /usr/bin/ld ]; then
    LD=/usr/bin/ld
    $LD -V
else
    LD=""
fi

```

Nowymi częściami w stosunku do poprzednich elementów są zapamiętanie ścieżki, z której skrypt został uruchomiony, oraz sprawdzenie, czy linker jest zainstalowany w systemie.

Zmienna \$SUPERSTART przyjmuje ścieżkę uruchomienia. Na jej podstawie, oraz na podstawie parametru drugiego, który przyjął nazwę pliku, zmienna \$CONFIG przyjmuje ścieżkę do pliku konfiguracyjnego.

if [ -x /usr/bin/ld ]; sprawdza, czy linker jest w systemie. Jeśli jest, będzie on użyty do oznaczania stanu. Można zmienić tę ścieżkę, jeśli w naszym systemie jest ona inna. Przypisanie LD="" oznacza brak linkera, co później jest sprawdzane przy oznaczaniu stanu.

## OZNACZANIE LINKEREM – FUNKCJA

**Listing 11. Funkcja ldfile. Oznaczanie linkerem**

```

function ldfile {
# First parameter $1 - log path
# Second parameter $2 - file path
# Third parameter $3 - linker
echo "$3 -M $2" >> $1
$3 -M $2 -o /dev/void >> $1
}

```

Funkcja **ldfile** – Listing 11 – loguje komendę przekierowaniem strumienia na koniec logu >> \$1 do parametru \$1, po czym wykonyuje polecenie ld -M /foo/bar – wypisanie mapy linków [21]. Linker jest wywoływany ze ścieżki z parametrem \$3. Wyjście czytelne dla człowieka jest logowane, a wyjście binarne kierujemy do /dev/null jako element niepotrzebny.

## OZNACZANIE SUMAMI KONTROLNYMI – FUNKCJA

**Listing 12. Funkcja sumthefile. Oznaczanie sumą kontrolną i warunkowy skok do funkcji ldfile**

```

function sumthefile {
# First parameter $1 - log path
# Second parameter $2 - file path
# Third parameter $3 - linker

echo $2
if [ -x /usr/bin/sha256sum ]; then
    /usr/bin/sha256sum $2 >> $1
elif [ -x /usr/bin/cksum ]; then
    /usr/bin/cksum -a sha256 $2 >> $1
fi
#sha224sum $2 >> $1
#sha256sum $2 >> $1
#sha384sum $2 >> $1
#sha512sum $2 >> $1

if [ -z $3 ]; then
    echo "Not checking linker"
elif [ -x $LD ]; then
    echo "Linker Info"
    ldfile $1 $2 $3
else
    echo "No linker checks considered"
fi
}

```

Funkcja **sumthefile** – Listing 12 – sprawdza istnienie ścieżki /usr/bin/sha256sum jako pliku wykonywalnego. Istnienie takiej wskazuje na system GNU. Jeśli istnieje, suma kontrolna pliku z parametru \$2 zostaje zalogowana do logu z parametru \$1. W systemach UNIX możemy się spodziewać braku sha256sum (taką nazwę niektórzy użytkownicy Internetu nazywają *gnuzizmem*) i musimy wskazać plik wykonywalny cksum. Skrypt uruchamia go z katalogu /usr/bin. Przed uruchomieniem sprawdza, czy plik wykonywalny jest w tym miejscu. Jeśli wynik jest pozytywny, uruchamia linker z algorytmem sha256 – /usr/bin/cksum -a sha256, a wyjście przekierowuje na koniec logu. Następnie sprawdzane jest, czy funkcja została wykonana z wartością parametru trzeciego nakazującą użycie linkera, bądź nie. Jeśli parametr trzeci wskazuje ścieżkę linkera, zostanie on użyty poprzez wykonanie funkcji **ldfile**, jeśli nie, informacja zostanie wyświetcona na wyjściu standardowym.

## ROZPOZNAWANIE RODZAJÓW WĘZŁÓW SYSTEMU PLIKÓW – FUNKCJA

Przed oznaczeniem sumą kontrolną, oraz linkerem, należy sprawdzić, z jakiego rodzaju ścieżką mamy do czynienia. Funkcja **checkthefile** rozpoznaje rodzaje węzłów.

### KSH/BASH if

- ▶ if [ -x <ścieżka> ]; oznacza poszukiwanie pliku wykonywalnego jako celu ścieżki,
- ▶ if [ -f <ścieżka> ]; oznacza poszukiwanie pliku,
- ▶ if [ -e <ścieżka> ]; oznacza sprawdzenie, czy ścieżka prowadzi do węzła, który istnieje,
- ▶ if [ -d <ścieżka> ]; oznacza sprawdzenie, czy ścieżka wskazuje katalog,
- ▶ if [ -h <ścieżka> ]; oznacza sprawdzenie, czy ścieżka prowadzi do linku symbolicznego.

**Listing 13. Funkcja checkthefile. Badanie rodzaju węzła drzewa katalogów**

```

function checkthefile {
# First parameter $1 - path to log
# Second parameter $2 - path to file
LOGFILENAME=$1
ANALYZED=$2
if [ -x $ANALYZED ]; then
    echo $ANALYZED
    sumthefile $LOGFILENAME $ANALYZED $LD
elif [ -f $ANALYZED ]; then
    echo $ANALYZED
    sumthefile $LOGFILENAME $ANALYZED
elif [ -e $ANALYZED ]; then
    echo $ANALYZED
    sumthefile $LOGFILENAME $ANALYZED
else
    echo $ANALYZED
    echo "Target does not exists."
fi
}

```

Funkcja **checkthefile** – Listing 13 – przyjmuje dwa parametry – ścieżkę do logu oraz ścieżkę do oznaczenia. Szuka w kolejności od pliku wykonywalnego, poprzez dowolny plik, dowolny węzeł oraz cel, który nie pasuje do żadnego z powyższych. Oznaczanie pliku wykonywalnego przeprowadzane jest za pomocą sumy kontrolnej oraz linkera. Do pliku zwykłego wykorzystana jest tylko suma kontrolna, tak samo jak do linku symbolicznego. Brak dopasowania jest komunikowany na wyjściu standardowym.

## PRZYGOTOWANIE DO URUCHOMIENIA PETLI ELIMINACJI – GŁÓWNA CZĘŚĆ SKRYPTU

**Listing 14.** Pętla główna skryptu sumfsa.sh

```
RUNDATE=`./cutslash.sh $RUNDATE`
LOGDIR= `pwd` "/summedfs_"$RUNDATE$1
LOGFILENAME=$LOGDIR"/AllPaths$RUNDATE.log"
MATCHWRONG=0
mkdir $LOGDIR
touch $LOGDIR/error$RUNDATE_"$1.log
time {
    echo "Log name: $1" >> $LOGFILENAME
    echo "Config text:" >> $LOGFILENAME
    cat $CONFIG >> $LOGFILENAME
    echo "Tree:" >> $LOGFILENAME
    find / >> $LOGFILENAME
    echo "Cryptographic checksums and linker info:" >>
$LOGFILENAME
    time {
        for i in `find /` do
            < wnetrze petli >
        done
        echo "Sumfs Finished"
    } 2>$LOGDIR/error$RUNDATE_"$1.log
}
```

Skrypt ponownie sprawdza, gdzie został uruchomiony, tym razem w celu nadania nazwy, utworzenia katalogu i uformowania ścieżki głównego logu [...]summedfs"\$RUNDATE\$1 oraz AllPaths\$RUNDATE.log. Polecenie mkdir tworzy katalog. Komenda touch przygotowuje plik logujący błędy napotkane podczas wykonania wynikające z braku możliwości wykonania niektórych poleceń. Jeśli podejrzewamy, że brakuje w logu jakiegoś wpisu, można sprawdzić, czy w pliku z błędami są wzmianki o ścieżce, której ten wpis miał dotyczyć. Wpisy w tym logu nie oznaczają wprost, że główny log jest niepoprawny. Zachęcam do zaglądania do tego pliku i dostosowywania skryptów do swoich potrzeb.

Skrypt uruchamia funkcję mierzenia czasu i zaczyna logowanie od nazwy i daty logu oraz nazwy i zawartości pliku konfiguracyjnego cat \$CONFIG >> \$LOGFILENAME. Później całe drzewo systemu plików komputera jest także wpisywane do logu. Drzewo stanowi najprostszą formę oznaczenia stanu systemu find / >> \$LOGFILENAME. W logu pojawia się jeszcze nagłówek o zapisywaniu sum kontrolnych i logów z linkera, a potem uruchamiana jest pętla eliminacji, której blok zamyka się słowem kluczowym done. Skrypt informuje użytkownika o końcu wykonania zdaniem „Sumfs Finished” wysłanym na wyjście standardowe. Po tym zdaniu pojawi się także czas wykonania. Warto tutaj zaznaczyć, że skrypt działający metodą eliminacji wykonuje się dłużej niż skrypt działający na podstawie wskazania katalogów do oznaczenia. Dzieje się tak przez porównywanie każdej gałęzi ścieżki ze ścieżką eliminacji, gdzie za każdym razem potrzebne jest wczytanie pliku konfiguracyjnego. Druga metoda, omówiona później, działa szybciej, mimo zastosowania w niej rekursji. Cały skrypt oznaczania stanu konfigurowany metodą eliminacji znajduje się w Listingu 15.

**Listing 15** Skrypt sumfsa.sh

```
#!/bin/ksh
# SUMTHEFS
# This scripts maps the filesystem tree from / node
# and writes down cryptographic checksums for files
# and also linker info for executables
#
# First parameter $1 - sum name
# Second parameter $2 - config file

set -e
RUNDATE=`date +%x`
SUPERSTART=`pwd`
CONFIG=$SUPERSTART"/"$2

if [ -x /usr/bin/ld ]; then
    LD=/usr/bin/ld
    $LD -V
else
    LD=""
fi

function ldfile {
# First parameter $1 - log path
# Second parameter $2 - file path
# Third parameter $3 - linker
echo "$3 -M $2" >> $1
$3 -M $2 -o /dev/void >> $1
}

function sumthefile {
# First parameter $1 - log path
# Second parameter $2 - file path
# Third parameter $3 - linker
echo $2
if [ -x /usr/bin/sha256sum ]; then
    /usr/bin/sha256sum $2 >> $1
elif [ -x /usr/bin/cksum ]; then
    /usr/bin/cksum -a sha256 $2 >> $1
fi
#sha224sum $2 >> $1
#sha256sum $2 >> $1
#sha384sum $2 >> $1
#sha512sum $2 >> $1

if [ -z $3 ]; then
    echo "Not checking linker"
elif [ -x $LD ]; then
    echo "Linker Info"
    ldfile $1 $2 $3
else
    echo "No linker checks considered"
fi
}

function checkthefile {
# First parameter $1 - path to log
# Second parameter $2 - path to file
LOGFILENAME=$1
ANALYZED=$2
if [ -x $ANALYZED ]; then
    echo $ANALYZED
    sumthefile $LOGFILENAME $ANALYZED $LD
elif [ -f $ANALYZED ]; then
    echo $ANALYZED
    sumthefile $LOGFILENAME $ANALYZED
elif [ -e $ANALYZED ]; then
    echo $ANALYZED
    sumthefile $LOGFILENAME $ANALYZED
else
    echo $ANALYZED
    echo "Target does not exists."
fi
}

RUNDATE=`./cutslash.sh $RUNDATE`
LOGDIR= `pwd` "/summedfs_"$RUNDATE$1
LOGFILENAME=$LOGDIR"/AllPaths$RUNDATE.log"
MATCHWRONG=0
mkdir $LOGDIR
touch $LOGDIR/error$RUNDATE_"$1.log
time {
    echo "Log name: $1" >> $LOGFILENAME
    echo "Config text:" >> $LOGFILENAME
    cat $CONFIG >> $LOGFILENAME
    echo "Tree:" >> $LOGFILENAME
    find / >> $LOGFILENAME
    echo "Cryptographic checksums and linker info:" >>
$LOGFILENAME
    time {
        for i in `find /` do
            MATCHWRONG=0
            for j in `cat $CONFIG` do
                do
```

```

TOMATCH=`$SUPERSTART/searchstring.sh $i $j`
if [ -z $TOMATCH ]; then
    MATCHWRONG=1
    break
fi
done
if [ $MATCHWRONG -eq 0 ]; then
    if [ -d $i ]; then
        echo "Dir"
        # search $LOGFILENAME $i
    else
        checkthefile $LOGFILENAME $i
    fi
else
    echo $i
    echo "Out of scope"
    fi
done
echo "Sumfs Finished"
} 2>${LOGDIR}/error${RUNDAT}e_"${1}.log"
}

```

**Listing 16.** Plik konfiguracyjny dla wariantu „A” dla NetBSD

```

/dev
/cdrom
/kern
/mnt
/pendrive
/proc
/root

```

## PĘTŁA WYBORU KATALOGÓW – GŁÓWNA CZĘŚĆ SKRYPTU KONFIGUROWANEGO POPRZECZ WSKAZANIE KATALOGÓW

**Listing 17.** Pętla główna skryptu sumthefs.sh

```

for i in `cat $CONFIG`
do
    search $LOGFILENAME $i
    echo `pwd`$i
done

```

Pętla główna z Listingu 17 listuje wszystkie katalogi i pliki z pliku konfiguracyjnego i w pętli, na każdym wykonuje funkcję search.

**Listing 18.** Funkcja search

```

if [ -h $start ]; then
    echo "Symlink found."
else
for i in `ls $start`; do
    ANALYZED=$start"/"${i}
    echo $ANALYZED >> $LOGFILENAME
    echo $ANALYZED
    if [ -d $ANALYZED ]; then
        echo $ANALYZED
        search $LOGFILENAME $ANALYZED $LD
    elif [ -x $ANALYZED ]; then
        sumthefile $LOGFILENAME $ANALYZED $LD
    elif [ -f $ANALYZED ]; then
        sumthefile $LOGFILENAME $ANALYZED
    else
        echo $ANALYZED
        echo "Node does not match to anything interesting in this
version"
    fi
done
fi
cd $start
}

```

Funkcja search – Listing 18 – listuje zawartość katalogu, dla którego została uruchomiona, i dla każdego wiersza sprawdza, czy jest to katalog, plik binarny wykonalny czy zwykły plik. Gdy na-

potka katalog, rekurencyjnie uruchamiana jest nowa instancja funkcji search. Dla pliku wykonalnego uruchamiana jest funkcja sumthefile z włączoną opcją linkera, czyli zostanie zapisany log z linkera, dla obu rodzajów plików zostanie także wygenerowana suma kontrolna.

**Listing 19.** Skrypt sumthefs.sh

```

#!/bin/bash
# SUMTHEFS
# This script maps the filesystem tree from / node
# and writes down cryptographic checksums for files
# and also linker info for executables
#
# First parameter $1 - sum name
# Second parameter $2 - config file
set -e

RUNDAT=`date +%x`
SUPERSTART=`pwd`
CONFIG=$SUPERSTART"/"$2

if [ -x /usr/bin/ld ]; then
    LD=/usr/bin/ld
    $LD -V
else
    LD=""
fi

function ldfile {
# First parameter $1 - log path
# Second parameter $2 - file path
# Third parameter $3 - linker
echo "$3 -M $2" >> $1
$3 -M $2 -o /dev/void >> $1
}

function sumthefile {
# First parameter $1 - log path
# Second parameter $2 - file path
# Third parameter $3 - linker
echo $2
if [ -x /bin/sha256sum ]; then
    /bin/sha256sum $2 >> $1
elif [ -x /usr/bin/cksum ]; then
    /usr/bin/cksum -a sha256 $2 >> $1
fi
#sha224sum $2 >> $1
#sha256sum $2 >> $1
#sha384sum $2 >> $1
#sha512sum $2 >> $1

if [ -z $3 ]; then
    echo "Not checking linker"
elif [ -x $LD ]; then
    echo "Linker Info"
    ldfile $1 $2 $3
else
    echo "No linker checks considered"
fi

function search {
# First parameter $1 - path to log
# Second parameter $2 - path to dir
LOGFILENAME=$1
start=$2

if [ -h $start ]; then
    echo "Symlink found."
else
for i in `ls $start`; do
    ANALYZED=$start"/"${i}
    echo $ANALYZED >> $LOGFILENAME
    echo $ANALYZED
    if [ -d $ANALYZED ]; then
        search $LOGFILENAME $ANALYZED
    elif [ -x $ANALYZED ]; then
        sumthefile $LOGFILENAME $ANALYZED $LD
    elif [ -f $ANALYZED ]; then
        sumthefile $LOGFILENAME $ANALYZED
    else
        echo $ANALYZED
        echo "Node does not match to anything interesting in this
version"
    fi
done
fi
}

```

```

fi
done
fi
cd $start
}
RUNDATE=`./cutslash.sh $RUNDATE`
LOGDIR=`pwd`"/summedfs_"$RUNDATE$1
LOGFILENAME=$LOGDIR"/AllPaths$RUNDATE.log"
mkdir $LOGDIR
touch $LOGDIR/error$RUNDATE_"$1.log
time {
echo "Log name: $1" >> $LOGFILENAME
echo "Config text:" >> $LOGFILENAME
cat $CONFIG >> $LOGFILENAME
echo "Tree:" >> $LOGFILENAME
find / >> $LOGFILENAME
echo "Cryptographic checksums and linker info:" >> $LOGFILENAME
for i in `cat $CONFIG`
do
    search $LOGFILENAME $i
    echo `pwd`$i
done
echo "Sumfs Finished"
} 2>$LOGDIR/error$RUNDATE_"$1.log

```

**Listing 20. Plik konfiguracyjny wariantu „THE” dla NetBSD**

```

._cshrc
./profile
/altroot
/bin
/boot
/boot.cfg
/dev
/etc
/kern
/lib
/libdata
/libexec
/netbsd
/rescue
/sbin
/stand
/tmp
/usr
/var

```

**Listing 21. Skrypt backuploopthe.sh**

```

#!/bin/bash
# First parameter $1 - archive name
# Second parameter $2 - list of dirs to be backed up

set -e

RUNDATE=`date +%x`
RUNDATE=`./cutslash.sh $RUNDATE`
echo $RUNDATE
FILENAME="/backup/backup_"$RUNDATE_"$1".tar"
DIR=""
tar -cvf $FILENAME /backup/scriptnconf
touch $FILENAME
for i in `cat $2`
do
    DIR="/$i
    echo $DIR
    tar -rvf $FILENAME $DIR
done
echo ""
gzip $FILENAME

```

## PRZYWRACANIE BACKUPU

Jeśli katalog /backup wskazuje na inną partycję, w celu przywrócenia go trzeba uruchomić bliźniaczy system operacyjny. Może on rezydować na partycji z backupem, można też go uruchomić na przykład po zbootowaniu komputera z płyty.

Aby przywrócić backup, trzeba zamontować partycję root głównego systemu plików oraz zamontować partycję z kopią zapasową. Następnie zabezpieczyć dane użytkowników i root'a, skaso-

wać zawartość partycji, skierować powłokę do katalogu z główną partycją i rozpakować archiwum z kopią zapasową komendami:

```

cd /mnt/supersystem
tar -cvf <kopia zapasowa>

```

Trzeba też pamiętać o utworzeniu katalogów, które nie były wgrane do kopii zapasowej, oraz o przywróceniu poprawnego bootowania systemu.

Do przywracania kopii zapasowej wraz z katalogami na podstawie pliku konfiguracyjnego może posłużyć skrypt mkdirsafterbackup-upa.sh – Listing 22 – dostępny dla wariantu „A”. W wariantie „THE” wystarcza jedno-komendowy skrypt mkdirsafterbackupthe.sh – Listing 23. Wariant „THE” w fazie po przywróceniu kopii bezpieczeństwa niestety nie korzysta z pliku konfiguracyjnego. Oczywiście czytelnik może go przerobić, do czego zachęcam.

**Listing 22. Skrypt mkdirsafterbackupa.sh**

```

#!/bin/ksh
# First parameter $1 - config file

set -e

for i in `cat $1`
do
    mkdir $i
done

```

**Listing 23. Skrypt/komenda mkdirsafterbackupthe.sh**

```

mkdir /cdrom /mnt /pendrive /proc /root

```

## Przypisy

1. UNIX <http://www.opengroup.org/unix>
2. GNU <https://www.gnu.org/home.pl.html>
3. Security Patches Linux 2.2.20/21/22 <https://goo.gl/RRjZaY>
4. Linux From Scratch <http://www.linuxfromscratch.org/>
5. Gentoo Linux <https://www.gentoo.org/>
6. Net BSD <https://www.netbsd.org/>
7. RFC4634 <https://tools.ietf.org/html/rfc4634>
8. Advanced Intrusion Detection Environment <http://aide.sourceforge.net/>
9. Collect2, #Id <https://gcc.gnu.org/onlinedocs/gccint/Collect2.html>
10. <https://www.gnu.org/software/binutils/>
11. GCC <https://gcc.gnu.org/>
12. Skrypty w wersji A <https://github.com/potentejda/scriptnconf>
13. Skrypty w wersji THE <https://github.com/potentejda/scriptnconfthe>
14. Korn shell <http://www.kornshell.com/>
15. Bash <https://www.gnu.org/software/bash/>
16. #man sed
17. SED <https://goo.gl/2JvqNs>
18. System V ABI specification <https://goo.gl/2DUi1F>
19. Artur M. Kalinowski „Metody Inwigilacji i elementy informatyki śledczej” ISBN: 978-83-923745-4-1
20. #man bash
21. #man Id



### WOJCIECH BARTOSZ POTENTAS

[Wojciech.Potentas@gmail.com](mailto:Wojciech.Potentas@gmail.com)

System Engineer w Ericsson Polska Sp. z o.o. pracujący w obszarze LTE. Wcześniej programista i tester systemów GSM/WCDMA Core Network, Radio. Inżynier kierunku Informatyka na Wydziale Elektrotechniki, Elektroniki, Informatyki i Automatyki Politechniki Łódzkiej. Szczęśliwie żonaty. Ojciec dwójki dzieci.

# Lepsza jakość oprogramowania



**Usługi**  
testerzy.pl

Dołącz do naszego zespołu – [testerzy.pl/kariera](http://testerzy.pl/kariera)

# Programowanie interaktywnych diagramów z wykorzystaniem GoJS

Od dłuższego czasu możemy zauważać transformację tego, czym jest sieć WWW. Podczas gdy początkowo mieliśmy do czynienia jedynie z prostymi stronami tekstowymi, dziś potrafią być pełnoprawnymi aplikacjami. Programowanie ich, w szczególności graficznych i interaktywnych, nie jest wcale tak prostym zadaniem – jednak z pomocą przychodzi sporo bibliotek JavaScriptowych mających na celu to ułatwić. W niniejszym artykule chciałbym wziąć na tapet GoJS – bibliotekę stworzoną z myślą o interaktywnych wizualizacjach danych.

## WPROWADZENIE DO GOJS

GoJS to rozbudowana biblioteka dla języka JavaScript do tworzenia interaktywnych diagramów oraz wizualizacji wykorzystujących HTMLowy Canvas, działających na różnych nowoczesnych platformach i przeglądarkach. Programista dostaje do dyspozycji pełen zestaw narzędzi do zapewnienia interaktywności, wśród których wymienić można by na przykład wsparcie dla drag'n'drop, obsługę schowka, szeroką gamę gotowych layoutów diagramów, palety, podglądy czy wsparcie dla cofania i ponawiania. Co warto podkreślić, biblioteka jest napisana bez żadnych dodatkowych zależności – ani od zewnętrznych bibliotek, ani od funkcjonalności serwerowych. Dzięki temu nie jesteśmy w żaden sposób ograniczani – możemy z użyciem biblioteki budować dowolny rodzaj aplikacji (frontendową, mobilną, desktopową), a tą wykorzystywać tylko do przeznaczonego jej celu.

Biblioteka jest jednak płatna, aczkolwiek możliwości, jakie oferuje, potrafią zaoszczędzić sporo czasu developerskiego w porównaniu do pracy na ogólnodostępnych darmowych bibliotekach wykorzystywanych w podobnych celach, takich jak D3 czy Fabric.js. Główna różnica to udostępnienie pełnej interaktywności i mnóstwa gotowych narzędzi, dzięki którym bardzo wiele różnego rodzaju aplikacji diagramowych (i nie tylko) można stworzyć w bardzo krótkim czasie. Sprawia to, że dla programisty aplikacje służące na przykład do prezentacji przebiegu procesów (w tym z danymi aktualizowanymi w czasie rzeczywistym), edycji UML, tworzenia map myśli czy też projektowania układów elektronicznych stają się prostsze do zrobienia.

## TWORZYMY PROSTĄ APLIKACJĘ

W artykule tym chciałbym przedstawić podstawowe możliwości biblioteki poprzez stworzenie bardzo prostej aplikacji (pod względem kodu) do tworzenia niezbyt skomplikowanych diagramów. Punktem wyjściowym dla nas w tym momencie powinien być plik HTML, gdzie zimportujemy bibliotekę. Możemy ją pobrać (zarówno bezpośrednio ze strony, jak i z npm) bądź wykorzystać CDN. Kolejnym krokiem, jaki wykonujemy, jest dodanie elementu div, w którym diagram będzie się znajdować. Co ważne, nie tworzymy sami canvasu, robi to za nas biblioteka. Sam div natomiast możemy dowolnie ostylować, by pasował do naszej witryny. Nasz plik powinien wyglądać mniej więcej tak:

**Listing 1. Początkowy wygląd pliku index.html**

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/
      gojs/1.8.12/go.js"></script>
  </head>
  <body>
    <div id="diagram-content" style="height: 800px;
border: 1px solid black;"></div>
  </body>
</html>
```

Kolejnym krokiem będzie stworzenie właściwego kodu JavaScript tworzącego diagram. W tym celu utworzymy plik *diagram.js*, w którym zamieścimy całą logikę aplikacji. Możemy tworzyć go w dowolnej konwencji – GoJS nie narzuca nam żadnego konkretnego sposobu budowy źródeł. Osobiście na potrzeby tego przykładu utworzę IIFE (Immediately Invoked Function Expression) o nazwie *diagram*, z udostępnioną na zewnątrz funkcją *initDiagram*. Dzięki zastosowaniu IIFE będziemy mogli część naszej implementacji ukryć przed dostępem z zewnątrz.

