

Magazyn programistów i liderów zespołów IT



Cena 23,90 zł (w tym VAT 5%)



# APLIKACJE WEBOWE

TRANSMISJA DANYCH DŹWIĘKIEM  
W JAVASCRIPT OD PODSTAW:  
WEB AUDIO API



## MICROSOFT BOT FRAMEWORK



## JAK ROZPROSZYĆ SWOJĄ APLIKACJĘ



ISSN 2084-9400





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. Dowborczyków 25, 90-019 Łódź

## Algorytmiczne czarne skrzynki

Podczas naszych programistycznych podbojów niekiedy zdarza nam się stanąć przed koniecznością rozwiązyania jakiegoś skomplikowanego problemu. O ile algorytmika bardzo często wyciągnie do nas pomocną dłoń, o tyle nadal możemy się spotkać z sytuacją, gdzie prawidłowe opisanie relacji pomiędzy wejściem a oczekiwany wyjściem jest niezwykle trudne. Na ratunek przychodzi wtedy sieci neuronowe i algorytmy ewolucyjne. Podstawy działania tego typu narzędzi z punktu widzenia implementującego je programisty wyłożył Grzegorz Grzeda w swoim artykule „GANN w natariu, czyli genetycznie programowane sieci neuronowe”.

Gotową już implementację warto sprawdzić pod kątem wydajności. Optymalizacja nie jest sztuką prostą i wiąże się z wieloma nieintuicyjnymi kwestiami. Warto pamiętać, że już w przypadku języka C, pomimo relatywnie niskiego poziomu abstrakcji w porównaniu do innych technologii, od prawdziwego sprzętu dzieli nas naprawdę długa droga. „Co robi Twój program i dlaczego tak wolno?” to opracowanie na temat wydajności, o której w tym numerze pisze Maciej Czekaj.

Na temat wszechstronności Pythona w różnych zastosowaniach można powiedzieć bardzo wiele, ale czy mieliśmy świadomość, że posiada on zestaw narzędzi do opisywania sprzętu? MyHDL to paczka do wysokopoziomowego modelowania umożliwiająca konwersję kodu do VHDLa lub Veriloga. Kilka przykładowych implementacji z użyciem wspomnianego toolkitu pokazał i omówił w swoim artykule Marek Sawerwain.

To naturalnie nie wszystko, co można znaleźć w bieżącym numerze „Programisty”. Kontynuujemy bowiem serię na temat przetwarzania sygnałów cyfrowych, dużo miejsca poświęcono również innym rozwiązaniom z zakresu Webdev. Zapraszamy do lektury!

Michał Leszczyński

## PRENUMERATA

### BIBLIOTEKI I NARZĘDZIA

**GANN w natariu, czyli genetycznie programowane sieci neuronowe**.....4  
Grzegorz Grzeda

**Pakiet MyHDL. Python w służbie sprzętu i układów FPGA**.....14  
Marek Sawerwain

### PROGRAMOWANIE SYSTEMOWE

**Co robi Twój program i czemu tak wolno?**.....22  
Maciej Czekaj

### PROGRAMOWANIE APLIKACJI WEBOWYCH

**CMS as a Service, czyli Umbraco w chmurze!**.....30  
Marcin Zajkowski

**Microsoft Bot Framework. Tworzenie inteligentnych kanałów komunikacyjnych**.....36  
Dawid Borycki

**Jak rozproszyć swoją web aplikację?**.....42  
Maciej Nabożyny

**Transmisja danych dźwiękiem w JavaScript od podstaw. Część 2: Web Audio API**.....48  
Robert Rypuła

### FELIETON

**Zawód: Data Scientist, czyli jak zostać jednoróżcem**.....66  
Łukasz Kobyliński

### KLUB LIDERA IT

**Jak sprzedać refaktoryzację? Przypadek Nordea Bank AB**.....68  
Michał Bartylek, Łukasz Korczyński

### STREFA CTF

**BSidesSF 2017 – Pinlock**.....74  
Jarosław Górný

### PLANETA IT

**Angielski dla programistów. Lekcja 2**.....78  
Łukasz Piwko

### KLUB DOBREJ KSIĄŻKI

**WordPress i Joomla! Zabezpieczanie i ratowanie stron WWW**.....80  
Mariusz "maryush" Witkowski

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](http://prenumerata.programistamag.pl).

**Współpraca:** Michał Bartylek, Mariusz Sierackiewicz, 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 „KrzaQ” Zakrzewski.