W GoJS podstawą, od której zaczynamy tworzenie wszelkich obiektów, jest funkcja *go.GraphObject.make*, która wywołuje metodę fabrykującą wybranej klasy biblioteki. Jest ona na tyle często wykorzystywana, że warto przypisać ją sobie do zmiennej o krótszej nazwie. Ogólnie przyjętą konwencją jest stosowanie do tego znaku dolara (warto uważać na to w przypadku korzystania równolegle z jQuery). Funkcja ta przyjmuje jako pierwszy argument klasę, której instancja ma zostać utworzona, a jako kolejne wartości poszczególnych pól obiektu. Warto zaznaczyć, że zwykle jako drugi argument jest podawany string z wartością jednego wybranego pola, zdefiniowanego w obiekcie – zwykle którego wartość jest najbardziej istotna w danym kontekście, np. rodzaj tworzonej figury.

Pierwsze, co chcemy utworzyć, to sam diagram. Znajduje się on pod klasą *go.Diagram*, a najważniejszym parametrem jest id elementu, w którym ma zostać umieszczony. Stąd też będzie my wywoływać *\$(go.Diagram, 'diagram-content')*. Warto przejrzeć w dokumentacji, jakie pola posiada obiekt diagramu, aby lepiej sprecyzować jego działanie. Możemy na przykład przypisać polu *initialContentAlignment* wartość *go.Spot.Center*, dzięki czemu elementy diagramu na starcie aplikacji będą umieszczone na jego środku (domyślnie lewy górny róg). Zamieścmy to



## Utknałeś w nudnym projekcie?

Mamy na to rozwiązanie!

Dołącz do Synergy Codes – zacznij ciekawą pracę projektowania i programowania aplikacji webowych

**pasjonat Javascript  
i Front-End developmentu**

**eksperci od wizualizacji  
danych i aplikacji webowych**

**RAZEM**

Zmierzymy się  
z wdrażaniem najnowszych  
technologii zgodnie  
ze światowymi trendami



Pracujemy w technologiach:



Synergy Codes to firma  
z pięknego Wrocławia!

PS Jeżeli jesteś backendowcem szukamy też .NET developerów.  
Projekty krótko i długoterminowe! Zero nudy gwarantowane!

w naszym `initDiagram`. Następnie pamiętajmy, aby wywołać tę funkcję w naszym pliku HTML. W rezultacie powinniśmy otrzymać następujące pliki JS i HTML:

#### Listing 2. Szkielet diagram.js i tworzenie pustego diagramu

```
diagram = (function(){
    var $ = go.GraphObject.make;
    var diagram;
    var initDiagram = function () {
        diagram = $(go.Diagram,
            'diagram-content',
            {initialContentAlignment: go.Spot.Center
        });
        return { initDiagram };
    })();
});
```

#### Listing 3. Wywołanie tworzenia diagramu

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/
    gojs/1.8.12/go.js"></script>
    <script src="diagram.js"></script>
</head>
<body>
    <div id="diagram-content" style="height: 800px;
border: 1px solid black;"></div>
    <script>
        window.onload = function() {
            diagram.initDiagram();
        };
    </script>
</body>
</html>
```

Na tę chwilę niestety nie zobaczymy jeszcze żadnego działania, gdyż nasz diagram jest pusty. Jedyne, po czym rozpoznamy, że GoJS działa, to blokada domyślnej akcji kliknięcia prawego przycisku myszy na diagramie.

## PREZENTACJA DANYCH NA DIAGRAMIE

Podstawą dla diagramu jest utworzenie modelu danych, na którym będzie on operować. Model składa się z dwóch tablic – `nodedataArray` (dane węzłów) i `linkdataArray` (dane krawędzi). Węzły to główne elementy diagramu reprezentujące graficznie obiekty (np. encja na diagramie związków encji). Krawędzie (linki) to połączenia między węzłami (skierowane bądź nie), które w zależności od typu diagramu mogą mieć różne znaczenie (np. relacja między encjami na diagramie związków encji).

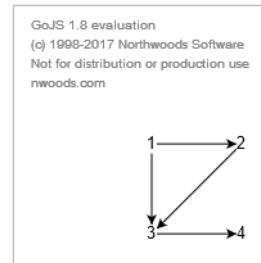
Poszczególne węzły definiujemy jako obiekty, gdzie jedynym wymaganym polem jest `key`, czyli unikalny identyfikator. Oprócz tego możemy definiować opcjonalnie kategorie obiektów (`category`), a także uzupełnić dowolnymi własnymi danymi, które będą przechowywane wraz z węzłem na diagramie. Krawędzie definiujemy również jako obiekty, które mają wymagane dwa pola – `from` i `to`, których wartości określają, jakie węzły są łączone. Podobnie jak w przypadku węzłów, możemy dopisać także własne dodatkowe dane.

Model diagramu znajduje się pod polem `model` obiektu diagramu. Najlepiej zdefiniować go, tworząc obiekt typu `go.GraphLinksModel`, do którego przekazujemy węzły i krawędzie. Przykładowa definicja modelu może wyglądać następująco:

#### Listing 4. Definicja modelu diagramu

```
diagram.model = $(go.GraphLinksModel, {
    nodedataArray: [
        {key: 1, category: 'first'},
        {key: 2, category: 'second'},
        {key: 3, category: 'second'},
        {key: 4, category: 'third'}
    ],
    linkdataArray: [
        {from: 1, to: 2},
        {from: 2, to: 3},
        {from: 1, to: 3},
        {from: 3, to: 4}
    ]
});
```

Po zamieszczeniu jej w naszym `initDiagram` powinniśmy otrzymać po uruchomieniu bardzo prosty diagram, gdzie zobaczymy wartości wpisane w polach `key` połączone między sobą krawędziami.



Rysunek 1. Nasz pierwszy diagram

Kolejnym krokiem, jaki powinniśmy podjąć, jest nadanie naszym obiektom wyglądu. Wygląd tworzymy poprzez utworzenie obiektu typu `go.Node`, w którym następnie tworzymy kolejne obiekty definiujące kształt (`go.Shape`). Najprostsza definicja wyglądu obiektu (`template`) wyglądałaby więc tak:

#### Listing 5. Najprostsza definicja wyglądu obiektu

```
diagram.nodeTemplate = $(go.Node, 'Auto', $(go.Shape,
    'Circle'));
```

W tym przypadku utworzymy czarne koło, które będzie reprezentować obiekt. Auto oznacza sposób rozmieszczania poszczególnych kształtów. W przypadku Auto każdy kształt, jaki byśmy zdefiniowali wewnątrz `go.Node`, będzie układany warstwowo na środku obiektu. W przypadku kształtów wybrany został tutaj Circle, aczkolwiek jest udostępnionych wiele różnych, które możemy znaleźć pod adresem: <https://gojs.net/latest/samples/shapes.html>. Moglibyśmy także w obiekcie kształtu ustawić wartości pól – moglibyśmy wówczas określić takie rzeczy jak wypełnienie, kontur czy rozmiar. Dodatkowo wartości te możemy pobierać z modelu danych – wykorzystujemy wówczas do tego klasy `go.Binding`.

Inny sposób definicji wyglądu to stworzenie kształtu z wykorzystaniem definicji ścieżek SVG. Wtedy taką ścieżką zamieszczamy jako wartość pola `geometryString`. Przykładowa definicja template wyglądałaby wtedy następująco (dodatkowo ustawimy rozmiar i kolor wypełnienia):

#### Listing 6. Definicja wyglądu wykorzystująca ścieżkę SVG

```
diagram.nodeTemplate = $(go.Node,
    'Auto',
    $(go.Shape, {
```

```
geometryString:
'F M 0 0 L100 0 Q150 50 100 100 L0 100 Q50 50 0 0z',
fill: 'white',
width: 100,
height: 100
});
```

Możemy także definiować wygląd za pomocą obrazków w dowolnym formacie (w tym SVG). Wówczas, zamiast go.Shape, korzystamy z go.Picture. Co ciekawe, go.Picture może przechowywać nie tylko obrazki, ale także inne elementy typu canvas, dzięki czemu możemy tworzyć jeszcze bardziej zaawansowane wizualizacje.

Kolejnym sposobem na definicję wyglądu obiektów jest składanie wielu różnych kształtów w jeden. Zaletą takiego podejścia jest to, że po podzieleniu naszego obiektu na kilka mniejszych możemy definiować różne zachowania dla każdej z części (np. jeden z kształtów może być przyciskiem, na którym zdefiniujemy akcję kliknięcia – tylko ten fragment wówczas będzie miał przypisane to zdarzenie) W celu tworzenia złożonych obiektów warto zapoznać się ze sposobami, jak mogą być rozmieszczane kształty na obiekcie. Do tej pory wykorzystywaliśmy Auto, czyli umieszczenie wszystkiego na środku. Oprócz niego mamy do dyspozycji także inne, z czego najbardziej podstawowe to: Horizontal (od lewej do prawej), Vertical (z góry na dół), Spot (umieszczenie na odgórnio określonych pozycjach jak Center, Top, Bottom itd.). Kolejne kształty przekazujemy jako kolejne argumenty funkcji tworzącej Node i w takiej też kolejności są umieszczone. Na przykład dwa prostokąty, każdy o innym wyglądzie, położone jeden na drugim, byłyby zdefiniowane w następujący sposób:

#### Listing 7. Definicja wyglądu składającego się z kilku kształtów

```
diagram.nodeTemplate = $(go.Node,
    'Vertical',
    $(go.Shape,
        'Rectangle', {
            width: 50,
            height: 50,
            strokeWidth: 0,
            fill: 'yellow',
        }),
    $(go.Shape,
        'Rectangle', {
            width: 100,
            height: 50,
            strokeWidth: 5,
            stroke: 'red',
            fill: 'green'
        }));
});
```

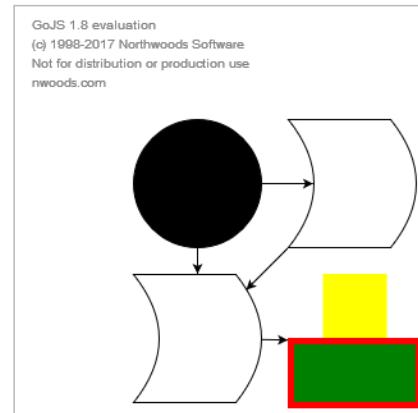
Co jednak, jeżeli chcielibyśmy różne rodzaje obiektów? W tym celu zamiast przypisywać wygląd do nodeTemplate, uzupełniamy mapę nodeTemplateMap, gdzie kluczem jest kategoria obiektu (category, które definiowaliśmy w modelu), a wartością template. Na potrzeby tego przykładu założymy, że wykorzystamy wyżej opisane definicje dla naszych kategorii: first będzie kołem, second kształtem SVG, third dwoma prostokątami. Ze względów praktycznych warto jednak nie przypisywać template bezpośrednio do mapy, tylko stworzyć funkcję zwracającą, jaki template przypiszemy do jakiej kategorii, i następnie uzupełniać danymi z niej mapę nodeTemplateMap. Powód tego wyjaśnię w dalszej części artykułu. Warto zwrócić uwagę, że mapę tę należy uzupełnić zanim zasiliśmy diagram modelem. Na przykład mogliby to wyglądać następująco (ze względu na czytelność pominięto niektóre fragmenty kodu):

#### Listing 8. Propozycja sposobu korzystania z nodeTemplateMap

```
var getTemplates = function () {
    return [
        category: 'first',
        template: $(go.Node, 'Auto', $(go.Shape, 'Circle'))
    ],
    category: 'second',
    template: $(go.Node,
        'Auto',
        $(go.Shape, {
            // ...
        })
    ),
    category: 'third',
    template: $(go.Node,
        'Vertical',
        // ...
    ))
];
}

var initDiagram = function () {
// ...
getTemplates().forEach(x =>
    diagram.nodeTemplateMap.add(x.category, x.template));
diagram.model = $(go.GraphLinksModel, {
    // ...
});
}
```

Po napisaniu tego powinniśmy uzyskać następujący efekt:



Rysunek 2. Diagram z trzema różnymi template dla węzłów

Jak widać, nasz diagram zaczyna nabierać kształtów. Dodatkowo warto wspomnieć, że oprócz wyglądu węzłów można także ustalać wygląd krawędzi. Definiuje się go analogicznie, z tą różnicą, że zamiast nodeTemplate czy nodeTemplateMap mamy linkTemplate i linkTemplateMap, a przy definiowaniu template zamiast tworzyć obiekt go.Node, tworzymy obiekt go.Link.

## USTAWIANIE I EDYCJA KRAWĘDZI

Nasz diagram zaczyna wyglądać coraz lepiej, jednak można zauważać, że po pierwsze, brakuje mu możliwości edycji (jedynie, co można, to przesuwać węzły i je klonować z użyciem klawisza CTRL), a po drugie – moglibyśmy chcieć, aby krawędzie rysowały się między konkretnymi punktami naszych węzłów. Szczególnie widać to na obiektach, które powstały ze ścieżek SVG – najbardziej pożąданie byłoby ustawnienie tutaj wejścia z lewej strony i wyjścia z prawej. Aby to osiągnąć, musimy zdefiniować porty.

Porty są częściami węzłów, które służą do niczego innego jak ustawiania krawędzi względem reszty obiektu. Mogą nimi być do-

wolne już istniejące elementy węzła (nawet cały węzeł), jak i specjalnie do tego celu utworzone nowe elementy. Definiuje się je poprzez ustawienie wartości pola `portId`. Oprócz tego możemy m.in. także definiować, czy możemy z portu tworzyć nowe linki (`pole fromLinkable`) oraz czy możemy do niego dołączać nowe linki (`toLinkable`). Na potrzeby przykładu porty będą definiowane tutaj jako widoczne kształty, aczkolwiek mogą nimi być dowolne obiekty (w tym niewidzialne).

W przypadku pierwszej kategorii chcielibyśmy port umieścić w samym środku. Aby było widać, że krawędzie tam trafiają, zmienimy przy okazji kolor koła, aby było białe, oraz ustalimy mu stały rozmiar. Od razu ustawiemy też, żeby można było tworzyć z tego portu połączenia w obie strony. Definicja kształtu będzie wówczas wyglądać następująco:

#### Listing 9. Definicja kształtu rozbudowanego o port

```
$(go.Node, 'Auto',
$(go.Shape,
'Circle', {
width: 100,
height: 100,
fill: 'white'
}),
$(go.Shape,
'Rectangle', {
portId: '',
width: 20,
height: 20,
fill: 'black',
fromLinkable: true,
toLinkable: true
}))
```

Jak wspomniałem wcześniej, istotne jest, aby pole `portId` miało ustawioną wartość, jednak nie zabrania nam to użycia pustego stringa. W przypadku nadania pustego stringa port ten będzie portem domyślnym.

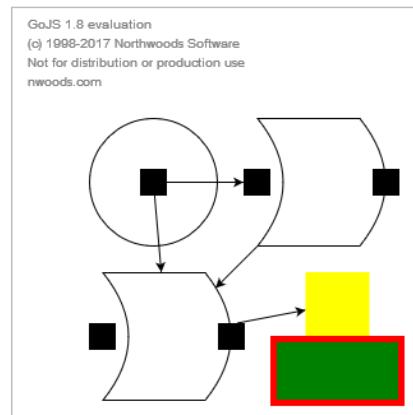
Dla drugiej kategorii chcielibyśmy zdefiniować port po jej lewej i prawej stronie. Możemy to osiągnąć na dwa sposoby: zmieniając rozmieszczenie elementów na `Horizontal` i ustawiając kształty w kolejności port-główny kształt-port lub zmieniając na `Spot` i wówczas ustawiamy, że porty mamy w pozycjach `Left` i `Right`, a główny kształt w `Center`. Bardziej elastyczny jest ten drugi sposób i też z niego skorzystamy. Z lewej strony umieszcmy taki sam prostokąt jak na pierwszym kształcie, ustawiając, że można jedynie linkować do niego, natomiast z prawej taki, że możemy jedynie wyprowadzać krawędzie z niego. Dla rozróżnienia ich powinniśmy nadać im inne identyfikatory niż pusty string (np. `entry` i `exit`). Taka definicja wyglądałaby wówczas następująco:

#### Listing 10. Definicja kształtu rozbudowanego o dwa porty

```
$(go.Node,
'Spot',
$(go.Shape, {
// ...
}),
$(go.Shape,
'Rectangle', {
portId: 'entry',
width: 20,
height: 20,
fill: 'black',
alignment: go.Spot.Left,
fromLinkable: false,
toLinkable: true
}),
```

```
$(go.Shape,
'Rectangle', {
portId: 'exit',
width: 20,
height: 20,
fill: 'black',
alignment: go.Spot.Right,
fromLinkable: true,
toLinkable: false
}))
```

Dla trzeciej kategorii jedynie zmodyfikujmy kwadrat na górze, aby stał się on portem, poprzez dopisanie `portId` z pustym stringiem oraz `fromLinkable` i `toLinkable` ustawionych na `true`. Po stworzeniu portów diagram powinien prezentować się następująco:



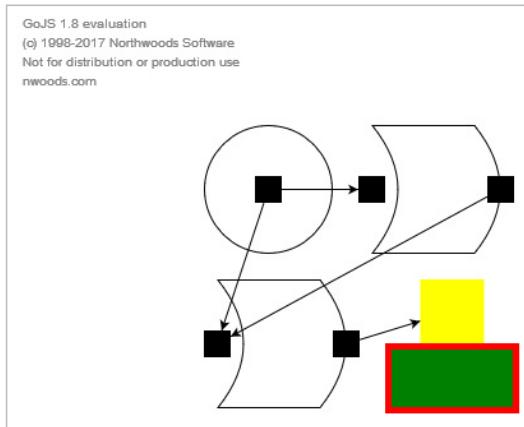
Rysunek 3. Diagram ze wstępnią konfiguracją portów

Jak widać na powyższym rysunku, w przypadku pierwszej i trzeciej kategorii porty działają doskonale, natomiast nie działają dla drugiej. Podobnie jest z rysowaniem linków. Bez problemu działa rysowanie między pierwszą i trzecią kategorią, natomiast w drugiej krawędzie wyskakują z portów. Powodem są dwie kwestie. Po pierwsze, w modelu linków nie zdefiniowaliśmy, między jakimi portami mają zachodzić połączenia. Po drugie, nie zostało w modelu ustawione, jak przechowywać połączenia między portami. Przenieśmy się więc z powrotem do `initDiagram` i definicji modelu. Do pól `GraphLinksModel` musimy dodać dwa pola: `linkFromPortIdProperty` oraz `linkToPortIdProperty`, którymi ustalamy, jak w modelu definiujemy wykorzystywane porty. Dodatkowo w `linkdataArray` powinniśmy zdefiniować, jak prowadzone będą krawędzie dla węzłów 2 i 3 (czyli tych, co należą do drugiej kategorii). Zmiany te powinny wyglądać następująco:

#### Listing 11. Zmiany w modelu umożliwiające wykorzystanie portów

```
diagram.model = $(go.GraphLinksModel, {
linkFromPortIdProperty: 'fromPort',
linkToPortIdProperty: 'toPort',
nodedataArray: [
// ...
],
linkdataArray: [
{from: 1, to: 2, toPort: 'entry'},
{from: 2, to: 3, fromPort: 'exit', toPort: 'entry'},
{from: 1, to: 3, toPort: 'entry'},
{from: 3, to: 4, fromPort: 'exit'}
]
});
```

Teraz krawędzie na diagramie wyglądają tak jak powinny:



Rysunek 4. Diagram po prawidłowym skonfigurowaniu portów

## PALETA I KOŃCOWE SZLIFY

Aby nasz diagram był w pełni interaktywny, brakuje nam jeszcze możliwości wstawiania elementów. Do tego celu utworzymy wbudowaną w GoJS paletę. Paleta to oddzielnny canvas, który działa w taki sposób, że wyświetla wszystkie zdefiniowane w niej węzły i umożliwia przeniesienie ich na główny diagram. Konfigurujemy ją analogicznie do diagramu, jednak poprzez utworzenie obiektu typu `go.Palette`. Aby jednak to zrobić, najpierw powinniśmy dodać do pliku HTML dodatkowy div, w którym paleta zostanie umieszczona. Do tego możemy dopisać w pliku JS funkcję do inicjalizacji palety, którą również udostępnimy na zewnątrz (`initPalette`). W naszym przykładzie umieścimy paletę obok diagramu, więc plik HTML możemy zedytować w następujący sposób:

### Listing 12. Dodanie palety do pliku HTML