**Adres wydawcy:** Dereniowa 4/47, 02-776 Warszawa.

**Druk:** Drukarnia Edit ul. Dworkowa 2, 05-462 Wiązowna, Nakład: 4500 egz.

Nota prawa

Redakcja zastrzega sobie prawo do skrótów i opracowania 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.

# GANN w natarciu, czyli genetycznie programowane sieci neuronowe

Programowanie jest nierozerwalnie związane z rozwiązywaniem rzeczywistych problemów. Stosując różne techniki, jesteśmy w stanie zamodelować skomplikowane procesy, wyszukiwać relacje między obiektami oraz przetwarzać gromadzone dane z ogromną szybkością. Jednak co zrobić, gdy nie istnieje optymalny przepis na rozwiązanie danego problemu? Gdy ograniczone zasoby nie pozwalają na sprawdzenie wszystkich możliwości, jak mieć pewność, że wyznaczone rozwiązanie jest prawidłowe? Pomocne mogą okazać się algorytmy ewolucyjne oraz sieci neuronowe.

Celem tego artykułu jest pokazanie, jak w prosty sposób zaimplementować skuteczne mechanizmy sztucznej inteligencji. O ile istnieją w Internecie niezliczone biblioteki programistyczne zarówno do sieci neuronowych, jak i algorytmów genetycznych, to trudno znaleźć przepis na małe, proste rozwiązania. Prezentowany kod jest przygotowany w języku Java, jednak nic nie stoi na przeszkodzie, aby czytelnik przepisał go na dowolnie inny. Języki obiektowe w naturalny sposób oddają dynamikę związaną np. podczas „reprodukcyj” lub doboru naturalnego w algorytmach genetycznych. Nie oznacza to jednak, że implementacje w C przedstawionych niżej rozwiązań będą nieefektywne. Własna wydajna biblioteka w C np. na mikrokontroler będzie w stanie skutecznie przetwarzać dane, bez znacznego ustępowania profesjonalnym, prekompilowanym bibliotekom ze sprzętową akceleracją.

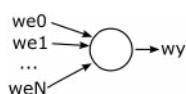
W ramach tego artykułu pokażę, jak implementować proste rozwiązania sieci neuronowych oraz algorytmów ewolucyjnych. Artykuł ten można potraktować jak mapę drogową, wprowadzającą we wspariały świat implementowania metod sztucznej inteligencji. Przecież każda podróż zaczyna się od pierwszego kroku...

## PODSTAWY SIECI NEURONOWYCH

### Perceptron

Idea sieci neuronowych polega głównie na symulowaniu zachowania (ludzkiego) mózgu. Jeśli popatrzy się z odpowiedniej perspektywy na ten wyjątkowy narząd, zauważymy, że jest to po prostu sieć komórek nerwowych odpowiednio połączonych ze sobą. Dla potrzeb tego artykułu taki poziom szczegółów jest wystarczający.

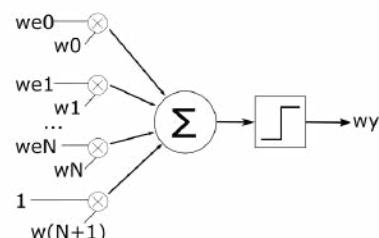
Każda sieć, nawet ta bardzo skomplikowana, składa się z pojedynczych neuronów. Najprostszą odmianą neuronu jest zaproponowany pod koniec lat 50. ubiegłego wieku model zwany *perceptronem*. Jest to prosta jednostka obliczeniowa, posiadająca N wejść (przynajmniej jedno) oraz jedno wyjście (Rysunek 1).



Rysunek 1. Pojedynczy perceptron

Klasyczny perceptron działa według następującej zasady: wartości wejściowe zostają przemnożone przez odpowiednie wagи, następ-

nie powstałe iloczyny są sumowane. Suma tych produktów jest podawana jako argument tak zwanej funkcji aktywacji, która służy podejmowaniu decyzji przez perceptron. Decyzja jest kierowana na wyjście (Rysunek 2). Pojawia się również dodatkowy składnik. Symbolicznie zaznaczono go jako iloczyn wagi przez liczbę 1. Jest to tzw. polaryzacja (ang. bias) perceptronu. Bez tego składnika perceptron nie mógłby wykonać wszystkich operacji.



Rysunek 2. Szczegółowa budowa perceptronu

Osobny komentarz należy się wspomnianej funkcji aktywacji. Tak naprawdę jest to zwykła funkcja, która przetwarza wyliczoną sumę do postaci zgodnej z formatem wyjściowym. Na przestrzeni lat przebadano wiele funkcji, które można by zastosować w roli aktywatora odpowiedzi perceptronu. Ostatecznie opracowano zbiór funkcji, które dobrze sprawdzają się w tej roli. Będą one szerzej omówione w drugiej części artykułu. Teraz posłużymy się najprostszą z nich, czyli funkcją  $\text{sign}(x)$ . Funkcję tę opisuje zależność:  $\text{sign}(x > 0) = 1$ ,  $\text{sign}(x < 0) = -1$  oraz  $\text{sign}(0) = 0$ . Można też przyjąć prostszy wariant:  $\text{sign}(x \geq 0) = 1$  oraz  $\text{sign}(x < 0) = -1$ .

Kod implementujący funkcjonalność perceptronu pokazano na Listingu 1.

### Listing 1. Podstawowy kod perceptronu

```
public class Perceptron{
    private double[] weights;

    public double compute(double[] x){
        double sum = 0.0;
        for(int i = 0; i < x.length; i++){
            sum += x[i] * weights[i];
        }
        sum += weights[x.length + 1];
        return (sum > 0) ? 1 : -1;
    }
}
```

# TKH Development Center



praca z pasją

[kadry@cctechnology.pl](mailto:kadry@cctechnology.pl)

Jak widać, kod symulujący działanie perceptronu jest bardzo prosty. Oczywiście w tym artykule skupiamy się na algorytmach, a nie zabezpieczaniu kodu przed niezdefiniowanym zachowaniem. W docelowym programie należy dopisać mechanizmy obsługujące sytuacje wyjątkowe, takie jak przekazanie zbyt długiej lub zbyt krótkiej tablicy danych wejściowych itp.

## Uczenie sieci neuronowych

Nasza jednokomórkowa sieć neuronowa potrafi już liczyć. Niestety na niewiele się przyda, ponieważ nie wyznaczyliśmy jeszcze wartości wag, które są niezbędne do obliczeń. Musimy nauczyć perceptron, jak prawidłowo odpowiadać na stawiane przez nas pytania. Są różne rodzaje uczenia sieci neuronowych:

- » Uczenie pod nadzorem. Nadzorca (zwany też nauczycielem) posiada zestaw przykładowych danych wraz z prawidłowymi odpowiedziami. Perceptron wykonuje swoje działania na tych danych, a w przypadku zwarcia błędnych odpowiedzi jego wag są modyfikowane zgodnie z algorytmem uczenia. Początkowe wartości wag są przypadkowe. Ten algorytm jest najmniej złożony obliczeniowo, dlatego zostanie zastosowany w dalszych rozważaniach.
- » Uczenie przez wzmacnianie. Jest to odmiana uczenia pod nadzorem, jednak w odróżnieniu od poprzedniej nie jest znana dokładna wartość oczekiwanej odpowiedzi. Wiadome jest jedynie, czy udzielona odpowiedź jest prawidłowa lub nie. Sieć neuronowa buduje w ten sposób bazę wiedzy na temat środowiska, w którym pracuje.
- » Uczenie bez nadzoru. Jest to przypadek, gdy nie są w ogóle znane odpowiedzi na dane wymuszenia. Sieć neuronowa koryguje swoje wagę w ten sposób, by neurony aktywowane pewnymi wzorcami sygnałów wejściowych jeszcze mocniej na nie reagowały. Jeśli odpowiedź danego neuronu była silnie pozytywna, jego wagę są korygowane tak, aby jeszcze silniej reagował. Podobnie z neuronem, którego odpowiedź była silnie negatywna. Takie samouczające się sieci bardzo dobrze wykrywają wzorce i powtarzające się sekwencje danych ukryte w sygnałach wejściowych.

## Przygotowania

Przed przystąpieniem do pracy nad mechanizmem uczenia naszej jednokomórkowej sieci musimy przygotować środowisko do testów. Aby dobrze zobrazować wyniki prac, utworzymy prosty interfejs operatora, który w konsoli systemowej wyświetli pseudo-graficzną reprezentację wartości referencyjnych stosowanych w omawianym algorytmie oraz wyniki pracy sieci. Dla uproszczenia przyjmijmy, że perceptron posiada dwa wejścia, dzięki czemu dane wejściowe będzie można zaprezentować na płaszczyźnie X-Y, a wyniki jako liczby. Dodatkowo dane wejściowe będą w zakresie [-1; 1], a możliwe odpowiedzi to {0, 1}.