```
<div id="palette-content" style="height: 800px; width: 200px;
display: inline-block; border: 1px solid black"></div>
<div id="diagram-content" style="height: 800px; width:
calc(100% - 220px);
display: inline-block; border: 1px solid black;"></div>
<script>
window.onload = function() {
    diagram.initDiagram();
    diagram.initPalette();
};
</script>
```

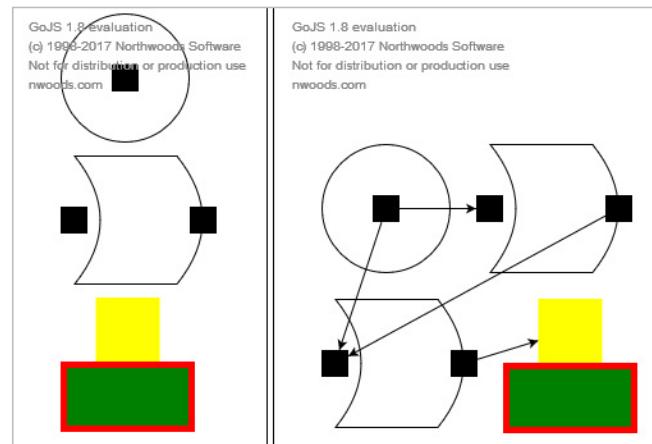
Kolejnym krokiem będzie uzupełnienie utworzonej przez nas funkcji `initPalette`. Jak wspomniałem wcześniej, inicjalizacja palety niewiele się różni od inicjalizacji diagramu. Właśnie w tym momencie wykorzystamy po raz drugi naszą funkcję `getTemplates()` – dzięki temu nie musimy definiować wyglądu kształtów po raz kolejny, oddzielnie dla palety. Jednak warto zaznaczyć, że możemy utworzyć inny – przy przenoszeniu na diagram znaczenie ma kategoria elementu, a nie jej wygląd. Dodatkowo, aby umożliwić przenoszenie z palety na diagram, musimy do inicjalizacji diagramu dodać pole `allowDrop` i ustawić mu wartość `true`. Kod powinien prezentować się następująco:

### Listing 13. Inicjalizacja palety

```
var initDiagram = function () {
    diagram = $(go.Diagram,
    'diagram-content', {
```

```
    initialContentAlignment: go.Spot.Center,
    allowDrop: true
});
// ...
}
var initPalette = function () {
    palette = $(go.Palette, 'palette-content');
    getTemplates().forEach(x => palette.nodeTemplateMap.add(x,
category, x.template));
    palette.model.nodeDataArray =
    [key: 1, category: 'first'],
    {key: 2, category: 'second'},
    {key: 3, category: 'third'}
]
}
```

A aplikacja będzie prezentować się teraz tak:



Rysunek 5. Diagram z dodaną paletą

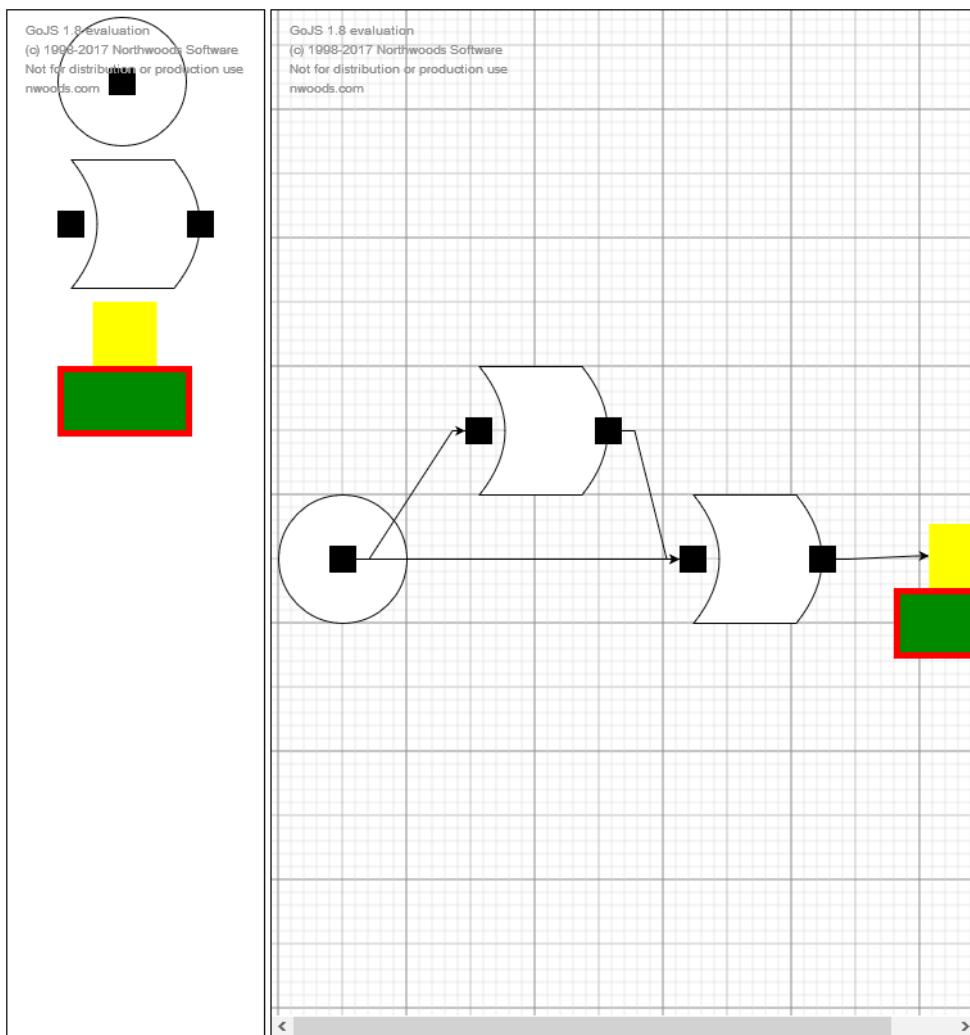
Dodatkowo na sam koniec, aby nieco poprawić wygląd aplikacji, dodajmy jeszcze dodatkowo dwie rzeczy: siatkę, do której będą przylegać elementy, oraz ich autopozycjonowanie.

Zaczniemy od siatki. Tę znajdziemy pod `diagram.grid` i jedynie, co musimy zrobić, to zmienić jej właściwość `visible` na `true`. Natomiast aby dodać przyciąganie, należy je ustawić we wbudowanym narzędziu przeciągania dostępnym pod `diagram.toolManager.draggingTool`. Możemy tu wykorzystać dwie właściwości: `isGridSnapEnabled` do włączenia przyciągania oraz `gridSnapCellSize` do zdefiniowania wielkości siatki (dla naszego przykładu zdefiniujmy rozmiar `50x50`). Jeżeli korzystalibyśmy również z modyfikacji rozmiaru węzłów (`resizingTool`), to ustawilibyśmy także w nim analogiczne pola. Kod odpowiedzialny za ustawienie siatki powinien wyglądać następująco:

### Listing 14. Ustawienie siatki na diagramie

```
diagram.grid.visible = true;
diagram.toolManager.draggingTool.isGridSnapEnabled = true;
diagram.toolManager.draggingTool.gridSnapCellSize = new
go.Size(50, 50);
```

Pozycjonowanie elementów najlepiej osiągnąć poprzez zmianę layoutu diagramu. Domyślany layout nie definiuje w żaden sposób automatycznego rozmiawczania elementów (jedynie układa je tak, aby nie nachodziły na siebie). Oferują je natomiast inne: `GridLayout` (układanie na siatce), `TreeLayout` (drzewo), `ForceDirectedLayout` (grafy z odpychaniem od siebie elementów), `LayeredDigraphLayout` (graf rozmiawczony w kolumny i wiersze) oraz `CircularLayout` (rozmiawczanie węzłów na okręgu). Layout ustawiamy jako kolejne pole przy inicjalizacji diagramu



Rysunek 6. Diagram z włączoną siatką i ustawionym LayeredDigraphLayout

– layout, któremu przekazujemy nową instancję wybranego layoutu wraz z jego ewentualną konfiguracją. Na przykład możemy to zrealizować następująco dla LayeredDigraphLayout:

#### **Listing 15. Ustawienie layoutu Layered Digraph dla diagramu**

```
diagram = $(go.Diagram,
  'diagram-content',
  {
    initialContentAlignment: go.Spot.Center,
    allowDrop: true,
    layout: $(go.LayeredDigraphLayout)
  });
});
```

Efekt został przedstawiony na Rysunku 6.

Można także sprawdzić inne dostępne layouty, jednak należy pamiętać o ich ograniczeniach. Na przykład TreeLayout nie poradzi sobie z naszą sytuacją, gdzie do jednego węzła prowadzą dwie krawędzie.

## PODSUMOWANIE

Jak można było się przekonać w trakcie lektury tego artykułu, bardzo niewielkim nakładem pracy uzyskaliśmy dość rozbudowaną aplikację, korzystając jedynie z udostępnionej funkcjonalności biblioteki GoJS, bez dodatkowego hackowania. Całość kodu Ja-

vaScript zamknęła się w około 120 liniach kodu, co tym bardziej pokazuje, że nie robiliśmy nic złożonego. GoJS pozwala nam zaoszczędzić sporo czasu, który zapewne poświęcilibyśmy na stworzenie chociażby palety, obsługi elementów na canvasie czy też nietrywialnych rzeczy jak automatyczne pozycjonowanie elementów. Plusami tej biblioteki są także przejrzysta dokumentacja, mnóstwo gotowych przykładów oraz bardzo dobry support producenta. Zwolennicy TypeScript także będą zadowoleni, ponieważ są udostępnione oficjalne typingi, a twórcy także udostępniają za darmo rozszerzenia biblioteki napisane właśnie w TypeScript.

Jednak w trakcie tego krótkiego artykułu nie sposób było przedstawić wszystkich możliwości biblioteki, co warto zrobić na własną rękę poprzez zapoznanie się z przykładami użycia dostępnymi na stronie producenta.

### TOMASZ ŚWISTAK



Absolwent informatyki na Politechnice Wrocławskiej. Od 3 lat pracuje jako .NETowy full-stack developer w firmie Synergy Codes, gdzie tworzy zaawansowane wizualizacje danych w aplikacjach webowych. Oprócz tego interesuje się tematyką sztucznej inteligencji i gamedevem.

# follow the blue rabbit...

coming soon

# Algorytmy ewolucyjne: wydajne metody rozwiązywania złożonych problemów optymalizacyjnych

Kiedy mówimy o metodach Soft Computingu, nierzadko wymieniamy jedynie sztuczne sieci neuronowe. Jednakże innym koronnym przykładem algorytmów służących do rozwiązywania złożonych problemów optymalizacyjnych są algorytmy ewolucyjne. Pozwalają one bowiem na jednoczesne rozpatrywanie wielu rozwiązań, które mogą prowadzić do rozwiązania optymalnego.

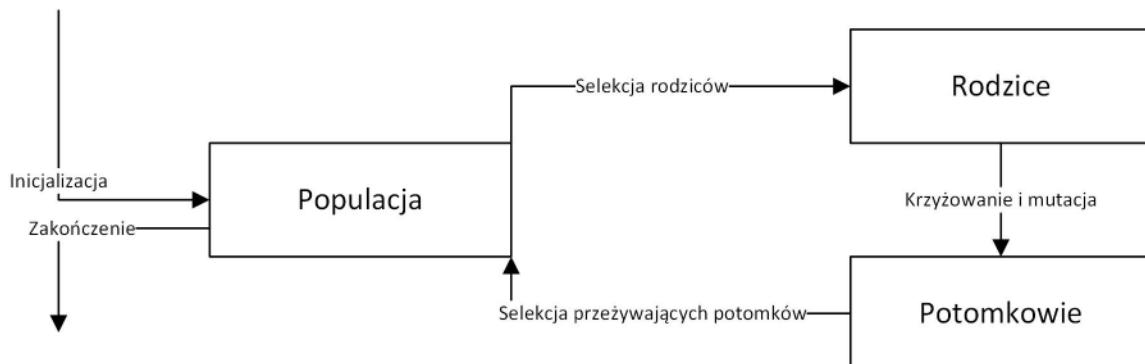
## ALGORYTMY EWOLUCYJNE

Algorytmy ewolucyjne bardzo często błędnie kojarzone są z zagadnieniami skorelowanymi z biologią. Tak naprawdę stanowią one bardzo ważną część informatyki. Co prawda zostały one zainspirowane procesem ewolucji, który rzeczywiście zachodzi w przyrodzie, jednakże nie jest ich celem jak najdokładniejsze modelowanie tegoż procesu, ale stworzenie narzędzi, luźno bazujących na inspiracjach biologicznych, pozwalających na rozwiązywanie pewnych klas problemów. W tym miejscu należałoby również podkreślić, jaką jest podstawowa różnica pomiędzy klasycznym algorymem optymalizacji a algorytmami ewolucyjnymi. Otóż w ramach pierwszej z wymienionych grup w danym momencie przetwarzany jest tylko i wyłącznie jeden punkt, na mocy którego weryfikuje się założenie o jego optymalności. W przypadku algorytmów ewolucyjnych w każdej iteracji przetwarzana jest pewna populacja rozwiązań. Na ten zbiór składają się osobniki, które reprezentują pewne rozwiązania postawionego problemu. Ponadto w ramach algorytmów ewolucyjnych możemy wyróżnić pewne specyficzne operatory, których nie uświadczymy w ramach klasycznych algorytmów optymalizacyjnych – są to chociażby operatory: mutacji i rekombinacji (które są źródłem zmian w ramach populacji) oraz selekcja (która z kolei redukuje różnorodność i prowadzi do wzrostu dopasowania w populacji). Każdy osobnik w ramach populacji jest opisywany przez wartość swojej funkcji dopasowania. Odzworowuje ona przestrzeń osobników w zbiór liczb rzeczywistych. Możemy zatem stwierdzić, że podstawowym celem algorytmów ewolucyjnych jest znalezienie osobnika o jak największej

wartości funkcji dopasowania. Zwrócmy również uwagę na to, że w tym przypadku dokonujemy optymalizacji lokalnej. Pojedyncza iteracja odnajduje lokalnie najlepszego osobnika. Ogólny schemat algorytmu ewolucyjnego przedstawiamy na Rysunku 1.

Jak zostało przedstawione na tym rysunku, pierwszym krokiem algorytmu jest inicjalizacja populacji, kiedy ten etap zostanie zakończony, przechodzimy do selekcji rodziców. Do populacji rodziców zaliczane są wybrane osobniki z populacji pierwotnej. Gdy zostanie już przygotowana populacja rodzicielska, następuje krzyżowanie osobników w parach (z jednej pary rodziców powstają dwaj potomkowie), a następnie mutacja osobników potomnych. W ten sposób tworzona jest populacja potomna. Kolejnym krokiem jest utworzenie nowej populacji pierwotnej na mocy przygotowanej populacji potomnej. Algorytm ewolucyjny może zakończyć swoje działanie w dwojakim sposobie: albo poprzez przekroczenie ustalonej (na przykład jako parametr algorytmu) liczby iteracji, albo poprzez obserwację najlepszego rozwiązania i stwierdzenie, że nie następują już żadne zmiany, co z kolei prowadzi do wniosku, że dane rozwiązanie jest najprawdopodobniej rozwiązaniem optymalnym.

Dodatkowo możemy wyróżnić trzy podstawowe grupy charakteryzujące algorytmy ewolucyjne. Pierwszą z nich, najbardziej znaną, są algorytmy genetyczne. W ramach tej grupy wszystkie osobniki mają postać ciągu bitów o stałej długości. Oczywiście wykorzystując odpowiednie przekształcenia, jesteśmy w stanie wykorzystać algorytmy z tej grupy również do rozwiązywania problemów optymalizacyjnych w przestrzeni liczb rzeczywistych. Drugą grupą są strategie ewolucyjne – w tym przypadku osobniki w po-



Rysunek 1. Ogólny schemat algorytmu ewolucyjnego

pulacji są reprezentowane przez wektor liczb rzeczywistych. Ostatnim zbiorem algorytmów jest z kolei programowanie genetyczne – tym razem każdy osobnik jest przedstawiany jako drzewo, które reprezentuje program komputerowy. W ramach tego artykułu zajmiemy się problemem optymalizacji w ramach zbioru rzeczywistego i odniesiemy się do niego z wykorzystaniem Standardowego Algorytmu Genetycznego (SGA) oraz algorytmu z reprezentacją zmiennopozycyjną (tzw. algorytmu Michalewicza). Jednakże zanim przejdziemy do prezentacji obydwu algorytmów, zwróćmy uwagę na wady i zalety algorytmów ewolucyjnych.

### Zalety algorytmów ewolucyjnych

Przede wszystkim należy wskazać, że są one uniwersalne. Mianowicie jeżeli chcielibyśmy badać inną funkcję (w celu znalezienia jej wartości optymalnej), wtedy należałoby jedynie zmienić postać funkcji przystosowania. Ponadto algorytmy ewolucyjne potrafią sobie poradzić nawet w sytuacjach, w których funkcje bazowe są funkcjami zaszumionymi czy też posiadają wiele ekstremów lokalnych (standardowe algorytmy optymalizacyjne mogłyby zatrzymać się na pierwszym odnalezionym ekstremum lokalnym). Dodatkowo algorytmy ewolucyjne są metodami wydajnymi – zwróćmy uwagę, że w każdej iteracji przetwarzamy pewien zbiór możliwych rozwiązań.

### Wady algorytmów ewolucyjnych

Algorytmy ewolucyjne mają jednak pewną podstawową wadę – są to algorytmy w dużej mierze (co zostanie przedstawione w dalszej części artykułu) oparte o wartości losowe. Implikuje to fakt, że niestety nie jesteśmy w stanie przy jednym przebiegu uzyskać dokładnej informacji o znalezionych rozwiązaniach. Najczęściej w przypadku wykorzystania algorytmów ewolucyjnych należy wielokrotnie wykonać dany program i albo uśrednić uzyskane rezultaty, albo bazować na wartościach najlepszych (najlepiej przystosowane rozwiązanie ze wszystkich przebiegów algorytmu). Co więcej, sukces (czyli odnalezienie wartości optymalnej) będzie możliwy tylko i wyłącznie wtedy, gdy prawidłowo zostanie zakodowany stawiany problem oraz odpowiednio określony funkcję przystosowania. Autorstwo sukcesu możemy przypisać programiście, gdyż nierzadko bardzo trudne jest odpowiednie sformułowanie wspomnianych parametrów.

## STANDARDOWY ALGORYTM GENETYCZNY (SGA)

W tym artykule chcielibyśmy odnieść się do Standardowego Algorytmu Genetycznego (SGA) oraz algorytmu Michalewicza. Jednakże chcielibyśmy najpierw przedstawić czytelnikowi prosty schemat takiego algorytmu, który przedstawiamy na Rysunku 2, oraz prosty opis, który pokazuje, jak można rozumieć SGA w odniesieniu np. do gry komputerowej „Football Manager”.

Pierwszym krokiem takiego algorytmu jest utworzenie populacji startowej w sposób losowy. Czymże zatem jest populacja? Otóż jest to zbiór osobników, opisywanych pewnymi parametrami. Te parametry, które stanowią deskryptor osobnika (ich ilość oraz zakres), są bezpośrednio zależne od funkcji, której optymalizację dokonujemy (liczba parametrów), oraz od użytkownika (zakres poszczególnych parametrów). Przyjmijmy zatem, że naszą optymalizowaną funkcją jest  $f(x,y) = x^2 + y^3$ . Stąd każdy osobnik będzie

opisany przez zbiór parametrów  $(x,y)$ . Aby jednak dopełnić tego obrazu, musimy podać zakres dla każdego parametru. Niech zatem  $5 \leq x \leq 20$ , natomiast  $-2 \leq y \leq 35$ . Na tej podstawie wiemy, jak mogą wyglądać osobniki w ramach naszej pierwotnej populacji. Aby to sobie zwiastować, możemy przyjąć, że gramy w popularną grę, jaką jest Football Manager. W tej grze każdy z piłkarzy jest opisany przez pewne parametry. Niech zatem nasz osobnik odpowiada piłkarzowi, natomiast  $x$  – niech oznacza siłę kopnięcia piłki, a  $y$  – celność strzału. Na początku zatem jesteśmy prezesem zespołu piłkarskiego, który nie ma jeszcze żadnych zawodników, w związku z tym angażujemy każdego, kto do nas się zgłosi. Ten element realizujemy poprzez losowanie wartości poszczególnych parametrów. Przyjmijmy zatem, że rozmiar populacji (czyli liczba piłkarzy, których chcemy ściągnąć do zespołu, wynosi 4). Po losowaniu uzyskaliśmy następujących piłkarzy:  $\{(x,y)\} = \{(5,0); (6,6); (8,11); (15,-1)\}$ . Widzimy zatem, że są to zawodnicy dosyć przeciętni, aczkolwiek stanowią oni dla nas pewną bazę.



Rysunek 2. Schemat Standardowego Algorytmu Genetycznego

Drugim krokiem naszego algorytmu jest obliczenie dopasowania wszystkich elementów populacji. Na czym polega to w ramach naszej gry? Otóż każdy z piłkarzy ma pewną notę całkowitą, która stanowi o jego wartości. Zauważmy bowiem, że dosyć trudno byłoby analizować każdego z piłkarzy poprzez wszystkie jego parametry (w naszym przypadku byłoby to banalne, aczkolwiek w realnym świecie każdy zawodnik jest opisany przez wiele parametrów – w grze Football Manager nawet około 300). Przyjmijmy więc, że taką funkcją jest funkcja  $f(x,y) = x + y$ . Zatem nasi zawodnicy otrzymają następujące noty: {5, 12, 19, 14}.

Trzeci krok to pytanie, czy trener jest usatysfakcjonowany poziomem swojego najlepszego (gdy szukamy maksimum funkcji) albo najgorszego (gdy szukamy minimum) zawodnika. Jeżeli tak jest, to algorytm kończy swoje działanie, jeżeli jednak nie, to może swobodnie przejść do kroku czwartego.

Czwarty krok algorytmu SGA to wybór osobników do populacji rodziców. Otóż w naszym przypadku polega to na wybraniu tych zawodników, którzy będą prowadzili zajęcia z juniorami (uwaga, jeden zawodnik może prowadzić więcej niż jedną grupę juniorską, a każda grupa juniorska może mieć dwóch trenerów – czasami może być to jeden zawodnik jako dwaj trenerzy). Na czym zatem polegać będzie nasz wybór? Otóż trener zarządził przedstawienie wyników not całkowitych w formie liniowej funkcji dopasowania. Jak zatem wykonać takie polecenie menedżera? Otóż musimy tak dobrą parametry funkcji liniowej, która przekształci dotychczasową wartość dopasowania (czyli notę całkowitą), aby wartość średnia się nie zmieniła, natomiast najlepszy zawodnik będzie pełnił rolę k trenerów (k jest parametrem algorytmu i jest wybierane

przez użytkownika). Aby tego dokonać, korzystamy z następujących wzorów:

$$\begin{cases} f_{AVG} = a \cdot f_{AVG} + b \\ k \cdot f_{AVG} = a \cdot f_{MAX} + b \end{cases} \quad (1)$$

Zauważmy, że w tym przypadku wartość o indeksie AVG to wartość średnia, natomiast MAX – oznacza wartość maksymalną. Nieustety takie skalowanie wiąże się również z pewnymi obostrzeniami – mianowicie w bardzo dojrzalej populacji (gdzie większość osobników – piłkarzy – ma zbliżoną notę całkowitą, z wyjątkiem kilku osobników znacząco się różniących) możemy otrzymać ujemne wartości po wykonaniu tej operacji. W związku z tym powinniśmy drugi z warunków przedstawionych w ramach (1) zastąpić przez warunki zaprezentowane w (2).

$$\begin{cases} f_{AVG} = a \cdot f_{AVG} + b \\ 0 = a \cdot f_{MIN} + b \end{cases} \quad (2)$$

Gdzie wartość z indeksem MIN oznacza wartość minimalną funkcji przystosowania (oceny ogólnej zawodnika). Po wykonaniu tego kroku mamy pewność, że najlepszy zawodnik będzie pełnił rolę k trenerów.

Kolejnym elementem w budowaniu populacji rodzicielskiej (czyli zbioru trenerów) jest procedura Stochastic Remainder Sampling (SRS). Jej głównym celem jest obliczenie, dla ilu grup każdy z zawodników na pewno będzie pełnił rolę trenera. Te wartości są obliczane na podstawie prawdopodobieństwa wylosowania zawodnika i całkowitej liczby osobników w populacji. Warto również wyjaśnić, czym jest wspomniane „prawdopodobieństwo wylosowania zawodnika”. Otóż jest to stosunek noty ogólnej zawodnika do sumy not wszystkich zawodników. Policzymy zatem tę wartość dla zawodnika o numerze pierwszym, jego nota ogólna wynosi 5, suma wszystkich not z kolei wynosi 50, co zatem oznacza, że:

$$p(x_1) = \frac{5}{50} = 0,1$$

Ilu zatem zawodników  $x_1$  będzie na pewno prowadzić zajęcia? Tę wartość obliczamy poprzez wzór następujący:

$$[s \cdot p(x_1)] = [4 \cdot 0,1] = [0,4] = 0$$

W danym wzorze parametr oznacza liczbę populacji. Widzimy zatem, że pierwszy zawodnik nie ma żadnej pewności, że poprowadzi zajęcia. Analogicznie obliczamy liczbę ról trenera, jakie będą pełnić pozostała zawodnicy. Na tej postawie wiemy, ile miejsc w populacji trenerskiej mamy już obsadzone.

Następnym krokiem po przeprowadzeniu SRS jest uzupełnienie populacji o brakujących trenerów. Otóż bazując na wartościach:

$$s \cdot p(x_i) - [s \cdot p(x_i)]$$

weryfikujemy, czy zadany trener powinien brać udział w dalszym obsadzaniu posad. Jeżeli ta wartość jest niezerowa, wtedy danego trenera bierzemy pod uwagę z wartością dopasowania równą:

$$s \cdot p(x_i) - [s \cdot p(x_i)]$$

Po utworzeniu populacji, która będzie brała udział w dalszym obsadzaniu miejsc w populacji trenerskiej, możemy przejść do obserwacji wolnych miejsc. Wykonamy ten element za pomocą algorytmu koła ruletki, który polega na tym, że na pewnym odcinku odkładamy prawdopodobieństwo wystąpienia każdego elementu z populacji (na mocy nowych wartości przystosowań). W ten sposób tworzymy pewne przedziały, które określają poszczególne osobniki. Przeprowadzamy losowanie (wielokrotnie – tyle razy, ilu trenerów nam brakuje). W momencie, w którym wylosowana wartość trafia do któregoś z przedziałów, ten trener (osobnik) jest dodawany do populacji trenerskiej (rodzicielskiej). W ten sposób utworzyliśmy populację trenerów (rodziców).

Kolejnym etapem algorytmu jest kojarzenie trenerów w pary (czyli osobników z populacji rodzicielskiej w pary). Ten element jest wykonywany w sposób losowy i za każdym razem powinien być wykonywany z innym ziarnem (zostało to przez nas zapewnione, aczkolwiek odniesiemy się do tego elementu w części dotyczącej samej implementacji). Po połączeniu trenerów w pary przechodzimy do kolejnego kroku naszego algorytmu.

Zgodnie z opisem algorytmu w tym momencie następuje krzyżowanie w parach. Aby uzmysłowić czytelnikowi, jak może to wyglądać w ramach naszej gry, uznajmy, że każda para trenerów pracuje nad dwoma zawodnikami i każdy z dwóch zawodników uczy się od swojego trenera, przejmując część jego umiejętności. Aby móc to efektywnie wykonać, każdy z osobników (trenerów) powinien zostać przedstawiony w formie kodu binarnego. Pamiętajmy również, że wartości mogą być ujemne, stąd należy również rozważyć kodowanie dla zmiennych o wartościach rzeczywistych. Ustalmy zatem, na ilu bitach chcemy kodować poszczególne parametry. Oznaczmy tę wartość przez  $L_1$ , wtedy rzeczywista wartość parametru będzie obliczana zgodnie ze wzorem (3).

$$x = a + \frac{b - a}{2^{L_1} - 1} \cdot U \quad (3)$$

W powyższym wzorze oznaczenia są następujące:  $a$  – jest to początek przedziału dla zadanego parametru,  $b$  – jest to koniec przedziału dla danego parametru, natomiast  $U$  – jest to dowolna wartość zakodowana przez  $L_1$  bitów w naturalnym kodzie binarnym. Jak zatem następuje krzyżowanie? Otóż dla każdej pary trenerów losujemy punkt przecięcia. Ten punkt przecięcia wyznacza nam dwie części naszego kodu. Oznacza to również, że powstaną juniorzy, którzy pierwszą część umiejętności przejmą od pierwszego trenera (albo od drugiego trenera), natomiast drugą część od pozostałego trenera. Aby to lepiej uzmysłowić, przedstawmy pewien przykład.

Niech pierwszy trener będzie opisany przez ciąg binarny 10010001, natomiast drugi 11100111. Niech punktem dzielącym będzie punkt o numerze czwartym, czyli 1001|0001 oraz 1110|0111. Stąd powstaną dwaj juniorzy o reprezentacjach 10010111 oraz 11100001. Przyjmijmy również, że każda z umiejętności jest kodowana na czterech bitach – stąd uzyskujemy juniorów o umiejętnościach (9,7) oraz (14,1).

Powstałą w ten sposób populację (która uzyskujemy po krzyżowaniu wszystkich par trenerów) uznajemy za nową populację – co oznacza, że juniorzy wchodzą w wiek seniorski i zastępują swoich mentorów w składzie drużyny. Ponownie obliczamy wartości sumaryczne umiejętności poszczególnych zawodników (nowych) i sprawdzamy warunek stopu. W momencie, w którym jest on nie spełniony, ponownie rozpoczynamy selekcję trenerów do populacji trenerskiej. Powtarzamy wszystkie kroki do momentu, w którym warunek stopu zostanie spełniony. Zależy nam bowiem, aby nasz zespół składał się z jak najlepszych zawodników.

## Implementacja SGA w języku Java

Pierwszym krokiem algorytmu jest wyznaczenie populacji osobników początkowych. Początkowo planowaliśmy wykonać losowanie wartości z wybranego zakresu, a następnie dokonać jej konwersji z wykorzystaniem `Long.toBinaryString(long)` do ciągu binarnego. Jednakże ze względu na obostrzenia, jakie są wprowadzane przez język Java, okazało się to niemożliwe. Powodem był brak sposobności konwersji wartości ujemnej do kodu binarnego (otrzymywaliśmy wyjątek o braku takiej możliwości). W związku z tym został napisany własny generator wartości binarnych. Pierwszym krokiem było wylosowanie liczby z zakresu dla każdego z parametrów, a następnie (jeżeli była ona ujemna, to poprzez `Math.abs(value)`) była ona przenoszona do wartości dodatnich. Na mocy wylosowanej liczby określaliśmy liczbę bitów, na której zostanie zapisany zadany parametr. Kolejnym krokiem było przygotowanie ciągu binarnego. Ustaliliśmy pewien próg, na mocy którego decydowaliśmy, czy wylosowaliśmy 0, czy też 1. Jeżeli wartość wylosowana była mniejsza od progu, wtedy do ciągu binarnego dodawaliśmy 0, w innym przypadku była to jedynka. Następnym elementem, który został wykonany, było obliczenie wartości funkcji przystosowania. Ze względu na to, że bazujemy na reprezentacji binarnej, musielibyśmy skorzystać ze wzoru (3) w celu uzyskania wartości dziesiętnych. Na tej podstawie przeprowadziliśmy kalkulację wartości funkcji przystosowania. Wykonanie tych elementów przedstawiono w Listingu 1.

**Listing 1. Losowanie populacji początkowej i utworzenie wartości dziesiętnej na podstawie ciągu binarnego**