Na początek zdefiniujmy interfejs, który będzie wspólny zarówno dla perceptronu, jak i modułu wyświetlającego wyniki pracy (Listing 2). „Odpowiadacz” składa się z dwóch funkcji – udzielającej odpowiedzi na zadane sygnały wejściowe oraz zwracającej nazwę danego modułu.

### Listing 2. Interfejs perceptronu oraz wyświetlacz

```
public interface IAnswerer {
    public double answer(double[] x);
    public String getName();
}
```

Teraz przyszedł czas na moduł wyświetlacza. Będzie można w nim rejestrować obiekty implementujące interfejs „odpowiadacza”. Po wywołaniu odpowiedniej metody w konsoli pojawi się zestawienie płaszczyzn X-Y wszystkich zarejestrowanych obiektów, z wyszczególnieniem odpowiedzi, których każdy z nich udzielił (Listing 3). Klasa ConsolePrinter posiada publiczne metody: addAnswerer oraz printResults. Pierwsza z nich rejestruje obiekt, który implementuje omawiany wyżej interfejs, natomiast druga drukuje wyniki w konsoli programu. O ile działanie pierwszej nie jest zagadkowe, o tyle druga z nich wymaga nieco komentarza.

Najważniejsze rzeczy dzieją się w podfunkcji printAnswerFields. W tej metodzie znajdują się trzy pętle, kolejno: argument Y (oś pionowa), referencja danego obiektu, który ma udzielać odpowiedzi, oraz pętla argumentu X (oś pozioma). Dzięki takiej organizacji kodu płaszczyzny odpowiedzi wszystkich zarejestrowanych obiektów będą w jednej linii na ekranie. Struktura kodu pozwala uniknąć używania sekwencji sterujących przesuwającymi kursor na daną pozycję, co niekoniecznie może działać we wszystkich rozdajach konsoli systemowych. W samym wnętrzu pętli znajduje się wywołanie funkcji printSingleResult, do której przesyłana jest referencja bieżącego obiektu oraz bieżący punkt na płaszczyźnie odpowiedzi. Wewnątrz tej metody wywołane jest obliczanie wartości w danym obiekcie oraz formatowanie odpowiedzi. Zastosowano tutaj sekwencje sterujące konsolą zgodne ze standardem terminala VT100, które są powszechnie implementowane w konsolach systemowych (przynajmniej te odnoszące się do sterowania kolorami). Jeśli odpowiedź jest nieujemna, drukowana jest zielona jedynka, w przeciwnym wypadku czerwone zero.

### Listing 3. Klasa wyświetlacz

```
public class ConsolePrinter {
    private final ArrayList<IAnswerer> answerers = new
    ArrayList<>();

    public void addAnswerer(IAnswerer ans) {
        answerers.add(ans);
    }

    public void printResults() {
        printDivisionLine();
        printPaddedNames();
        printAnswerFields();
    }

    private void printDivisionLine() {
        for (IAnswerer a : answerers) {
            System.out.print(String.format("%-46s", "=").replace(' ', '='));
        }
        System.out.println("");
    }

    private void printAnswerFields() {
        for (double y = 1.0; y > -1.1; y -= 0.1) {
            for (IAnswerer ans : answerers) {
                for (double x = -1.0; x < 1.0; x += 0.1) {
                    printSingleAnswer(ans, x, y);
                }
                System.out.print("    ");
            }
            System.out.println("");
        }
    }

    private void printSingleAnswer(IAnswerer ans, double x, double y) {
        double[] d = new double[2];
        d[0] = x;
        d[1] = y;
        if (ans.answer(d) > 0) {
            System.out.print("\u033[31m1\u033[39m ");
        } else {
            System.out.print("\u033[32m0\u033[39m ");
        }
    }
}
```

```

private void printPaddedNames() {
    for (IAnswerer ans : answerers) {
        System.out.print(String.format("%-46s", ans.getName()));
    }
    System.out.println("");
}
}

```

Pozostała jeszcze kwestia przygotowania płaszczyzny referencyjnej potrzebnej do uczenia perceptronu. Skoro wartości referencyjne mają być drukowane w konsoli obok odpowiedzi perceptronu, ta klasa również musi implementować interfejs IAnswerer (Listing 4). Klasa ReferenceFrame została napisana w taki sposób, aby móc rejestrować bariery, które określają obszary wartości dodatnich oraz ujemnych na płaszczyźnie X-Y. Dla prostych przypadków, gdzie znajduje się jedna linia prosta, dzieląca płaszczyznę na dwa obszary odpowiedzi, wydaje się to przerostem kodu nad treścią. Niemniej w drugiej części artykułu, gdzie będziemy tworzyć skomplikowane zamknięte obszary odpowiedzi dodatnich, np. wnętrze koła oraz wycinek płaszczyzny ograniczony funkcją kwadratową, omawiana własność będzie nie do przecenienia.