```
private static Specimen getAnotherInitialElement(List<Specimen> initialPopulation, int spaceDimension, List<Double> beginEnds) {
    List<Double> attributes = new ArrayList<>();
    List<String> binaryAttributes = new ArrayList<>();
    for (int i = 0; i < spaceDimension * 2; i += 2) {
        Double start = Math.abs(beginEnds.get(i));
        Double end = Math.abs(beginEnds.get(i + 1));
        String str = Integer.toBinaryString(end > start ? end : intValue());
        String randBinary = getBinary(str.length());
        Double value = beginEnds.get(i) + (beginEnds.get(i + 1) - beginEnds.get(i)) * Long.parseLong(randBinary, 2) / (Math.pow(2, randBinary.length()) - 1);
        binaryAttributes.add(randBinary);
        attributes.add(value);
    }
    Specimen specimen = new Specimen();
    specimen.addAttribute(attributes);
    specimen.addBinaryAttributes(binaryAttributes);
    return specimen;
}

private static String getBinary(int bits) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < bits; i++) {
        int value = rnd.nextInt(64);
        sb.append(value < 32 ? "0" : "1");
    }
    return sb.toString();
}
```

Zgodnie z algorytmem kolejnym krokiem był wybór populacji rodzicielskiej, na podstawie której utworzona zostanie populacja potomna. W tym miejscu zastosowaliśmy algorytm SRS oraz algorytm koła ruletki. Zanim jednak to nastąpiło, przygotowaliśmy implementację skalowania funkcji przystosowania. Wykorzystaliśmy w tym celu wzory (1) oraz (2). Implementację skalowania funkcji przystosowania przedstawiamy w Listingu 2.

**Listing 2. Algorytm skalowania wartości funkcji przystosowania**

```
private static List<Double> getScaledFitnessValue(List<Double> fitnessValues, double k) {
    double avg = getAverageFitnessValue(fitnessValues);
    double max = getMaxFitnessValue(fitnessValues);
    double a = avg * (k - 1) / (max - avg);
    double b = avg * (1 - avg * (k - 1) / (max - avg));
    boolean scaledUnderZero = false;
    List<Double> scaledFitnessValues = new ArrayList<>();
    for (Double d : fitnessValues) {
        double result = a * d + b;
        if (result < 0) {
            scaledUnderZero = true;
            break;
        }
        scaledFitnessValues.add(result);
    }
    if (scaledUnderZero) {
        double min = getMinFitnessValue(fitnessValues);
        a = avg / (avg - min);
        b = -avg * min / (avg - min);
        scaledFitnessValues = new ArrayList<>();
        for (Double d : fitnessValues) {
            double result = a * d + b;
            scaledFitnessValues.add(result);
        }
    }
    return scaledFitnessValues;
}
```

Wracając do algorytmu SRS i koła ruletki, należy zauważyć, że drugi z wymienionych algorytmów był używany wyłącznie wtedy, gdy pierwszy z algorytmów nie zwrócił nam pełnej populacji rodzicielskiej. W tym celu sprawdzamy, czy liczba wylosowanych obiektów jest już wystarczająca (czy też nie). Dodatkowo należy zwrócić uwagę na to, że w tej procedurze obliczamy prawdopodobieństwo wystąpienia obiektu zarówno w „pierwszej rundzie”, w której pobieramy elementy na podstawie algorytmu SRS, jak i dla wystąpienia w „rundzie drugiej”, która jest warunkowana przez algorytm koła ruletki. Poniżej w Listingu 3 przedstawiamy zarówno sposób generacji kolejnej populacji, jak i sposób, w który były pobierane kolejne (brakujące) elementy do zbioru populacji rodzicielskiej.

**Listing 3. Sposób, w jaki utworzona została populacja rodziców**

```
public static List<Specimen> getAnotherPopulation(List<Specimen> currentPopulation, List<Double> fitnessValues, double k, List<Double> beginEnds) {
    List<Double> scaledFitnessValue = getScaledFitnessValue(fitnessValues, k);
    List<Double> possibilities = calculatePossibilities(scaledFitnessValue);
    List<Specimen> selectedParents = new ArrayList<>();
    List<Double> possibilitiesInSecondRound = new ArrayList<>();

    for (int i = 0; i < fitnessValues.size(); i++) {
        double numberOfWorkCopies = Math.floor(SPECIMENS_IN_POPULATION * possibilities.get(i));
        double possibilityForSecondRound = SPECIMENS_IN_POPULATION * possibilities.get(i) - numberOfWorkCopies;
        possibilitiesInSecondRound.add(possibilityForSecondRound);
        for (int j = 0; j < numberOfWorkCopies; j++) {
            selectedParents.add(currentPopulation.get(i));
        }
    }
    if (selectedParents.size() > SPECIMENS_IN_POPULATION) {
        throw new RuntimeException("New population cannot have more specimens than current population");
    }
    double possibilitiesSum = 0;
    for (int i = 0; i < possibilitiesInSecondRound.size(); i++) {
        possibilitiesSum += possibilitiesInSecondRound.get(i);
    }
    List<Double> possibilitiesInSecondRoundDiff = new ArrayList<>();
    for (Double d : possibilitiesInSecondRound) {
        possibilitiesInSecondRoundDiff.add(d / possibilitiesSum);
    }
}
```

```

int missingSpecimens = SPECIMENS_IN_POPULATION -
selectedParents.size();
selectedParents.addAll(getAdditionalSpecimens(currentPopulation, possibilitiesInSecondRoundDiff, missingSpecimens));

return crossingAndMutation(selectedParents,
currentPopulation, beginEnds);
}

private static List<Specimen> getAdditionalSpecimens(List<Specimen> currentPopulation, List<Double> possibilities, int specimensNumber) {
List<Specimen> selectedSpecimens = new ArrayList<>();
List<Double> thresholdForSpecimens = new ArrayList<>();
for (int i = 0; i < possibilities.size(); i++) {
if (possibilities.get(i) != 0) {
if (thresholdForSpecimens.isEmpty()) {
thresholdForSpecimens.add(possibilities.get(i) * 10);
} else {
thresholdForSpecimens.add(thresholdForSpecimens.get(thresholdForSpecimens.size() - 1) + possibilities.get(i) * 10);
}
}
for (int j = 0; j < specimensNumber; j++) {
double value = thresholdForSpecimens.get(thresholdForSpecimens.size() - 1) * rnd.nextDouble();
for (int p = 0; p < thresholdForSpecimens.size(); p++) {
if (value < thresholdForSpecimens.get(p)) {
selectedSpecimens.add(currentPopulation.get(p));
break;
}
}
}
return selectedSpecimens;
}

```

Zgodnie ze schematem przedstawionym na Rysunku 2 kolejnym krokiem było krzyżowanie i mutacja osobników znajdujących się w ramach populacji rodzicielskiej. Ze względu na to, że podczas krzyżowania powinniśmy łączyć osobniki w sposób losowy, wykonaliśmy przemieszanie kolekcji na podstawie funkcji `Collections.shuffle(List, Random)`. Przyjęliśmy również założenie, że jeżeli liczba osobników w ramach naszej populacji jest wartością nieparzystą, wtedy ostatni osobnik (po przemieszaniu kolekcji) jest automatycznie przepisywany do populacji rodzicielskiej. Krzyżowanie zostało wykonane na wartościach uzyskanych z użyciem kodu Graya. Ponownie napisaliśmy własną procedurę, którą przedstawiamy w Listingu 4, która zamieniała obiekt klasy `String` na również obiekt `String`, z tym że przechowywał on już wartość po zamianie do kodu Graya.

**Listing 4. Funkcja przeliczająca wartość ciągu binarnego do kodu Graya**

```

public static String encodeGray(String value) {
StringBuilder sb = new StringBuilder();
for(int i = 0; i < value.length(); i++) {
if(i == 0) {
char ch = value.charAt(0);
sb.append(ch == '0' ? "0" : "1");
} else {
char current = value.charAt(i);
char previous = value.charAt(i - 1);
sb.append(current == previous ? "0" : "1");
}
}
return sb.toString();
}

```

Kolejnym, naturalnym krokiem było przeprowadzenie procedury krzyżowania oraz mutacji. Oczywiście, zgodnie z opisem algorytmu SGA, krzyżowanie jest wykonywane z pewnym prawdopodobieństwem. Oznacza to, że do naszego algorytmu jako parametr należało przekazać informację o progu, który to umożliwiał przeprowadzenie krzyżowania. W ramach naszej implementacji uzna-

śmy, że przed wykonaniem krzyżowania należy wylosować liczbę ze zbioru od 0 do 1. Jeżeli zadana liczba jest mniejsza od progu krzyżowania, wtedy krzyżowanie następuje, w innym przypadku zwracane są dwa obiekty, które zostały przekazane do funkcji. Należy również wskazać, że w tym momencie badamy zależności pomiędzy długościami poszczególnych chromosomów. Jeżeli nie są one sobie równe, wtedy na początek krótszego z chromosomów dopisujemy wartości 0 (nie mają one wpływu na reprezentowaną przez chromosom liczbę). Różnica pomiędzy długościami chromosomów może wynikać z tego, że pierwszym krokiem, jaki wykonujemy w naszym algorytmie (w ramach procedury losowania osobników początkowych), jest losowanie liczb z podanego na wejściu zakresu. W takim przypadku możemy natrafić na liczbę, która będzie reprezentowana przez mniejszą liczbę bitów aniżeli jej odpowiedniczka z drugiego z wybranych chromosomów. Przykładem mogą być liczby 5 oraz 15 – odpowiednio będą one reprezentowane przez 101 oraz 1111.

Krzyżowanie jest wykonywane na populacji w sposób losowy. Co rozumiemy pod takim stwierdzeniem? Otóż należy zauważyć, że w ramach samej procedury wykonyjącej krzyżowanie oraz mutację pierwszym krokiem jest przemieszanie zbioru wejściowego. Jest to wykonywane w celu zapewnienia pewnej losowości połączeń obiektów w ramach populacji. Ten krok był niezbytny, albowiem po przemieszaniu zbiorów wykonujemy łączenie w pary. Aby uprościć implementację, łączenie w pary polega na połączeniu ze sobą dwóch sąsiednich obiektów w populacji (dlatego wcześniej dokonaliśmy przemieszania zbiorów). Po wybraniu dwóch obiektów do procedury krzyżowania postępujemy zgodnie z metodą krzyżowania jednopunktowego – mianowicie losujemy jeden punkt i na jego podstawie dzielimy chromosomy na dwie części. Potomkowie otrzymują jedną część od jednego z rodziców, a drugą od drugiego.

Następną procedurą, która została przez nas wykonana, była procedura mutacji. Mutacja jest wykonywana na osobnikach po krzyżowaniu. Polega ona na przeskanowaniu całego chromosomu, bit po bicie, i losowaniu wartości z przedziału od 0 do 1 przy każdym kolejnym bicie. Gdy wylosowana wartość jest mniejsza od obranego progu mutacji, wtedy następuje zamiana wartości bitu na przeciwną (z 0 na 1 i z 1 na 0). W Listingu 5 przedstawiamy kod źródłowy procedur wykonyujących krzyżowanie i mutację.

**Listing 5. Funkcje odpowiedzialne za wykonanie procedury krzyżowania oraz mutacji**

```

private static List<Specimen> crossingAndMutation(List<Specimen> parentPopulation, List<Specimen> currentPopulation, List<Double> beginEnds) {
long seed = System.nanoTime();
Collections.shuffle(parentPopulation, new Random(seed));

List<Specimen> rebuiledSpecimens = new ArrayList<>();
if (parentPopulation.size() % 2 == 1) {
rebuiledSpecimens.add(parentPopulation.get(parentPopulation.size() - 1));
}

for (int i = 0; i < parentPopulation.size() - 1; i += 2) {
Specimen one = parentPopulation.get(i);
Specimen two = parentPopulation.get(i + 1);
String oneRepresentation = "";
String twoRepresentation = "";
List<Double> attributesSizes = new ArrayList<>();
for (int j = 0; j < one.getSpecimenAttributes().size(); j++) {
String oneAttr = one.getBinaryAttributes().get(j);
String twoAttr = two.getBinaryAttributes().get(j);
String oneAttrAsGC = GrayCode.encodeGray(oneAttr);
String twoAttrAsGC = GrayCode.encodeGray(twoAttr);
}
}
}

```

```

    if (oneAttrAsGC.length() > twoAttrAsGC.length()) {
        int diff = oneAttrAsGC.length() - twoAttrAsGC.length();
        String addedEmpty = "";
        for (int p = 0; p < diff; p++) {
            addedEmpty = addedEmpty.concat("0");
        }
        twoAttrAsGC = addedEmpty.concat(twoAttrAsGC);
        attributesSizes.add((double) twoAttrAsGC.length());
    } else if (twoAttrAsGC.length() > oneAttrAsGC.length()) {
        int diff = twoAttrAsGC.length() - oneAttrAsGC.length();
        String addedEmpty = "";
        for (int p = 0; p < diff; p++) {
            addedEmpty = addedEmpty.concat("0");
        }
        oneAttrAsGC = addedEmpty.concat(oneAttrAsGC);
        attributesSizes.add((double) oneAttrAsGC.length());
    } else {
        attributesSizes.add((double) oneAttrAsGC.length());
    }
    oneRepresentation = oneRepresentation.
    concat(oneAttrAsGC);
    twoRepresentation = twoRepresentation.
    concat(twoAttrAsGC);
}
List<String> crossedSpecimens =
getCrossing(oneRepresentation, twoRepresentation);
List<String> specimensAfterMutation =
getMutation(crossedSpecimens);
rebuiltedSpecimens.addAll(rebuildSpecimens(specimensAfterMutation,
    attributesSizes, beginEnds));
}
if (ELITE_SUCCESSION_MODEL) {
    List<Double> parentsPopulationFitnessValue = calculatePopulationFitnessValue(currentPopulation, SELECTED_FUNCTION);
    int index = 0;
    double max = Double.MIN_VALUE;
    for (int i = 0; i < parentsPopulationFitnessValue.size(); i++) {
        if (parentsPopulationFitnessValue.get(i) > max) {
            max = parentsPopulationFitnessValue.get(i);
            index = i;
        }
    }
    int indexMin = 0;
    double min = Double.MAX_VALUE;
    List<Double> rebuiltedSpecimensFitnessValue = calculatePopulationFitnessValue(rebuiltedSpecimens, SELECTED_FUNCTION);
    for (int i = 0; i < rebuiltedSpecimensFitnessValue.size(); i++) {
        if (rebuiltedSpecimensFitnessValue.get(i) < min) {
            min = rebuiltedSpecimensFitnessValue.get(i);
            indexMin = i;
        }
    }
    if (max > min) {
        rebuiltedSpecimens.remove(indexMin);
        rebuiltedSpecimens.add(currentPopulation.get(index));
    }
}
return rebuiltedSpecimens;
}

private static List<String> getMutation(List<String> representatives) {
    List<String> specimensAfterMutation = new ArrayList<>();
    for (String representative : representatives) {
        StringBuilder rep = new StringBuilder(representative);
        for (int i = 0; i < representative.length(); i++) {
            double possibility = rnd.nextDouble();
            if (possibility < POSSIBILITY_OF_MUTATION) {
                char c = representative.charAt(i);
                rep.setCharAt(i, c == '1' ? '0' : '1');
            }
        }
        specimensAfterMutation.add(rep.toString());
    }
    return specimensAfterMutation;
}

private static List<String> getCrossing(String oneRepresentation, String twoRepresentation) {
    double d = rnd.nextDouble();
    List<String> crossedSpecimens = new ArrayList<>();
    if (d < CROSSOVER POSSIBILITY) {
        int crossingPosition = rnd.nextInt(oneRepresentation.

```

```

length() - 2);
        String oneAfterCrossing = oneRepresentation.
        substring(0, crossingPosition).concat(twoRepresentation.
        substring(crossingPosition, twoRepresentation.length()));
        String twoAfterCrossing = twoRepresentation.
        substring(0, crossingPosition).concat(oneRepresentation.
        substring(crossingPosition, oneRepresentation.length()));
        crossedSpecimens.add(oneAfterCrossing);
        crossedSpecimens.add(twoAfterCrossing);
    } else {
        crossedSpecimens.add(oneRepresentation);
        crossedSpecimens.add(twoRepresentation);
    }
    return crossedSpecimens;
}

```

Zanim opiszemy, w jaki sposób wykonywane są następne kroki, musimy na moment zatrzymać się przy procedurze przedstawionej w Listingu 5. Otóż w ramach metody *crossingAndMutation* jest również zaimplementowany model sukcesji elitarnej. Model ten jest odpowiedzialny za kopiowanie najlepszego osobnika z populacji rodzicielskiej do populacji potomnej (w miejscu najlepszego osobnika) pod warunkiem, że najlepszy osobnik z populacji potomnej charakteryzuje się gorszym przystosowaniem od najlepszego osobnika wśród rodziców. Ten element został przez nas zaimplementowany w celu zweryfikowania, czy może mieć on realny wpływ na wynik końcowy. Co więcej, aby móc go uruchomić, należy ustawić flagę *ELITE\_SUCCESSION\_MODEL*. Słownem wyjaśnienia należy także dodać, że wspomniany model sukcesji nie dotyczy większej liczby obiektów (a tylko pojedynczego), gdyż jego zastosowanie może prowadzić do przedwczesnej zbieżności do minimum lokalnego.

Ostatnim etapem w procedurze tworzenia nowej populacji jest odbudowanie osobników po zakończeniu krzyżowania i mutacji. Poprzez odbudowanie rozumiemy w tym miejscu możliwość wyznaczenia wartości liczbowej (w systemie dziesiętnym), która będzie użyta podczas obliczania wartości funkcji przystosowania. Korzystamy w tym miejscu z dwóch dodatkowych elementów – pierwszego, którym jest dekodowanie wartości z kodu Graya do NKB, a następnie przeliczenie z NKB do wartości typu long, oraz drugiego, którym jest przeliczenie uzyskanej wartości do zbioru liczb rzeczywistych (zgodnie z treścią zadania). Kod wykonujący procedurę odbudowy przedstawiamy w Listingu 6.

**Listing 6.** Kod wykonujący odbudowę obiektu (przeniesienie do pożądanego obszaru poszukiwań)

```

private static List<Specimen> rebuildSpecimens(List<String>
representatives, List<Double> attributesSizes, List<Double>
beginEnds) {
    List<Specimen> createdSpecimens = new ArrayList<>();
    List<Double> thresholds = new ArrayList<>();
    for (int i = 0; i < attributesSizes.size(); i++) {
        thresholds.add(thresholds.isEmpty() ? attributesSizes.
        get(i) : attributesSizes.get(i) + thresholds.
        get(thresholds.size() - 1));
    }
    for (String str : representatives) {
        List<Double> params = new ArrayList<>();
        List<String> binaryRepresentation = new ArrayList<>();
        for (int j = 0; j < thresholds.size(); j++) {
            if (j == 0) {
                int threshold = thresholds.get(j).intValue();
                long value = Long.parseLong(str.substring(0,
                threshold), 2);
                binaryRepresentation.add(str.substring(0, threshold));
                params.add(getRebuiltedValue(GrayCode.
                decodeGray(value), beginEnds.get(j * 2), beginEnds.
                get(j * 2 + 1), thresholds.get(j).intValue()));
            } else {

```

```

        long value = Long.parseLong(str.substring(thresholds.
get(j - 1).intValue(), thresholds.get(j).intValue()), 2);
        binaryRepresentation.add(str.substring(thresholds.get(j -
1).intValue(), thresholds.get(j).intValue())));
        params.add(getRebuiltedValue(GrayCode.
decodeGray(value), beginEnds.get(j * 2), beginEnds.
get(j * 2 + 1), thresholds.get(j).intValue())));
    }
}
Specimen spec = new Specimen();
spec.addAttribute(params);
spec.addBinaryAttributes(binaryRepresentation);
createdSpecimens.add(spec);
}
return createdSpecimens;
}

private static double getRebuiltedValue(long value, double
begin, double end, int attributeSize) {
    return begin + (end - begin) * value / (Math.pow(2,
attributeSize) - 1);
}

```

Na zakończenie opisu Standardowego Algorytmu Genetycznego (SGA) chcielibyśmy przedstawić procedurę odpowiedzialną za wykonanie całego algorytmu. W ramach niej wykonywane są wszystkie niezbędne kroki, które pozwalają na uzyskanie rezultatów na podstawie wykonywanych obliczeń ewolucyjnych. TIME\_TO\_LIVE, który pojawia się w ramach kodu, oznacza dodatkowy parametr odpowiedzialny za wskazanie, jaką liczbę iteracji będziemy wykonywać w ramach naszego programu. Dodatkowo sprawdzamy, czy populacja nie składa się z dokładnie takich samych obiektów, a w pętli for odpowiedzialnej za porównanie wyników z bieżącą populacją z wynikami najlepszymi sprawdzamy, czy badane jest minimum, czy też maksimum. W Listingu 7 przedstawiono postać wspomnianej metody głównej.

**Listing 7. Metoda główna odpowiedzialna za uruchomienie obliczeń ewolucyjnych skorelowanych ze Standardowym Algorytmem Genetycznym (SGA)**

```

public static void startStandardAlgorithm(List<Double>
beginEnds) {
    int iterationsLastChange = 0;
    int currentNumberOfIterations = 0;
    List<Specimen> initialPopulation =
    getInitialPopulation(SPECIMENS_IN_POPULATION, NUMBER_OF_
    PARAMETERS, beginEnds);
    List<Double> calculatedFitnessValues = calculatePopulationFit-
    nessValue(initialPopulation, SELECTED_FUNCTION);
    while (currentNumberOfIterations < TIME_TO_LIVE) {
        initialPopulation = getAnotherPopulation(initialPopulation,
        calculatedFitnessValues, K_PARAMETER, beginEnds);
        calculatedFitnessValues = calculatePopulationFitnessValue(in-
        itialPopulation, SELECTED_FUNCTION);
        if(checkIfTheSame(calculatedFitnessValues)){
            break;
        }
        Boolean changeFlag = false;
        for (int i = 0; i < calculatedFitnessValues.size(); i++) {
            if (MAXIMUM) {
                if (calculatedFitnessValues.get(i) > CURRENT_FITNESS_
                VALUE_BEST) { //we are looking for maximum
                    CURRENT_FITNESS_VALUE_BEST = calculatedFitnessValues.
                    get(i);
                    CURRENT_BEST = initialPopulation.get(i);
                    iterationsLastChange = 0;
                    changeFlag = true;
                }
            } else {
                if (calculatedFitnessValues.get(i) < CURRENT_FITNESS_
                VALUE_BEST) { //we are looking for minimum
                    CURRENT_FITNESS_VALUE_BEST = calculatedFitnessValues.
                    get(i);
                    CURRENT_BEST = initialPopulation.get(i);
                    iterationsLastChange = 0;
                    changeFlag = true;
                }
            }
        }
    }
}

```

```

        }
    }
    if (!changeFlag) {
        ++iterationsLastChange;
    }
    ++currentNumberOfIterations;
}
System.out.println(" " + CURRENT_FITNESS_VALUE_BEST);
}

```

## ALGORYTM MICHALEWICZA

W przypadku Standardowego Algorytmu Genetycznego mieliśmy do czynienia z reprezentacją binarną wartości poszczególnych osobników. Jednakże został zaproponowany algorytm, który wykorzystuje zmiennoprzecinkową reprezentację danych. Oczywiście uważny czytelnik mógłby powiedzieć, że skoro zmieniamy reprezentację, to również należy przemyśleć, jak powinny działać metody krzyżowania oraz mutacji. W tym przypadku jest to jak najbardziej uzasadnione, albowiem standardowe metody krzyżowania i mutacji nie są adekwatne do wartości zmiennoprzecinkowych.

Aby uzmysłowić czytelnikowi, jak wygląda taka reprezentacja, raz jeszcze skupmy się na zawodniku z naszej drużyny. Niech będzie on ponownie opisany przez dwa parametry ( $x, y$ ) oraz niech te dwa atrybuty przyjmują następujące wartości: (15,6; 22,3). Jest to koronny przykład reprezentacji zmiennoprzecinkowej.

W ramach niniejszego podrozdziału opiszymy, w jaki sposób przeprowadzana jest procedura krzyżowania oraz mutacji. Pozostałe kroki są w przypadku tego algorytmu analogiczne jak dla algorytmu SGA.

### Operator mutacji nierównomiernej

Jest to operator, który bierze pod uwagę dwa parametry, bieżącą iterację oraz całkowitą liczbę iteracji. Pierwszy z wymienionych jest oznaczany literą  $t$ , natomiast drugi oznaczamy literą  $T$ . Nowa wartość parametru  $x$  naszego osobnika będzie wyznaczana na mocy wzoru (4).

$$x = \begin{cases} x + \Delta(t, b_i - x) \text{ z prawdopodobieństwem } \frac{1}{2} \\ x - \Delta(t, x - a_i) \text{ z prawdopodobieństwem } \frac{1}{2} \end{cases} \quad (4)$$

Oczywiście należy wyjaśnić, czym jest  $a_i$ ,  $b_i$ ,  $\Delta$ , które pojawiły się w ramach wzoru (4). Pierwszy z wymienionych parametrów oznacza początek i-tego przedziału, czyli początek przedziału dla i-tego atrybutu. Z kolei drugi z wymienionych oznacza koniec przedziału dla i-tego atrybutu (w naszym przypadku  $x$  jest pierwszym atrybutem, zatem  $i = 1$ ). Delta jest to kolejna funkcja, którą opisujemy w ramach (5).

$$\Delta(t, m) = m \cdot \left(1 - r^{(1-\frac{t}{T})c}\right) \quad (5)$$

W przypadku wzoru powyższego  $c$  jest stałą i jest równa 2, natomiast  $r$  to liczba losowa z przedziału od 0 do 1. Jak zatem wygląda procedura postępowania przy tej mutacji? Otóż losujemy prawdopodobieństwo wykorzystania danego wzoru. Jeżeli jest ono mniejsze bądź równe 0,5, skorzystamy z pierwszej z przedstawionych formuł, w innym przypadku wykorzystamy drugą wartość. Następnie obliczamy wartość na podstawie wybranego wzoru i podstawiamy ją jako wartość  $x$ .

## Operator krzyżowania arytmetycznego

Ponownie jest to operator specyficzny tylko i wyłącznie dla algorytmów operujących na wartościach zmiennoprzecinkowych. Mianowicie opieramy się ponownie o wykorzystanie pewnych wzorów. Przyjmijmy raz jeszcze, że mamy dwóch osobników (czy też trenerów), pierwszy z nich jest opisany wartościami  $(x_1, y_1)$ , natomiast drugi reprezentują wartości  $(x_2, y_2)$ . Oznaczmy również przykładowych juniorów (następów trenerów) jako osobników opisanych wartościami  $(x_3, y_3)$ ;  $(x_4, y_4)$ . Na przykładzie cechy  $x$  opiszemy, jak zachowuje się operator krzyżowania arytmetycznego. Nasze rozważania są zaprezentowane w ramach (6).

$$\begin{cases} x_3 = \alpha \cdot x_1 + (1 - \alpha) \cdot x_2 \\ x_4 = (1 - \alpha) \cdot x_1 + \alpha \cdot x_2 \end{cases} \quad (6)$$

Zauważmy zatem, że wartości parametrów bazują bezpośrednio na wartościach rodziców (trenerów), aczkolwiek są również zależne od losowej wartości  $\alpha$ , która przynależy do zakresu  $[0,1]$ .

## Implementacja algorytmu Michalewicza w języku Java

Zgodnie z informacjami, które zostały zaprezentowane powyżej, algorytm Michalewicza wprowadza jedynie pewne zmiany, które pozwalają na prawidłową realizację algorytmu ewolucyjnego dla osobników reprezentowanych w postaci ciągów liczb zmiennoprzecinkowych. Jak zostało już wspomniane, tymi elementami są algorytmy krzyżowania arytmetycznego (zastosowany przez nas algorytm Whole Arhythmetic Crossover) oraz algorytm mutacji nierównomiernej. Aby uniknąć duplikowania kodu, który jest przedstawiany w ramach tego artykułu, prezentujemy jedynie dwie metody, które stanowią główną oś algorytmu Michalewicza – są nimi wspomniane: algorytm krzyżowania arytmetycznego oraz algorytm mutacji nierównomiernej. W Listingu 8 prezentujemy algorytm krzyżowania arytmetycznego, natomiast Listing 9 zawiera informacje o implementacji algorytmu mutacji nierównomiernej.

### Listing 8. Algorytm krzyżowania arytmetycznego

```
private static List<MichalewiczSpecimen> crossingTwoMichalewiczSpecimens(MichalewiczSpecimen one, MichalewiczSpecimen two) {
    List<MichalewiczSpecimen> crossingSpecimens = new ArrayList<>();
    double alfa = ALFA;
    List<Double> paramsSpecimenOne = new ArrayList<>();
    List<Double> paramsSpecimenTwo = new ArrayList<>();
    for (int i = 0; i < one.getAttrbitues().size(); i++) {
        if (RANDOM_ALFA) {
            Random rnd = new Random();
            rnd.setSeed(System.nanoTime());
            alfa = rnd.nextDouble();
        }
        double paramOne = alfa * one.getAttrbitues().get(i) + (1 - alfa) * two.getAttrbitues().get(i);
        double paramTwo = (1 - alfa) * one.getAttrbitues().get(i) + alfa * two.getAttrbitues().get(i);
        paramsSpecimenOne.add(paramOne);
        paramsSpecimenTwo.add(paramTwo);
    }
    MichalewiczSpecimen michalewiczOne = new MichalewiczSpecimen();
    MichalewiczSpecimen michalewiczTwo = new MichalewiczSpecimen();
    michalewiczOne.addAttribute(paramsSpecimenOne);
    michalewiczTwo.addAttribute(paramsSpecimenTwo);

    crossingSpecimens.add(michalewiczOne);
    crossingSpecimens.add(michalewiczTwo);
    return crossingSpecimens;
}
```

Słowem wyjaśnienia należy dodać, że w ramach naszego programu możemy ustawić parametr w dwojak sposob – albo poprzez ustawienie flagi RANDOM\_ALFA na wartość false i wpisanie ręcznie wartości do parametru ALFA (w pliku), albo poprzez losowanie przy każdym krzyżowaniu (wtedy flaga RANDOM\_ALFA musi przyjąć wartość true).

### Listing 9. Algorytm mutacji nierównomiernej

```
private static List<MichalewiczSpecimen> mutationOfAllMichalewiczSpecimens(List<MichalewiczSpecimen> michalewiczSpecimens,
List<Double> beginEnds, double currentIteration) {
    List<MichalewiczSpecimen> mutationSpecimens = new ArrayList<>();
    for (MichalewiczSpecimen specimen : michalewiczSpecimens) {
        List<Double> paramsAfterMutation = new ArrayList<>();
        for (int i = 0; i < specimen.getAttrbitues().size(); i++) {
            Random rnd = new Random();
            rnd.setSeed(System.nanoTime());
            double value = rnd.nextDouble();
            double calculatedMutationValue = 0;
            if (value <= 0.5) {
                double end = beginEnds.get(2 * i + 1);
                calculatedMutationValue = specimen.getAttrbitues().get(i) + calculateUnevenMutation(end - specimen.getAttrbitues().get(i), currentIteration);
            } else {
                double begin = beginEnds.get(2 * i);
                calculatedMutationValue = specimen.getAttrbitues().get(i) - calculateUnevenMutation(specimen.getAttrbitues().get(i) - begin, currentIteration);
            }
            paramsAfterMutation.add(calculatedMutationValue);
        }
        mutationSpecimens.add(new MichalewiczSpecimen(paramsAfterMutation));
    }
    return mutationSpecimens;
}