Naturalnie metoda addBarrier pozwala na dodanie obiektu implementującego znajomy nam interfejs i jednocześnie barierę odpowiedzi dodatnich i ujemnych. Metoda genCoord pozwala wygenerować losową wartość z zakresu [-1, 1], co okaże się pomocne podczas uczenia perceptronu. Implementacja odziedziczonej metody answer zawiera w sobie pętlę, która przegląda wszystkie zarejestrowane bariery w poszukiwaniu dodatniej odpowiedzi. Jeśli zadeklarowany punkt (x,y) znalazł się wewnątrz jednej z nich, odpowiedź płaszczyzny referencyjnej jest dodatnia, w przeciwnym razie – ujemna.

#### Listing 4. Klasa płaszczyzny wzorcowej

```

public class ReferenceFrame implements IAnswerer {
    private static final Random R = new Random();
    private final ArrayList<IAnswerer> answerers = new
    ArrayList<>();
    private String name;

    public void addBarrier(IAnswerer ans) {
        answerers.add(ans);
    }

    public ReferenceFrame(String name) {
        this.name = name;
    }

    public double genCoord() {
        return R.nextDouble() * 2 - 1;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public double answer(double[] x) {
        boolean result = false;
        for (IAnswerer ans : answerers) {
            if (ans.answer(x) > 0) {
                return 1;
            }
        }
        return -1;
    }
}

```

Pozostało jeszcze zaimplementować prostą barierę w postaci linii prostej, dzielącej płaszczyznę na dwa obszary (Listing 5). W konstruktorze podawane są współczynniki prostej a, b oraz nazwa bariery. Udzielenie odpowiedzi sprowadza się do obliczenia, czy dla podanych współczynników punkt (x,y) znajduje się ponad bądź też na prostej (odpowiedź dodatnia lub nie (odpowiedź negatywna).

#### Listing 5. Klasa implementująca prostą barierę w postaci linii prostej

```

public class SimpleLine implements IAnswerer {
    private double a, b;
    private String name;

    public SimpleLine(double a, double b, String name) {
        this.a = a;
        this.b = b;
        this.name = name;
    }

    @Override
    public double answer(double[] x) {
        return computeLine(a, x[0], b, x[1]);
    }

    @Override
    public String getName() {
        return name;
    }

    private double computeLine(double a, double x, double b,
    double y) {
        double result = a * x + b;
        return (y > result) ? 1 : -1;
    }
}

```

#### Algorytm uczenia pod nadzorem

Uczenie perceptronu pod nadzorem polega na podawaniu danych wejściowych, obliczeniu przez perceptron odpowiedzi oraz porównanie jej z prawidłową odpowiedzią, której powinien udzielić. Jeśli odpowiedź jest błędna, należy zmodyfikować wartości wag o odrobinę. Całą procedurę należy powtórzyć do momentu, aż perceptron zacznie udzielać (prawie) samych prawidłowych odpowiedzi. Na początku wartości wag powinny być losowe, co przyspieszy proces uczenia (jednakowe wartości wag, np. same zera, mogą w ogóle uniemożliwić proces uczenia).

Powyzszy opis to było „jak”. Teraz pora na „ile”. Modyfikowanie wartości wag o „odrobine” to pojęcie względne, więc czas je sprezyzować. Posłużymy się pojęciem błędu, zaczerpniętym z teorii sterowania. Błąd definiuje się jako różnicę między wartością oczekiwana a wartością otrzymaną. Wartość otrzymana to wynik obliczeń perceptronu. Wartość oczekiwana jest znana, ponieważ została wygenerowana do nauczenia perceptronu. Błąd jest informacją, w jakim stopniu każda z wag powinna być zmodyfikowana. Oznacza to, że nowa wartość wagi jest równa sumie starej wartości oraz pewnemu przyrostowi. Ten przyrost z kolei jest iloczynem błędu, wartości wejściowej odpowiadającej danej wadze oraz stałemu współczynnikiowi. Ten stały współczynnik to tak zwana stała uczenia. Określa dynamikę, jak mocno ma się zmieniać wartość wagi.