private static double calculateUnevenMutation(double y, double currentIteration) {
    Random rnd = new Random();
    rnd.setSeed(System.nanoTime());
    double r = rnd.nextDouble();
    double pow = (1 - currentIteration / (double) TIME_TO_LIVE) * 2;
    double bracket = 1 - Math.pow(r, pow);
    return y * bracket;
}
```

Jako wyjaśnienie należy dodać, że obliczenie mutacji nierównomiernej odbywa się na podstawie numeru bieżącej iteracji oraz ogólnej liczby iteracji, która ma zostać wykonana.

## PORÓWNANIE ALGORYTMÓW

W ostatniej części niniejszego artykułu chcielibyśmy przedstawić porównanie wyników uzyskiwanych dla obydwu algorytmów. W tym przypadku wykorzystamy dwie przykładowe funkcje, którymi są funkcja Rastrigina (w niej skupimy się na czasie wykonania 100 eksperymentów dla różnej liczby iteracji) oraz funkcja Griewanka – w niej z kolei odniesiemy się do uzyskiwanych wyników.

Niech zatem funkcja Rastrigina będzie funkcją dwuparametrową  $x$  oraz  $y$ , dla których przedziały są określone od -20 do 100 dla  $x$  oraz od -60 do 200 dla  $y$ , prawdopodobieństwo mutacji niech wynosi 0.01, a krzyżowania 0.6. Parametr K niech zatem wynosi 3, a parametr A funkcji Rastrigina będzie równy 5. W ramach Tabeli 1 oraz 2 przedstawiamy rezultaty naszych eksperymentów.

Pierwszym elementem, do którego chcielibyśmy się odnieść, jest czas, jaki został poświęcony na wykonanie 100 eksperymentów. Aby móc to zaobserwować, prezentujemy Rysunek 3, na któ-

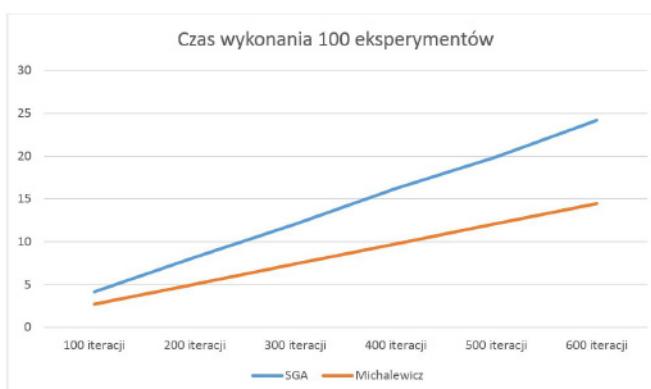
Badane parametry\Liczba iteracji	100	200	300	400	500	600
Czas	4,15 sekundy	8,175 sekundy	12,085 sekundy	16,238 sekundy	19,899 sekundy	24,152 sekundy
Najmniejsza wartość funkcji przystosowania	3361,328269317657	3361,6865725547705	3361,6865725547705	3361,328269317657	3361,328269317657	3361,328269317657
Odnalezione najlepszy obiekt	x = -0,15748031496062964 y = -57,97662282174139	x = 0,7874015748031482 y = -57,97662282174139	x = 0,7874015748031482 y = -57,97662282174139	x = -0,15748031496062964 y = -57,97662282174139	x = -0,15748031496062964 y = -57,97662282174139	x = -0,15748031496062964 y = -57,97662282174139

Tabela numer 1. Wyniki dla algorytmu SGA

Badane parametry\Liczba iteracji	100	200	300	400	500	600
Czas	2,707 sekundy	5,07 sekundy	7,405 sekundy	9,775 sekundy	12,116 sekundy	14,422 sekundy
Najmniejsza wartość funkcji przystosowania	0,007320126009734196	0,015023429767028418	0,01436488215878029	0,012013880276125377	0,00200965248760987	0,05983152155496718
Odnaleziony najlepszy obiekt	x = 0,0634943140829165 y = -0,05545219593221873	x = -0,09684634711552675 y = 0,07215073627357071	x = 0,11779752005549682 y = -0,008328485844650402	x = 0,10488901493082636 y = 0,025720201742416293	x = 0,018437149696946875 y = -0,04013811494558439	x = -0,05552282628844374 y = 0,052208599993839755

Tabela numer 2. Wyniki dla algorytmu Michalewicza

rym przedstawiono w sposób graficzny czas, jaki był potrzebny na realizację poszczególnych 100 eksperymentów.



Rysunek 3. Wykres zależności czasu wykonania poszczególnych algorytmów od liczby iteracji (czas dla 100 eksperymentów)

Zauważmy, że algorytm Michalewicza za każdym razem potrzebował mniej czasu na wykonanie 100 eksperymentów (co również oznacza, że średni czas wykonania pojedynczego eksperymentu jest mniejszy dla tego rodzaju algorytmu aniżeli analogiczny czas dla algorytmu SGA). Zauważmy również, że wyniki, które zostały osiągnięte z użyciem algorytmu Michalewicza, są daleko bardziej zadowalające – gdyż są one zbliżone do optimum globalnego, jakim jest punkt 0.0 dla tak określonej funkcji Rastrigina. Musimy także skonkludować, że dla algorytmu SGA wszystkie uruchomienia znalazły pewne optimum lokalne. Jest to jedna z wad algorytmu standardowego. Ponadto zbyt restrykcyjna reprezentacja binarna może powodować znaczące różnice w rezultatach. Kolejną kwestią, którą należy podkreślić, jest to, że nie obserwujemy bezpośredniego wpływu pomiędzy liczbą iteracji a jakością najlepszego rezultatu. Zwróćmy uwagę na to, że jest to wniosek prawdziwy zarówno dla algorytmu standardowego, jak i algorytmu Michalewicza. Na podstawie pierwszego eksperymentu jesteśmy zmuszeni stwierdzić, że w zadanym przedziale znacznie lepiej poradził sobie algorytm Michalewicza.

Badane parametry\Liczba iteracji	100	200	300	400
MBF (SGA)	8,310773	8,309762	8,309076	8,307915
MBF (Michalewicz)	0,034461	0,023452	0,02189	0,018227

Tabela 3. Wyniki dla pierwszego eksperymentu

Badane parametry\Liczba iteracji	100	200	300	400
MBF (SGA)	44,99688	44,99017	44,98679	44,9858
MBF (Michalewicz)	0,075919	0,052704	0,038264	0,034158

Tabela 4. Wyniki dla drugiego eksperymentu

Jako drugi przykład weźmy zatem funkcję Griewanka, dla której wykonaliśmy dwa eksperymenty. Pierwszy z nich został przygotowany dla czterech atrybutów, z których każdy jest określony na przedziale od -100 do 100, natomiast w ramach drugiego testu operujemy na znacznie szerszym zakresie dla każdego z parametrów, jakim jest od -300 do 300 (3 parametry). W Tabeli 3 przedstawiamy wyniki dla pierwszego testu, natomiast w Tabeli 4 zaprezentowano wyniki eksperymentu drugiego. W obu tabelach pojawia się wartość MBF, co oznacza Mean Best Fitness (Średnie najlepsze przystosowanie).

Zauważmy zatem, że również w tym przypadku algorytm Michalewicza zwraca osobnika o znacznie lepszym przystosowaniu. Jak zostało już wspomniane, algorytm SGA jest algorymem zbyt restrykcyjnym ze względu na zastosowaną reprezentację binarną.

## PODSUMOWANIE

W ramach niniejszego artykułu zostały przedstawione podstawowe informacje dotyczące algorytmów ewolucyjnych. Tak napraw-

dę stanowią one ciekawą alternatywę dla sztucznych sieci neuronowych. Zapewne byłyby one znacznie szerzej wykorzystywane, gdyby nie miały losowej natury. Wpływa ona bowiem znaczco na wynik, co z kolei powoduje, że dany algorytm należy wykonać wielekrotnie w celu uzyskania realnego wyniku.

W artykule przedstawione zostały również dwa algorytmy ewolucyjne, czyli SGA oraz algorytm Michalewicza. Jak zostało już podkreślone w części dotyczącej porównania algorytmów, algorytm Michalewicza zwraca znacznie lepsze rezultaty. Jest to powodowane przede wszystkim tym, że SGA posiada bardzo restrykcyjną reprezentację binarną i jej konwersja do przedziału liczb rzeczywistych jest dosyć trudna, natomiast algorytm Michalewicza bazuje na wartościach zmiennoprzecinkowych (co jest dla niego naturalne).

Zachęcam czytelników do samodzielnego zgłębiania zagadnień związanych z algorytmami ewolucyjnymi. Są one interesujące i mogą znaleźć zastosowanie w wielu dziedzinach informatyki (również w Biometrii).



**MACIEJ SZYMKOWSKI**

[maciej.szymkowski@symmetra.pl](mailto:maciej.szymkowski@symmetra.pl)

Inżynier Informatyk, Junior Software Developer w Symmetra Sp. z o. o., student informatyki (II stopień) na Wydziale Informatyki Politechniki Białostockiej. Do swoich zainteresowań zalicza: biometrię, przetwarzanie i analizę obrazów oraz sygnałów, elektronikę, architekturę komputerów oraz zagadnienia bezpieczeństwa komputerowego. Uwielbia poszerzać swoją wiedzę poprzez uczestnictwo w konferencjach oraz lekturę książek i artykułów. Programuje w takich językach jak: Java, C#, C++, C oraz Python.

reklama



**devstyle.pl**

ŚWIAT OKIEM PROGRAMISTY

# Konteneryzacja z Rancherem

W dobie popularności konteneryzacji warto zastanowić się, jakie rozwiązanie wybrać do zarządzania mikroserwisami na małą i średnią skalę. Poznajmy Ranchera – proste narzędzie do zarządzania kontenerami – oraz szybki sposób jego instalacji za pomocą Ansible.

## CZYM JEST RANCHER?

Rancher to darmowe, otwarte oprogramowanie rozwijane przez Rancher Labs przy udziale społeczności. Pozwala on na zarządzanie ekosystemem aplikacji skonteneryzowanych. Pośród innych rozwiązań w swojej klasie wyróżnia go wsparcie dla uruchamiania w środowiskach hybrydowych. Rancher w wersji pierwszej wspiera trzy silniki konteneryzacyjne: Docker Swarm, Kubernetes oraz Cattle. Wersja 2.0 Ranchera, której produkcyjne wydanie zapowiadano w planie projektu [1] na połowę pierwszego kwartału 2018 roku, ma opierać się w całości na Kuberentesies.

Dlaczego warto wypróbować Ranchera? Jego główną zaletą jest elastyczność. Pozwala on na przeprowadzenie lokalnej instalacji na dowolnej maszynie posiadającej Dockera. Został on zaprojektowany tak, aby wszystkie potrzebne mu usługi uruchamiane były w kontenerach.

Instalacja Ranchera może zostać wykonana na kilka sposobów – ze względu na oczekiwany dostępność usługi. Najważniejsze warianty to:

- » Minimalna instalacja – pozwala na uruchomienie Ranchera na jednym bądź dwóch hostach spełniających wymagania,
- » Instalacja bez wysokiej dostępności – jedna maszyna przeznaczona dla węzła zarządzającego, baza danych węzła zarządzającego uruchomiona w kontenerze lub na zewnętrznej maszynie oraz dodatkowe maszyny – agenci uruchamiający kontenery aplikacyjne,
- » Instalacja z wysoką dostępnością – kilka węzłów zarządzających, klaster bazy danych MySQL oraz węzły dla agentów.

Wszystkie warianty z opisami manualnej instalacji możemy znaleźć w dokumentacji [2].

Na potrzeby tego artykułu Ranchera zainstalujemy z minimalną ilością kroków manualnych. W tym celu wykorzystamy narzędzie Ansible.

## ANSIBLE

Ansible to narzędzie służące do automatycznego konfigurowania hostów i wydawania oprogramowania. Podobnie jak Rancher jest to otwarte oprogramowanie. Jest rozwijany przez społeczność przy wsparciu firmy Red Hat.

Istotną zaletą Ansible jest możliwość działania bez udziału dodatkowych serwerów. Jedynym wymaganiem do rozpoczęcia konfiguracji hostów za jego pomocą jest dostęp SSH oraz uprawnienia administratora.

Do realizacji zadań konfiguracyjnych i deploymentowych wykorzystujemy opisy w formacie YAML. Dzięki zastosowaniu tego formatu każdy deployment jest bardziej opisem efektu niż definicją sposobu jego osiągnięcia. W terminologii Ansible nazywane są one playbookami.

Chcąc lepiej zrozumieć zawartość playbooka, warto poznać nieco terminologii:

- » Moduł – rodzaj wtyczki, dzięki której Ansible wie, w jaki sposób wykonać opisane zadanie,
- » Vault – funkcjonalność Ansible pozwalająca na przechowywanie plików yaml z sekretami w postaci zaszyfrowanej,
- » Task – zbiór konfiguracji modułów składających się na realizację większego zadania, takiego jak instalacja Dockera,
- » Inventory – opis maszyn oraz ich przypisania do grup.

Ze względu na to, że Ansible po stronie operatora jedynie interpretuje pliki yaml i wykonuje odpowiednie polecenia na zdalnych serwerach, należy przestrzegać zasad budowania struktury katalogów. Jej opis wraz z innymi dobrymi praktykami tworzenia playbooków można znaleźć na stronie *playbooks best practices* [3] projektu Ansible.

## MINIMALNY PLAYBOOK ANSIBLE

Przed przystąpieniem do analizy przygotowanego na potrzeby artykułu playbooka instalującego Ranchera napiszemy prosty playbook na rozgrzewkę. Utworzenie minimalnego przykładu wymaga stworzenia dwóch plików:

- » Inventory – spisu hostów, na których wykonamy playbook,
- » Treści playbooka w pliku yaml – opisu czynności, które wykona dla nas Ansible.

Na potrzeby przykładu uruchomiłem lokalną maszynę wirtualną z systemem operacyjnym Ubuntu 17.10.1 w wersji serwer. Podczas instalacji utworzyłem użytkownika o danych logowania user/user, zainstalowałem serwer ssh i przesłałem użytkownikowi klucz publiczny. Sieć maszyny wirtualnej została skonfigurowana w trybie NAT z przekierowaniem portu gospodarza 2222 na port gościa 22. Domyślną wersję Pythona dla Ubuntu 17.10.1 jest Python 3, więc konieczne będzie poinformowanie Ansible, że z tej wersji należy korzystać.

Wszystkie wymienione wyżej fakty zostały odzwierciedlone w jednoliniowym pliku *inventory.ini*:

### Listing 1. Plik inventory.ini

```
localhost:2222 ansible_python_interpreter=/usr/bin/python3
ansible_user=user
```

Połowa drogi za nami, kolejnym krokiem będzie utworzenie playbooka. Nasz przykładowy playbook wykona dla nas trzy zadania: utworzy plik /tmp/hello na maszynie zdalnej, wylistuje zawartość katalogu /tmp/ oraz wydrukuje tę zawartość na ekranie.



Zapraszamy na autorskie szkolenia  
z zakresu **bezpieczeństwa IT**

- { Bezpieczeństwo aplikacji WWW }
- { Offensive HTML, SVG, CSS and other Browser-Evil }
- { Wprowadzenie do bezpieczeństwa IT }
- { Szkolenie przygotowujące do egzaminu CEH  
( Certified Ethical Hacker ) }

[www.securitum.pl/oferta/szkolenia](http://www.securitum.pl/oferta/szkolenia)

Patroni medialni: [sekurak.pl](http://sekurak.pl)



[rozwal.to](http://rozwal.to)



## Listing 2. Minimalny przykład playbooka Ansible

```
---
- hosts: all
  tasks:
    - name: Create hello file
      file:
        path: /tmp/hello
        state: touch
    - name: List files in tmp
      find:
        paths: /tmp
        recurse: yes
      register: tmp_files
    - name: Print found tmp_files
      debug: var=tmp_files
```

Plik zapisujemy pod dowolną nazwą, w moim przypadku *hello.yml*. Istotne jest, aby oba pliki znajdowały się w jednym katalogu. Plik playbooka jest dość prosty w swojej treści. Dwa podstawowe klucze definiują, na których hostach należy wykonać zadania oraz listę zadań do wykonania. W bardziej zaawansowanych konfiguracjach hosty możemy grupować, a playbooki wykonywać na jednej lub wielu grupach hostów.

Zadania opisane w playbooku odpowiadają wymienionym w specyfikacji powyżej. Pierwsze z nich wykorzystuje moduł `file`, za pomocą którego deklarujemy, gdzie należy utworzyć plik. Co ważne, nie definiujemy, w jaki sposób ma zostać utworzony plik – moduł sam powinien wiedzieć, jaka jest metoda implementacji. W drugim kroku za pomocą modułu `find` odczytujemy zawartość katalogu `/tmp`. Warto zauważyć, że to zadanie poza wykonaniem modułu `find` rejestruje zmienną o nazwie `tmp_files`. W bardziej zaawansowanych konfiguracjach możemy uruchamiać kolejne kroki playbooka warunkowo, porównując wartość zmiennej z użyciem klucza `when`. W naszym przykładzie ograniczyliśmy się do wypisania zawartości zmiennej z użyciem modułu `debug`.

Omówiliśmy treść playbooka, czas sprawdzić go w akcji. Uruchomienie jest bardzo proste i sprowadza się do komendy: `ansible-playbook -i inventory.ini hello.yml`. Przelącznik `-i` informuje Ansible, z którego pliku należy pobrać listę hostów. Warto zauważyć, że dzięki takiemu rozwiązaniu możemy wygodnie wykonywać te same zadania na różnych listach hostów w zależności od potrzeb.

## Listing 3. Log wykonania playbooka

```
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [localhost]

TASK [Create hello file] ****
changed: [localhost]

TASK [List files in tmp] ****
ok: [localhost]

TASK [Print found tmp_files] ****
ok: [localhost] => {
  "tmp_files": [
    {
      "changed": false,
      "examined": 11,
      "failed": false,
      "files": [
        ...
      ],
      "matched": 3,
      "msg": ""
    }
}

PLAY RECAP ****
localhost      : ok=4      changed=1      unreachable=0      failed=0
```

Jak zauważamy w Listingu 3, playbook wykonał się prawidłowo.

Za każdym razem operował na hoście `localhost`, co wynikało bezpośrednio z treści pliku `inventory.ini`.

Dzięki temu przykładowi pozyskaliśmy wiedzę o stworzeniu minimalnego przykładu. Playbook wykorzystany w instalacji Ranchera jest bardziej złożony, a struktura katalogów została podrządkowana tej, którą opisano w dokumencie wymieniającym dobre praktyki pracy z Ansible.

## RANCHER ANSIBLE PLAYBOOK

Do instalacji Ranchera wykorzystamy przygotowany przez jego autorów playbook [4], który został przygotowany dla hostów operujących pod systemami z rodziną Ubuntu. Playbook znajdziemy w repozytorium: <https://github.com/rancher/ansible-playbooks>.

W Listingu 4 przedstawiona strukturę katalogów w repozytorium podobną do tej, którą przedstawiono w dokumentacji. Główna różnica między nimi leży w zapisie `inventory`, ponieważ przygotowany przez autorów Ranchera playbook jest w stanie wydać aplikację również w infrastrukturze EC2. W tym artykule skupimy się jednak na przypadku instalacji w ramach własnej infrastruktury.

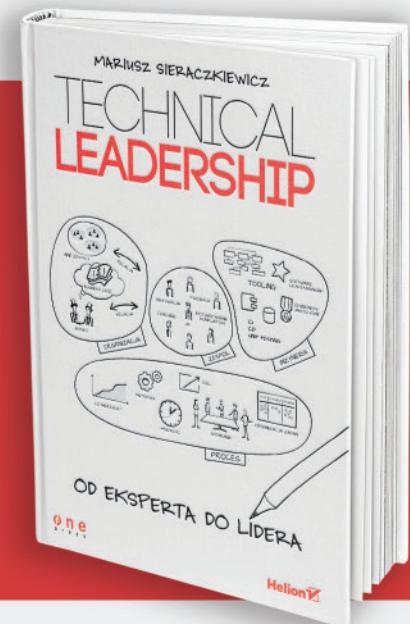
## Listing 4. Struktura katalogów w repozytorium

```
└── ansible.cfg
└── haproxy.yml
└── INSTALL_EC2.md
└── INSTALL.md
└── inventory
    ├── group_vars
    │   ├── all.yml
    │   ├── ec2.yml
    │   └── server.yml
    └── hosts
    └── host_vars
└── inventory_providers
    └── ec2.ini
    └── ec2.py
└── rancher.yml
└── README.md
└── roles
    ├── haproxy
    │   ├── defaults
    │   │   └── main.yml
    │   ├── files
    │   ├── handlers
    │   │   └── main.yml
    │   ├── meta
    │   ├── tasks
    │   │   └── haproxy.yml
    │   │       └── main.yml
    │   └── templates
    │       └── haproxy.cfg.j2
    └── vars
    └── rancher
        ├── defaults
        │   └── main.yml
        ├── files
        ├── handlers
        ├── meta
        ├── tasks
        │   └── docker.yml
        │   └── main.yml
        └── templates
            └── server_launch.j2
        └── vars
    └── site.yml
    └── vars
        └── default.yml
        └── private.yml
```

# BNS IT - SZKOLENIA OTWARTE

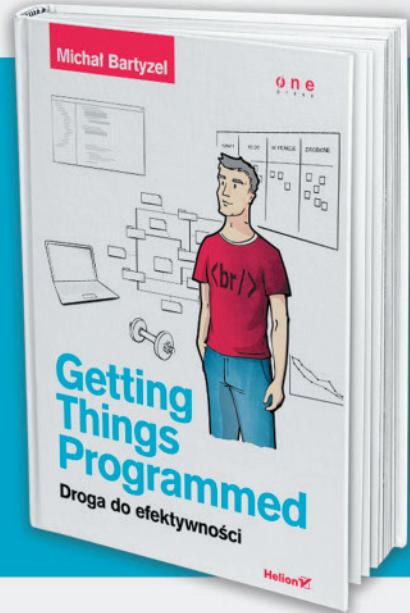
WARSZAWA / 18-20.04.2018  
**TECHNICAL LEADERSHIP™**  
ROLA LIDERA TECHNICZNEGO

1. Rola lidera technicznego
2. Motywacja własna i innych
3. Ludzie
4. Zespół
5. Kompetencje lidera



ŁÓDŹ / 12-13.04.2018  
**GETTING THINGS PROGRAMMED**  
DROGA DO EFEKTYWNOŚCI

1. Oprogramowanie „na czas”
2. Opracowywanie zadań
3. Planowanie pracy programisty
4. Wykonywanie zadań programistycznych
5. Szacowanie zadań programistycznych



## P O Z O S T A Ł E   S Z K O L E N I A   O T W A R T E :

Zbieranie wymagań i współpraca z klientem

Łódź 16-18.05.2018 2100,00 PLN

Wzorce projektowe i refaktoryzacja do wzorców

Warszawa 16-18.05.2018 2100,00 PLN

Tworzenie Microservices z użyciem Spring Boot

Warszawa 28-30.05.2018 2100,00 PLN

Wzorce projektowe i refaktoryzacja do wzorców

Łódź 11-13.06.2018 2100,00 PLN

Technical Leadership™

Warszawa 04-06.07.2018 2100,00 PLN

Nowoczesne architektury aplikacji

Warszawa 11-13.07.2018 2100,00 PLN

CENY NETTO

Istotą każdego playbooka Ansible są role. Wyrażają one zbiór oczekiwani, jakie administrator stawia wobec serwera odgrywającego daną rolę. W rozpatrywanym przykładzie zaimplementowano dwie role, kolejno haproxy oraz rancher.

Przed przystąpieniem do przygotowania maszyn wirtualnych i instalacji omówimy proces instalacji Ranchera zawarty w playbooku. Odbędzie się ona dwuetapowo. Pierwszym krokiem będzie instalacja węzła zarządzającego oraz Haproxy (jest ono opcjonalne). Do instalacji węzłów wykonawczych potrzebne będą klucz oraz sekret, pozwalające na przyłączenie maszyn do klastra. Po instalacji głównego węzła generujemy klucze dostępu, a następnie przeprowadzamy instalację reszty klastra.

Pierwszy krok zrealizujemy za pomocą polecenia: `ansible-playbook --limit server,loadbalancer rancher.yml`.

## Listing 5. Zawartość pliku `rancher.yml`

```
---  
- hosts:  
  - server  
  become: True  
  gather_facts: True  
  vars_files:  
    - vars/default.yml  
    - vars/private.yml  
  roles:  
    - { role: rancher, tags: rancher }  
    - { role: haproxy, tags: haproxy, when: "groups['loadbalancer']|count == 0"}  
  tags:  
    - rancher  
  
- hosts:  
  - node  
  become: True  
  gather_facts: True  
  vars_files:  
    - vars/default.yml  
    - vars/private.yml  
  roles:  
    - { role: rancher, tags: rancher }  
  tags:  
    - rancher
```

W pliku `rancher.yml` znajdziemy odzwierciedlenie opisanych wcześniej dwóch etapów. Wspomniane wyżej polecenie spowoduje, że w pierwszym etapie wykonamy role przypisane do hostów z grupy `server`, będą to kolejno `rancher` oraz `haproxy`.

Analizę roli `rancher` należy rozpocząć od katalogu `roles/rancher/tasks` – jest to katalog zawierający pliki `yaml` z opisami kroków składających się na wykonanie zadania. Plik `main.yml` wskazuje kolejność oraz warunki, w których należy uruchamiać zadania.

## Listing 6. Plik główny roli `rancher`

```
---  
- name: get public IP  
  ipify_facts:  
  when: not public_ip|default(False)  
  
- include: docker.yml  
  tags: docker  
  
- include: server.yml  
  tags: server  
  when: "'server' in group_names"  
  
- include: node.yml  
  tags: node  
  when: "'node' in group_names and rancher_env in api_keys|default([])"
```

Interpretacja pliku jest intuicyjna. Na rolę składają się cztery zadania: określenie adresu IP, przygotowanie Dockera, instalacja serwera oraz węzła. W zależności od grup, do których ograniczono wykonanie playbooka, oraz wartości zmiennych zawartych w plikach katalogu `/vars`, wykonane zostaną odpowiednie zadania. Pierwsze z wymienionych zadań wykorzystuje moduł Ansible o nazwie `ipify_facts`. Jego opis można znaleźć w oficjalnej dokumentacji [5]. Kolejne trzy role opisane są przez odpowiednie pliki konfiguracyjne.

## Listing 7. Opis instalacji Dockera

```
---  
- name: determine if docker is installed  
  command: which docker  
  register: docker_installed  
  ignore_errors: true  
  changed_when: false  
  
- name: get docker version  
  shell: docker --version | awk '{ print $3 }'  
  register: docker_installed_version  
  when: docker_installed|success  
  changed_when: false  
  
- name: install or upgrade docker  
  shell: curl https://releases.rancher.com/install-docker/{{ docker_version }}.sh | sh  
  when: not docker_installed|success or  
        not docker_installed_version.  
  std_out|search(docker_version|string)  
  
- name: uninstall pip  
  apt: name=python-pip state=absent  
  when: ansible_os_family == "Debian"  
  
# Debian ships with pip v1.5 in apt, which is too broken (#5)  
- name: install pip (Debian)  
  easy_install: name=pip state=latest  
  when: ansible_distribution == "Debian"  
  
- name: upgrade pip (Ubuntu)  
  pip: name=pip state=latest  
  when: ansible_os_family == "Debian" and not ansible_distribution == "Debian"  
  
- name: install additional dependencies (pip)  
  pip: name=docker-py state=present  
  tags: pip  
  
- name: add our login user to docker group  
  user: name={{ansible_user}} groups=docker append=yes  
  
- name: make sure docker is running  
  service: name=docker state=started enabled=true
```

Na instalację Dockera składa się dziewięć kroków. Pierwsze dwa kroki to sprawdzenie, czy Docker jest obecny na hoście oraz jaka jest jego wersja. Kolejnym etapem jest uruchomienie instalacji. Istotnym elementem polecenia jest warunek instalacji – jeśli poprzednio Ansible wykrył, że Docker jest obecny – instalacja nie zostanie przeprowadzona.

W tym miejscu warto zaznaczyć, że moduły Ansible są idempotentne – ich ponowne wykonanie ustawi oczekiwany stan na maszynie bez powodowania efektów ubocznych. Dzięki zachowaniu tej cechy opis roli ma postać deklaracji stanu końcowego, a nie kolejnych kroków do wykonania.

Świadomość tej cechy zdecydowanie ułatwia rozumienie konstrukcji roli. Korzystając z tej wiedzy, możemy wywnioskować, że w kolejnych krokach następują: aktualizacja narzędzia pip, dodanie aktualnego użytkownika do grupy Docker oraz upewnienie się, że usługa Docker jest aktywna. W ten sposób kończy się instalacja Dockera. Rola `rancher` przedstawiona w Listingu 6 przewiduje w tym miejscu dwie możliwości – instalację serwera lub węzła.

**Listing 8. Zadanie server roli rancher**

```

---
- name: install additional dependencies (apt)
  apt: name={{ item }} state=present
  with_items:
    - python-mysqldb
    - mysql-client
  tags: apt

- name: sanity check on options
  fail: msg="Please set external db options when running with >
1 server or with use_external_db=true"
  when: (use_external_db|default(False) or hostvars[inventory_
hostname]['groups']['server'].length > 1) and
        (not db_host or not db_name or not db_user or not db_pass or
        not db_port )

- name: create the database (if necessary)
  mysql_db:
    name: "{{ db_name }}"
    collation: utf8_general_ci
    encoding: utf8
    login_host: "{{ db_host }}"
    login_user: "{{ db_admin_user }}"
    login_password: "{{ db_admin_pass }}"
    state: present
    login_port: "{{ db_port }}"
  run_once: true
  when: use_external_db|default(False) or hostvars[inventory_
hostname]['groups']['server'].length > 1
  tags:
    - mysql

- name: create the database user (if necessary)
  mysql_user:
    name: "{{ db_user }}"
    host: '%'
    password: "{{ db_pass }}"
    login_user: "{{ db_admin_user }}"
    login_password: "{{ db_admin_pass }}"
    login_host: "{{ db_host }}"
    login_port: "{{ db_port }}"
    state: present
    priv: "{{ db_name }}.*:ALL"
  run_once: true
  when: use_external_db|default(False) or hostvars[inventory_
hostname]['groups']['server'].length > 1
  tags:
    - mysql

- name: check if rancher container is running
  shell: docker ps | grep rancher/server
  register: server_running
  ignore_errors: true
  changed_when: false

- name: remove any previous rancher server (to avoid conflict)
  docker_container: name={{ server_name }} state=absent
  when: server_name is defined and
        not server_running|success

- name: start rancher server
  command: '{{lookup("template", "server_launch.j2")}}'
  when: not server_running|success

```

Instalacja serwera została przedstawiona w Listingu 8. Składa się ona z siedmiu zadań, pośród których można wyróżnić dwie grupy. Pierwsza grupa to zadania instalujące bazę danych MySQL wraz z narzędziami służącymi do jego obsługi. Do przeprowadzenia instalacji wykorzystujemy moduł apt. Pozwala on na wykorzystanie menedżera paczek dostępnego w Ubuntu czy Debianie do instalacji pakietów o zadanych nazwach – w tym przypadku są to mysql\_user i mysql\_db.

Druga grupa zajmuje się uruchomieniem kontenera z Rancherem. W pierwszych dwóch krokach upewniamy się, że nie pracuje żaden kontener z serwerem Ranchera. Ostatnim zadaniem jest uruchomienie kontenera. Zadanie to jest o tyle ciekawe, że w jego implementacji zastosowano system szablonów. Komenda uruchomionowa Ranchera budowana jest przez uzupełnienie szablonu server\_launch.j2 zawartością zmiennych.

**Listing 9. Zadanie node roli rancher**

```

---
# tasks specific to nodes
- name: determine if we're already running an agent
  shell: "docker ps | fgrep -q rancher/agent"
  register: agent_installed
  ignore_errors: true
  changed_when: false

# this gets all of the available registration tokens
# for this environment.
- name: get registration tokens
  uri:
    method: GET
    status_code: 200
    url: "{{ server_url }}/v2-beta/projects/{{ rancher_env }}/
registrationTokens/"
    user: "{{ api_keys[rancher_env].access_key }}"
    password: "{{ api_keys[rancher_env].secret_key }}"
  register: registration_tokens
  when: agent_installed|failed

# some or all of the tokens might be inactive, so we need
# to find the first active one and use it to register
# our host
- name: register host (attempt one)
  command: "{{(registration_tokens.json.
data|selectattr('state', 'equalto', 'active'))|first}}.
command|replace('sudo ', '')"
  when: agent_installed|failed and
        registration_tokens.json.data|selectattr('state',
'equalto', 'active')|list|length > 0
  register: registration_attempt_one

# if we have no active registration tokens, we need
# to create one
- name: create registration url
  uri:
    method: POST
    status_code: 201
    url: "{{ server_url }}/v2-beta/projects/{{ rancher_env }}/
registrationTokens/"
    user: "{{ api_keys[rancher_env].access_key }}"
    password: "{{ api_keys[rancher_env].secret_key }}"
  when: agent_installed|failed and
        registration_attempt_one|skipped
  run_once: true

# having created one, we can now fetch it
- name: get registration tokens (again)
  uri:
    method: GET
    status_code: 200
    url: "{{ server_url }}/v2-beta/projects/{{ rancher_env }}/
registrationTokens/"
    user: "{{ api_keys[rancher_env].access_key }}"
    password: "{{ api_keys[rancher_env].secret_key }}"
  register: registration_tokens_redux
  when: agent_installed|failed and
        registration_attempt_one|skipped

# we now have an active registration token that we can
# use to register the host
- name: register host (attempt two)
  command: "{{(registration_tokens_redux.json.
data|selectattr('state', 'equalto', 'active'))|first}.command}}"
  when: agent_installed|failed and
        registration_attempt_one|skipped

```

Ostatnim zadaniem, wykonywanym w ramach roli rancher.yml, jest node.yml. Jego treść została zaprezentowana w Listingu 9. Zostanie ono wykonane tylko na hostach, które jeszcze nie mają uruchomionego agenta Ranchera. Instalacja węzła klienckiego wymaga pozyskania tokenu rejestracyjnego z serwera centralnego. Z tego powodu instalacja Ranchera będzie dwuetapowa, przed przystąpieniem do instalacji należy skonfigurować główny węzeł klastra.

Rejestracja agenta Ranchera na serwerze centralnym przebiega następująco:

- Po skonfigurowaniu centralnego serwera klient otrzymuje klucz i sekret API.

2. Dalsza komunikacja wymaga każdorazowej identyfikacji klienta za pomocą klucza i sekretu.
3. Klient pobiera listę tokenów rejestracyjnych z serwera, analizuje ją w poszukiwaniu aktywnych tokenów rejestracyjnych.
4. Jeśli nie odnaleziono aktywnego tokenu, klient zleca jego utworzenie, następnie powtarza krok 3.
5. Jeśli odnaleziono token aktywacyjny, klient używa go do zarejestrowania się w klastrze Ranchera.

## INSTALACJA RANCHERA ZA POMOCĄ ANSIBLE

Poznaliśmy strukturę i sposób budowy playbooków dla narzędzia Ansible. Przed przystąpieniem do konfiguracji zdalnych maszyn za jego pomocą należy przeprowadzić instalację. Zależnie od stosowanej platformy mamy kilka możliwości. Ansible został napisany w języku Python, więc sposobem instalacji, który zadziała na wszystkich platformach, jest użycie narzędzia pip: `pip install -u ansible`. Pozostałe sposoby instalacji można znaleźć w dokumentacji [6].

Do instalacji wykorzystamy dwie maszyny wirtualne z zainstalowanym systemem Ubuntu w wersji 16.04. Jeśli nie dysponujemy maszynami, na których można przeprowadzić instalację, zawsze można skorzystać z poradników dotyczących uruchamiania maszyny wirtualnej na VirtualBox [7].

Po utworzeniu maszyn wirtualnych przystępujemy do działania. Znając adres IP maszyn wirtualnych, uzupełniamy plik `/inventory/hosts` tak jak pokazano w Listingu 10.

**Listing 10. Początek pliku hosts**

```
# static groups
[static_server]
10.129.171.65
[static_node]
10.129.161.36
```

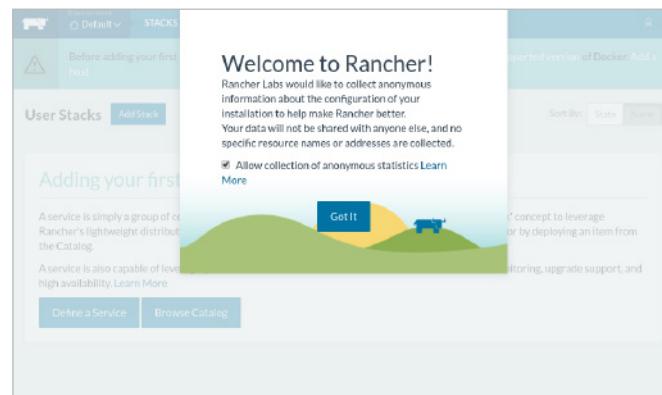
W moim przypadku adres serwera centralnego to 10.129.171.65, natomiast dla węzła wykonawczego 10.129.161.36. Dalszą część pliku pozostawiamy bez zmian. Maszyny, które wykorzystałem w tym przykładzie, są skonfigurowane tak, aby korzystać z nich za pomocą konta root (nie należy traktować tego jako konfigurację produkcyjną). W związku z tym w pliku `ansible.cfg` należy zmienić nazwę użytkownika, który będzie wykonywał polecenia Ansible.

Bardzo istotne jest, aby podczas instalacji, w katalogu domowym użytkownika domyślnego umieścić własny klucz publiczny w pliku `~/.ssh/authorized_keys`. Ostatnią potencjalną zmianą, którą należy wykonać przed uruchomieniem, jest ustawienie publicznego adresu IP. Podczas analizy roli `rancher` w Listingu 4 widzieliśmy zadanie pobrania publicznego adresu. Jeśli instalacja przebiega w prywatnej sieci firmowej, zadanie `ipfy` się nie powiedzie. W takim przypadku należy adres publiczny ustawić w pliku `/roles/rancher/default/main.yml` zamiast domyślnej wartości `False`.

Oficjalny playbook Ranchera niestety nie sprzyja siedmiom korpacyjnym. Brak domyślnych ustawień proxy dla pobieranych pakietów oraz usługi Docker w systemie powodują, że konieczne jest wykonanie sporej ilości kroków manualnych. Z tego powodu stworzyłem własną wersję playbooka, jest ona dostępna na <https://gitlab.com/majk-p/rancher-ansible>. Miejsca, które należy uzupełnić, zawierają domenę `company.net` oraz adnotacje `#CHANGE_ME`.

W poprawnie skonfigurowanym repozytorium należy wykonać polecenie `ansible-playbook --limit server,loadbalancer`

`rancher.yml`, a przy pytaniu o hasło należy wpisać `ansible`. Jest to domyślne hasło, które można zmienić, używając narzędzia `ansible-vault`. Po zakończeniu instalacji w przeglądarce wpisujemy adres `http://ADRES_HOSTA_MASTERA:8080`.



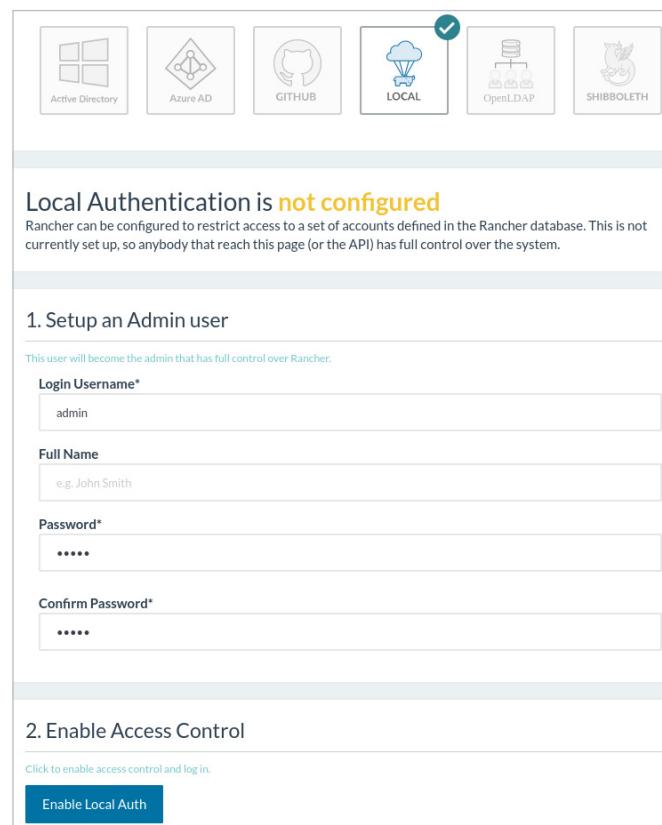
Rysunek 1. Ekran powitalny Ranchera

Pierwszym krokiem, który należy wykonać po instalacji, jest konfiguracja autoryzacji.



Rysunek 2. Menu główne Ranchera

W menu głównym klikamy w ikonę wykrzyknika. Zostaniemy przekierowani do ekranu ustawień autoryzacji.



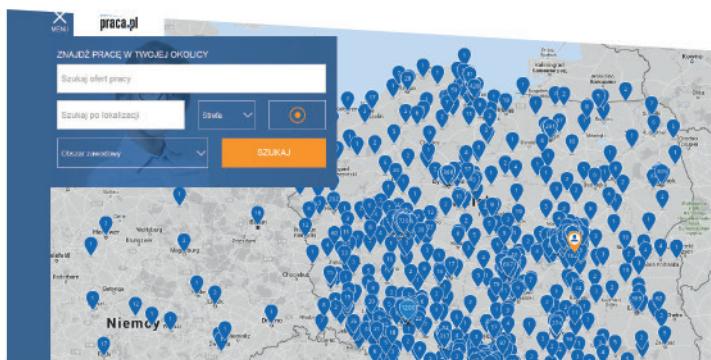
Rysunek 3. Ustawienia autoryzacji

Do wyboru mamy kilka sposobów autoryzacji. W moim przypadku jest to autoryzacja lokalna, z ręcznie utworzonym kontem admi-



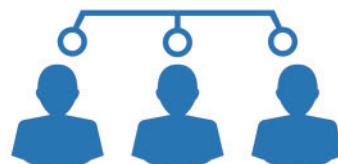
Chcesz dobrze zarobić?

Na Praca.pl codziennie znajdziesz ponad 3 000 ofert pracy z obszaru IT i nowe technologie



Znajdź pracę w Twojej okolicy

Lokalna.praca.pl



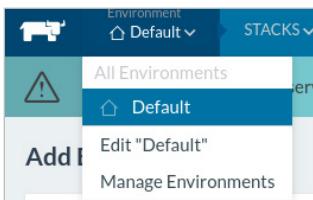
Poleć znajomego do pracy  
i zgarnij 1 000 zł

[Praca.pl/rekomendacje.html](http://Praca.pl/rekomendacje.html)

# ZARZĄDZANIE PROJEKTAMI

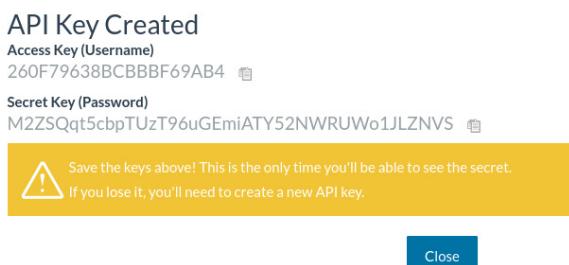
nistratora. Dostępne są również bardziej zaawansowane moduły, takie jak logowanie za pomocą LDAP, Active Directory czy konta GitHub.

Następnym krokiem jest utworzenie środowiska projektowego. W tym celu w menu głównym wybieramy *Manage environments*, jak pokazano na Rysunku 4.



Rysunek 4. Zarządzanie środowiskami

W następnym widoku klikamy *Add environment*, uzupełniamy formularz, wpisując nazwę projektu i wybierając *Cattle* jako menedżera konteneryzacji. Tak utworzony projekt aktywujemy w menu, w którym wcześniej przechodziliśmy do zarządzania środowiskami. W menu głównym wybieramy *Api → Keys* i rozwijamy pole *Advanced options*. Klikamy *Add environment API key* i uzupełniamy pola formularza nazwą charakterystyczną dla środowiska.



Rysunek 5. Utworzono klucze środowiska

Tak utworzone klucze należy koniecznie zapisać lokalnie, ponieważ nie będziemy mieli drugiej szansy, aby je odczytać (hasło do klucza przechowywane jest tylko w postaci skrótu).

Aktywne klucze środowiskowe znajdziemy w sekcji *Environment API Keys*.

Environment API Keys			
Add Environment API Key			
Environment API keys are tied to this specific Environment [ projekt ] and can only manipulate resources within there. Other account keys.			
Endpoint (v2-beta): <a href="http://10.129.171.65:8080/v2-beta/projects/1a7">http://10.129.171.65:8080/v2-beta/projects/1a7</a>			
Endpoint (v1): <a href="http://10.129.171.65:8080/v1/projects/1a7">http://10.129.171.65:8080/v1/projects/1a7</a>			
State	Name	Description	Access Key
Active	programista	None	260F79638BCBBBF69AB4

Rysunek 6. Klucze API

Po przejściu powyższych kroków nasza instalacja centralnego serwera Ranchera jest zakończona. Teraz możemy wykorzystać utworzone klucze API do konfiguracji węzłów wykonawczych. W celu konfiguracji w konsoli przechodzimy do głównego katalogu repozytorium i wykonujemy polecenie *ansible-vault edit vars/private.yml*.

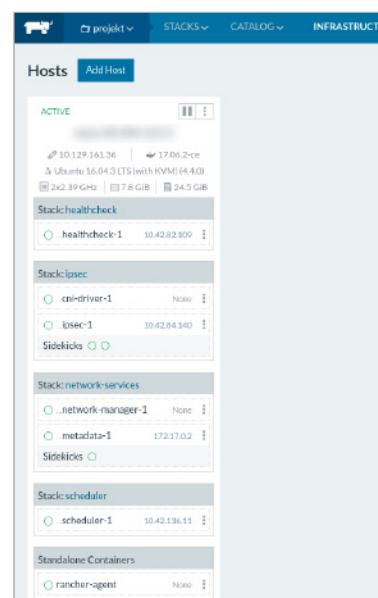
W ten sposób uruchomiliśmy okno edytora konsolowego. W otwartym pliku należy zmodyfikować sekcję *api\_keys*, wpisując tam dane klucza API:

## Listing 11. Klucze API w ansible-vault

```
# These are not real keys. They're only here to show you the
# format. Please change them.
api_keys:
  1a7:
    access_key: 260F79638BCBBBF69AB4
    secret_key: M2ZSQqt5cbpTUzT96uGEmiATY52NWRUWo1JLZNVS
```

Tak skonfigurowany playbook będzie w stanie zarejestrować nowo instalowane węzły wykonawcze do centralnego serwera Ranchera. Zaletą tego rozwiązania jest skalowalność horyzontalna – za każdym razem, kiedy nasz projekt będzie potrzebował nowych maszyn wykonawczych, wystarczy dopisać je w odpowiedniej sekcji pliku hosts.

Instalacja hostów wykonywana jest przez jedno polecenie *ansible-playbook --limit node\_rancher.yml*, po którego wykonaniu Ansible ponownie zapyta nas o hasło do rozszыfrowania sekretów. Po przeprowadzonej instalacji, w sekcji *Infrastructure → hosts* znajdziemy nowo dodane węzły.



Rysunek 7. Widok hostów wykonawczych

## URUCHAMIANIE APLIKACJI

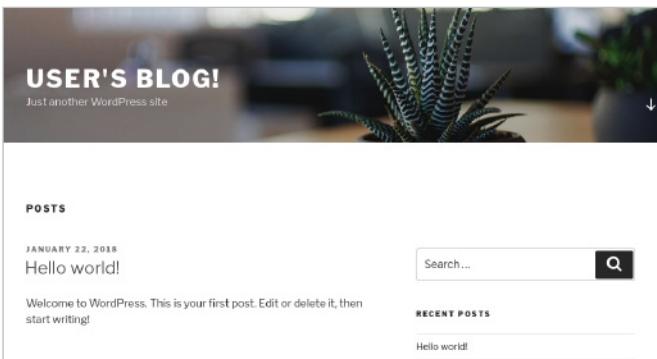
Jednym z ciekawych rozwiązań, wyróżniających Ranchera na tle podobnych rozwiązań, jest biblioteka aplikacji. Pozwala ona na uruchomienie konfiguracji zaproponowanych przez autorów Ranchera oraz jego społeczności. Uruchomienie aplikacji takich jak Wordpress sprowadza się do wybrania odpowiedniej pozycji z katalogu, zmodyfikowania konfiguracji według potrzeb i uruchomienia.

Po wybraniu Wordpressa z katalogu i oczekaniu minuty na inicjalizację, w widoku stosów aplikacyjnych znajdziemy wpisy przedstawione na Rysunku 8.

Stack:	wordpress	Add Service	Up to date	Active	⋮
Active	mariadb	Image: bitnami/mariadb:10.1		1 Container	⋮
Active	wordpress	Image: bitnami/wordpress:4		1 Container	⋮
Active	wordpress-lb	To: wordpress Ports: 80/tcp		1 Container	⋮

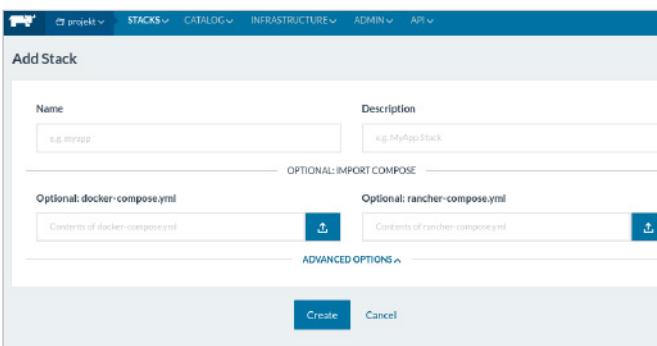
Rysunek 8. Zainstalowany Wordpress

Od tej chwili Wordpress jest gotowy do pracy. Po kliknięciu w link 80/tcp przy *load balancerze* zostaniemy przeniesieni do działającej instancji.



Rysunek 9. Działająca instancja Wordpressa

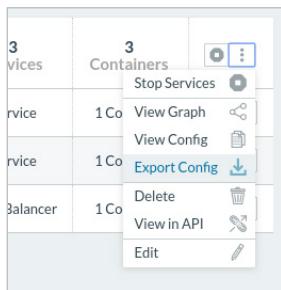
Poza gotowymi instalacjami Rancher umożliwia oczywiście uruchamianie własnych aplikacji. Srowadza się to do utworzenia nowego stosu i uzupełnienia formularza przedstawionego na Rysunku 10.



Rysunek 10. Nowy stos własnej aplikacji

Uruchamianie aplikacji, które wcześniej działały poprzez uruchomienie z użyciem narzędzia docker-compose, jest bardzo proste. W polu *name* wpisujemy nazwę naszego projektu, w polu *docker-compose.yml* wklejamy zawartość naszego pliku *docker-compose*. Plik *rancher-compose* jest opcjonalny i stanowi supplement informujący Ranchera o zasadach skalowania aplikacji, konfiguracji *load balancer* i konfiguracji początkowej.

Po uruchomieniu stosu aplikacji w każdym momencie mamy możliwość eksportu aktualnych ustawień w postaci dwóch plików konfiguracyjnych: *rancher-compose* oraz *docker-compose*. Dzięki temu Rancher nie wymaga od użytkownika znajomości składni swojej konfiguracji, a zmiany wykonane za pośrednictwem interfejsu użytkownika można wyeksportować z poziomu stosu, jak przedstawiono na Rysunku 11.



Rysunek 11. Eksport konfiguracji stosu

## RANCHER-COMPOSE

Wspomniany wcześniej *rancher-compose* to narzędzie analogiczne do *docker-compose* dla Dockera uruchamianego lokalnie. Jego źródła możemy znaleźć w oficjalnym repozytorium [8]. Twórcy narzędzia dążą do maksymalizacji kompatybilności z formatem *docker-compose* w celu uzyskania możliwie najlagodniejszego programu wejścia dla nowych użytkowników. Nawiązując do Rysunku 10 – *rancher-compose* poza wykorzystaniem pliku *docker-compose* przyjmuje jeszcze jeden plik, również zgodny z formatem YAML.

Format ten – podobnie jak narzędzie – nazywa się *rancher-compose* i opisuje właściwości serwisów specyficzne dla ich uruchamiania w ramach platformy Rancher. Najprostszym sposobem na utworzenie takiego pliku jest wykonanie kilku prostych kroków:

- » po zalogowaniu do interfejsu Rancher tworzymy nowy stos (Rysunek 10),
- » uzupełniamy nazwę i opis,
- » wklejamy *docker-compose.yml* naszego projektu i uruchamiamy stos,
- » dopasowujemy ustawienia aplikacji, modyfikując ustawienia dystrybucji na węzłach wykonawczych czy dodając *load balancer*,
- » eksportujemy pliki konfiguracyjne.

W ten sposób utworzyliśmy konfigurację pozbawioną błędów składniowych. Ponadto proces nie wymaga posiadania dodatkowej wiedzy z zakresu konfiguracji Ranchera, co czyni go jeszcze bardziej przystępny dla nowych użytkowników.

*Rancher-compose* jest doskonałym narzędziem do wkomponowania w procesie CI/CD takim jak Gitlab CI (więcej w moim artykule „Gitlab CI – od pomysłu do produkcji” w numerze 08/2017).

## PODSUMOWANIE

Rancher jest wygodnym i elastycznym rozwiązaniem, które w wielu przypadkach może okazać się najkorzystniejszym wyborem menedżera konteneryzacji. Dzięki zaproponowanej automatyzacji jego instalacja nie powinna zająć wiele czasu, a przystępność interfejsu i zgodność z *docker-compose* zostanie doceniona przez korzystających z Ranchera programistów.

### W sieci

- [1] <https://github.com/rancher/rancher/wiki/Rancher-Project-Plan>
- [2] <https://goo.gl/CTBTNE>
- [3] <https://goo.gl/q4QMqh>
- [4] <https://github.com/rancher/ansible-playbooks>
- [5] [https://docs.ansible.com/ansible/latest/\\_modules/ipify\\_facts.html](https://docs.ansible.com/ansible/latest/_modules/ipify_facts.html)
- [6] [https://docs.ansible.com/ansible/latest/intro\\_installation.html](https://docs.ansible.com/ansible/latest/intro_installation.html)
- [7] <https://www.wikihow.com/Install-Ubuntu-on-VirtualBox>
- [8] <https://github.com/rancher/rancher-compose>

**MICHał PAWLIK**

admin@michalp.net

Site reliability engineer w firmie Nokia, entuzjasta programowania funkcyjnego i języka Scala. Absolwent informatyki Politechniki Wrocławskiej.

# Podglądarki i podsłuchuj, czyli jak testować oprogramowanie z użytkownikami

Tytuł może brzmieć nieco zaczepnie. Chodzi jednak o proste zasady, związane z testowaniem z użytkownikiem końcowym. Czy to mowa o prostej stronie internetowej sklepu e-commerce czy oprogramowaniu do kompleksowej obsługi transakcji płatniczych online, trzeba pamiętać o tych, którzy finalnie będą z tego korzystać. Bardzo często wpada się bowiem w pułapkę myślenia, że jako architekt systemu, grafik czy programista – „wczuję się”, „wiem lepiej”, „sam wymyślę”. Tak naprawdę każdy, kto uczestniczy w tworzeniu oprogramowania, ma już zaburzoną wizję potrzeb użytkownika końcowego. Znając proces powstawania, wiedząc, jak coś będzie działać, trudno spojrzeć „świeżym okiem”. Dlatego warto testować: obserwować i słuchać użytkownika.

## OBSERWUJ!

Słowo „podglądarki” brzmi dosyć niepokojąco, ale w kontekście testów oprogramowania z użytkownikiem końcowym jest jak najbardziej wskazane. Obserwuj użytkowników, gdy używają oprogramowania, podejrzyj ich podczas prób wykonania zdefiniowanych przez Ciebie zadań. Ale obserwuj tak, by nie czuli „oddechu na plecach” i nie myśleli, że to jest jakiś sprawdzian umiejętności, gdyż w ten sposób cały test będzie zaburzony. Dlatego „podglądarki” jest najwłaściwszym określeniem. Dyskretnie patrz, co użytkownik robi, i zrób to tak, by nie tworzyć atmosfery „laboratorium”.

## SŁUCHAJ!

Jeśli chodzi o słuchanie, tu trzeba pamiętać o jednym. Słuchać warto, byle tylko to, co użytkownik opowiada, było na temat. Dlatego tak dobrze sprawdza się tzw. „protokół głośnego myślenia”, który polega na tym, by użytkownik opowiadał na głos, co i dlaczego robi. Test składa się wtedy z konkretnych zadań, które osoba testująca oprogramowanie musi wykonać (np. „wykliknąć” ścieżkę znajdowania produktu X, wrzucenie do koszyka, zakup i płatność), oraz z mówionego na głos komentarza, doprecyzowującego sposób, w jaki użytkownik używa oprogramowania. Pozwala to uniknąć słuchania opinii i porad od testera, które oczywiście mogą być też cenne. Jednak gdy testujesz oprogramowanie, chcesz dowiedzieć się, co konkretne sprawia trudność, a co działa dobrze. W tym momencie rozważania o tym, czy takie oprogramowanie ma w ogóle sens, niezbędny jest przydadzą.

## PYTAJ!

Mając w planach testy z użytkownikami końcowymi, warto pamiętać zasadę: nie pytaj o przeszłość. Jeśli już masz okazję porozmawiać z kimś, kto potencjalnie będzie używał oprogramowania, które tworzysz, nie pytaj: co by zrobił w przeszłości. To, co usłyszysz,

będzie tylko myśleniem życzeniowym i może okazać się nieprawdziwe, gdy sytuacja będzie realna.

Co więc zrobić? Warto zadawać pytania odnoszące się do sytuacji z przeszłością testowanego użytkownika.

### Przykład:

**TAK:** Czy zdarzyło Ci się kiedyś kupić podobny produkt w sklepie internetowym? Jeśli tak, to dlaczego, co Cię zainteresowało? Jeśli nie, to dlaczego, co Cię zniechęciło?

**NIE:** Czy kupiłeś ten produkt, gdybyś był na stronie sklepu internetowego z taką ofertą? Co by Cię mogło w tym produkcie zainteresować, a co zniechęcić?

Zadając pytania odnoszące się do przeszłości, dowieš się, jak osoba się zachowywała, jakie decyzje podjęła i dlaczego, w odniesieniu do wydarzeń realnych. Zwiększa to prawdopodobieństwo, że używając Twojego oprogramowania już w przyszłości, zachowią podobnie, jeśli nie identycznie. Dzięki temu wynikom testu z użytkownikiem będą informacje pomagające na przykład coś usprawnić, uzyskane na bazie prawdziwych doświadczeń użytkownika, nie zaś na jego „wydaje-mi-się”.

## NIE MYŚL, ŻE WIESZ LEPIEJ!

Każdy członek zespołu zaangażowany w proces powstawania oprogramowania nie może być obiektywnym użytkownikiem. Z prostej przyczyny. Osoby uczestniczące w danym projekcie nie będą postrzegały w pełni się zdystansować do swojej pracy. Za każdym razem będą mieć z tyłu głowy przekonanie, że wiedzą, jak i dlaczego coś w danym oprogramowaniu działa, albo jaka była przyczyna wybrania takiego, a nie innego rozwiązania np. graficznego.

Te same osoby, używające produktów stworzonych przez kompletnie inne zespoły, będą użytkownikiem obiektywnym. Ponieważ wiedzą o tym, czym jest produkt, jak działa, jak powstawał

itd., będzie im niedostępna, będą zadawać podstawowe i często niewygodne pytania: dlaczego?, po co?, jak to zrobić?, gdzie to jest? Nie znając ograniczeń budżetowych, systemowych czy sprzętowych, mogą odkrywać oprogramowanie tak, jak nie może tego zrobić nikt, kto był zaangażowany w proces jego tworzenia. Czyli obiektywnie.

Niestety łatwo o tym zapomnieć. Możesz popełnić ten błąd właśnie dlatego, że czujesz się obiektywny, używając tak wielkiej ilości produktów, które stworzył ktoś inny. Wydaje się więc, że patrząc na swoje dzieło, podchodzisz do niego tak samo – racjonalnie, krytycznie czy entuzjastycznie. To niestety nie jest prawda, o czym wspomniane było w poprzednim akapicie. Znając cały proces powstawania jakiegoś oprogramowania, od razu tłumacysz sobie występujące w nim braki i mylnie zakładasz, że końcowy użytkownik też to zrobi. Z góry zakładasz, że będzie wiedział tyle samo, co Ty.

To błędne postępowanie jest czymś, czego trudno uniknąć. Teści z rzeczywistym użytkownikiem końcowym to naprawią. Twórcy

oprogramowania będą mogli posłuchać i zobaczyć, co jest w produkcie świetne, a co ewidentnie sprawia kłopoty podczas używania.

## I NA KONIEC: WERYFIKUJ!

Podchodząc do testów oprogramowania z użytkownikami końcowymi, warto mieć pewien dystans do wyników. Osoby, z którymi będziesz testować swoje dzieło, mogą równocześnie rozaśniać wiele tematów, jak i je zagmatwać. Jeden użytkownik zaznaczy, że jakaś funkcjonalność błędnie działa, a drugi stwierdzi, że działa wyśmienicie. Co wtedy? Jak zweryfikować, kto ma rację? Rozwiązać jest kilka, dwa najprostsze to: poszerzenie grupy testowej i zweryfikowanie tego dysonansu na większej liczbie osób lub zamiana testu z użytkownikami (jakościowym) na test A/B (ilościowy). Ale to już inna bajka. Więcej w kolejnych numerach! :)



**KATARZYNA MAŁECKA**

katarzyna@teina.co

Autorka od 9 lat odpowiada za prowadzenie projektów IT, posiada także praktyczne doświadczenie z zakresu UX: tworzenia logiki userflows, prototypowania (makietki lo-fi) oraz prowadzenia badań jakościowych w formie indywidualnych wywiadów, przy użyciu eye-trackera oraz EEG. Zawodowo: właścicielka agencji badawczej. Prywatnie: autorka bajek dla dzieci.

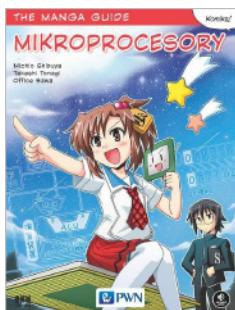
reklama

# Obserwuj nas na Twitterze

 @PROGRAMISTAMAG



# The Manga Guide. Mikroprocesory



Czy wydawcy w dzisiejszych czasach mogą nas czymś zaskoczyć? Okazuje się, że tak. Pojawiły się na naszym rodzimym rynku zupełnie nowe opracowania dość znanych tematów w zupełnie niespotykanym u nas stylu wydane przez wydawnictwo PWN. Pierwszą pozycję z serii *The Manga Guide*, jaka trafiła w moje ręce, są *Mikroprocesory*. Na pierwszy rzut oka może się wydawać, że temat jest już wyczerpany z możliwie każdej strony, jednak opracowanie tak złożonego zagadnienia w formie komiksu jest ogromnym powiewem świeżości na naszym rynku. Na 248 stronach autorzy w bardzo przystępny sposób omawiają zasady działania prostych procesorów. Cała pozycja jest praktycznie przeplatanką komiksu z dokładniejszym opisem bardziej skomplikowanych zagadnień zaprezentowanych w formie dialogu między głównymi bohaterami.

Tak, sama forma komiksu narzuca pewną konwencję, która jest utrzymana przez całą opowiedzianą historię. Samo to, że budowa i zasada działania mikroprocesorów opisana jest w formie lekkostrawnej historyjki, może intrygować. Jednak sposób prowadzenia dialogów, zachowania bohaterów dodaje dodatkowego smaczku całości.

Książka podzielona jest na sześć rozdziałów:

- » Rozdział 1: *Co robi procesor* – autorzy rozpoczynają swoją książkę od spotkania ze sobą głównych bohaterów, gdzie mistrzyni Shogi Ayumi zostaje poproszona o grę z komputerem zaprogramowanym przez Yuu, która to grę przegrywa. Jako że jest bardzo pewna siebie i zarozumiałą osobą, chce dowiedzieć się, jak to możliwe. Tak zaczyna się jej przygoda w odkrywanie, jak działa mikroprocesor.
- » Rozdział 2: *Działania na cyfrach* – kolejny rozdział poświęcony jest zagadnieniom związanym z przetwarzaniem liczb, systemem binarnym, działaniami logicznymi, dedykowanym układom arytmetycznym, wszystko opisane w tym samym lekkim stylu jak sam wstęp.
- » Rozdział 3: *Architektura procesora* – tu autorzy skupili się na podstawowych elementach składowych procesorów, omawiając pamięć, przetwarzanie instrukcji przez procesor, prze-

rwania, porty I/O, częstotliwość zegara, generatory na sygnale reset, na FLOPSach kończąc.

- » Rozdział 4: *Działania* – jest to kolejny dość złożony temat, zostały tutaj opisane typy działań, rodzaje instrukcji, typy argumentów, adresowanie, jednostka arytmetyczno-logiczna (ALU), rodzaje transmisji i finalnie przegląd podstawowych rejestrów i flag stanu.
- » Rozdział 5: *Programy* – rozdział ten poświęcony został programom, sposobom ich tworzenia, różnicami między językami wyższego poziomu a asemblerem, omówione zostały podstawy programowania i instrukcje skoków i warunków, autorzy pokusili się nawet o opisanie tego, co się dzieje po włączeniu komputera przed uruchomieniem programu.
- » Rozdział 6: *Mikrokontrolery* – ostatni rozdział poświęcony został mikrokontrolerom, ich zastosowaniu, funkcjom, architekturze, omówione zostały także układy DSP.

Lektura tej książki, czy może bardziej pasowałoby tutaj określenie komiksu, może być ciekawym doświadczeniem dla osób mających już określone pojęcie na temat mikroprocesorów. Dla pozostałych, którzy nie mieli styczności z tą tematyką, pozycja ta może być idealnym rozpoczęciem przygody z procesorami. Mimo tego, że w książce opisano bardzo proste układy, ilość przekazywanej wiedzy w ten komiksowy sposób jest zaskakująco duża.

Ze swojej strony mogę polecić *The Manga Guide. Mikrokontrolery* każdemu, kto chciałby spędzić w ciekawy sposób wolny czas, bo całość, zostawiając na chwilę na bok walory edukacyjne, jest też ciekawie opracowanym i momentami zabawnym komiksem z zakončeniem.

Mariusz „maryush” Witkowski

<b>Tytuł:</b>	<i>The Manga Guide. Mikroprocesory</i>
<b>Autor:</b>	Michio Shibuya, Takashi Tonagi, Office Sawa
<b>Stron:</b>	248
<b>Wydawnictwo:</b>	PWN
<b>Data wydania:</b>	2018

redakcja

Zamów prenumeratę magazynu Programista  
przez formularz na stronie:

<http://programistamag.pl/typy-prenumeraty/>

lub zrealizuj ją na podstawie faktury Pro-forma. W spisie faktur Pro-forma prosimy kontaktować się z nami drogą mailową:  
[redakcja@programistamag.pl](mailto:redakcja@programistamag.pl).

Prenumerata realizowana jest także przez RUCH S.A.

Zamówienia można składać bezpośrednio na stronie: [www.prenumerata.ruch.com.pl](http://www.prenumerata.ruch.com.pl)

Pytania prosimy kierować na adres e-mail: [prenumerata@ruch.com.pl](mailto:prenumerata@ruch.com.pl)

lub kontaktując się telefonicznie z numerem:

801 800 803 lub 22 717 59 59, godz. 7:00 – 18:00 (koszt połączenia wg taryfy operatora).

Magazyn Programista wydawany jest przez Dom Wydawniczy Anna Adamczyk

**Wydawca/Redaktor naczelny:** Anna Adamczyk ([annaadamczyk@programistamag.pl](mailto:annaadamczyk@programistamag.pl)).

**Redaktor prowadzący:** Michał Leszczyński ([mleszczyński@programistamag.pl](mailto:mleszczyński@programistamag.pl)).

**Korekta:** Tomasz Łopuszański. **Kierownik produkcji:** Havok. **DTP:** Havok.

**Dział reklamy:** [reklama@programistamag.pl](mailto:reklama@programistamag.pl), tel. +48 663 220 102, tel. +48 604 312 716.

**Prenumerata:** [prenumerata@programistamag.pl](mailto:prenumerata@programistamag.pl).

**Współpraca:** Michał Bartylek, Mariusz Sieraczkiewicz, Dawid Kaliszewski, Marek Sawerwain, Łukasz Mazur, Łukasz Łopuszański, Jacek Matulewski, Sławomir Sobótka, Dawid Borycki, Gynvael Coldwind, Bartosz Chrabski, Rafał Kocisz, Michał Sajdak, Michał Bentkowski, Mariusz „maryush” Witkowski, Paweł „KrazaQ” Zakrzewski.

**Adres wydawcy:** Dereniowa 4/47, 02-776 Warszawa.

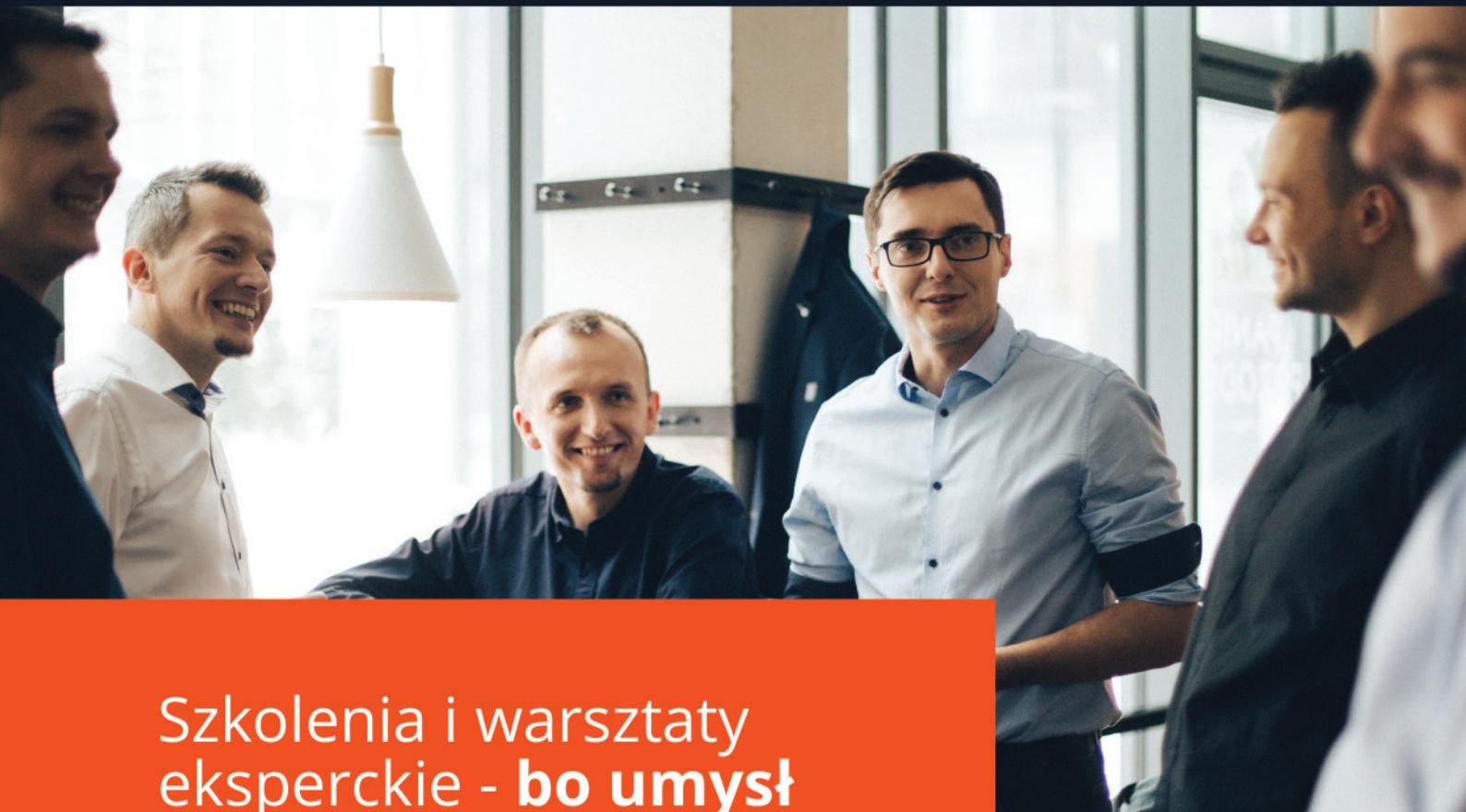
**Druk:** <http://www.moduss.waw.pl/>, Nakład: 4500 egz.

Nota prawa

Redakcja zastrzega sobie prawo do skrótów i opracowań tekstów oraz do zmian planów wydawniczych, tj. zmian w zapowiadanych tematach artykułów i terminach publikacji, a także nakładzie i objętości czasopisma.

O ile nie zaznaczono inaczej, wszelkie prawa do materiałów i znaków towarowych/firmowych zamieszczanych na łamach magazynu Programista są zastrzeżone. Kopiowanie i rozpowszechnianie ich bez zezwolenia jest zabronione.

Redakcja magazynu Programista nie ponosi odpowiedzialności za szkody bezpośrednie i pośrednie, jak również za inne straty i wydatki poniesione w związku z wykorzystaniem informacji prezentowanych na łamach magazynu Programista.



Szkolenia i warsztaty eksperckie - **bo umysł to Twoje najważniejsze narzędzie**



DDD



ARCH



TEST&CRAFT



AGILE&SOFT



JAVA



.NET



C&CPP



WEB



BAZY



MOBILNE



EIP

SPRAWDŹ **200**  
**AUTORSKICH**  
**PROGRAMÓW**  
SZKOLEŃ



ERICSSON

# JOIN THE E-TEAM

## Praca

Chcesz mieć wpływ na to, w którym kierunku zmierza postęp technologiczny? Brać udział w prestiżowych projektach IT realizowanych przez jednego z kluczowych graczy branży ICT? Rozwijać się zawodowo i żyć pełnią życia po pracy?

Dołącz do Ericsson. Stwarzamy naszym zespołom możliwości rozwijania umiejętności i dzielenia się pasjami. U nas zrealizujesz ciekawe projekty w dobrej, przyjacielskiej atmosferze.

Dowiedz się więcej o pracy w Ericsson:  
[Ericsson.com/careerspoland](http://Ericsson.com/careerspoland)



[tiny.cc/theEteam](http://tiny.cc/theEteam)