W Listingu 6 zawarto kod klasy Perceptron, wraz z mechanizmem treningu. Szczególnie istotna jest metoda train, w której zaszyto wyżej wymieniony algorytm. Nadzorca podaje wektor danych wejściowych, prawidłową odpowiedź oraz współczynnik uczenia, który powinien być w granicach 0.01-0.3 w celu osiągnięcia dobrych rezultatów. Wyliczona wartość adjustedError jest podstawą do modyfikowania wag perceptronu. Ostatnia waga odpowiada za polaryzację perceptronu (*bias*) i jako że jest domyślnie mnożona przez wartość 1, podczas uczenia jest modyfikowana wyłącznie przez adjustedError.

#### Listing 6. Ostateczna wersja klasy perceptronu.

```

public class Perceptron implements IAnswerer {
    private static final Random R = new Random();
    private static int cnt = 0;

```

## BIBLIOTEKI I NARZĘDZIA

```

private final String name;
private final double[] weights;

public Perceptron() {
    name = "Perceptron " + cnt++;
    weights = new double[3];
    generateWeigthValues();
}

public Perceptron(int size) {
    name = "Perceptron " + cnt++;
    weights = new double[size + 1];
    generateWeigthValues();
}

private void generateWeigthValues() {
    for (int i = 0; i < weights.length; i++) {
        weights[i] = R.nextDouble() * 2 - 1;
    }
}

public void train(double[] x, double expectedAnswer, double learnRate) {
    double adjustedError = (expectedAnswer - answer(x)) *
    learnRate;
    for (int i = 0; i < weights.length - 1; i++) {
        weights[i] += adjustedError * x[i];
    }
    weights[weights.length - 1] += adjustedError;
}

@Override
public double answer(double[] x) {
    double sum = 0;
    for (int i = 0; i < weights.length - 1; i++) {
        sum += x[i] * weights[i];
    }
    sum += weights[weights.length - 1];
    return activate(sum);
}

@Override
public String getName() {
    return name;
}

```

---

```

private double activate(double sum) {
    return (sum > 0) ? 1 : -1;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder(name + ": ");
    for (double d : weights) {
        sb.append(d).append(" ");
    }
    return sb.toString();
}

```

Cały powyższy kod jest niewiele wart bez solidnego przykładu. W Listingu 7 zawarto kod głównej funkcji programu, w której trenowane są dwa perceptrony, dla dwóch różnych płaszczyzn referencyjnych. Trenowanie odbywa się metodą Monte Carlo, czyli są generowane losowe pary współrzędnych ( $x,y$ ), które dla próby 5000 powtórzeń pokryją większość powierzchni płaszczyzn referencyjnych. Warunkiem końca jest liczba powtórzeń, ale nic nie stoi na przeszkodzie, aby warunkiem końca był np. procent prawidłowych odpowiedzi większy niż 99%. To zagadnienie pozostawiam do rozwiązania przez czytelnika. Wyniki działania programu pokazano na Rysunku 3. Jak widać, perceptrony bardzo szybko nauczyły się prawidłowo rozpoznawać obszary, w których się znajdują. Co ciekawe, same perceptrony „nie mają pojęcia”, co to jest linia prosta, która dzieli obszar na dwie części, jakie są jej współczynniki ani nawet, że dane wejściowe to współrzędne punktu. Zamiast dwóch liczb, danymi wejściowymi może być np. wektor 128 bitów, reprezentujący dane wejściowe czujnika kolorów na taśmie produkcyjnej. Jeśli tylko proces uczenia jest prawidłowo przeprowadzony, nawet pojedynczy perceptron może służyć do podejmowania szybkich i niezawodnych, choć na pewno prostych decyzji.



Rysunek 3. Wyniki pracy programu

# Zintegruj własną aplikację z SMSAPI

Skorzystaj z gotowych bibliotek w językach:



Załącz konto firmowe z kodem polecenia i odbierz pakiet SMS-ów na przetestowanie naszych usług:

FORMULARZ REJESTRACYJNY:

[www.smsapi.pl/rejestracja](http://www.smsapi.pl/rejestracja)

KOD POLECENIA:

BONUS:

**PR56 500 SMS-ÓW**

**SMSAPI**



### **Listing 7. Główna funkcja programu**

```
public class SimpleNN {
    public static void main(String[] args) {
        ConsolePrinter cp = new ConsolePrinter();

        ReferenceFrame rf1 = new ReferenceFrame("Referencja");
        rf1.addBarrier(new SimpleLine(1, 0, "linia do góry"));

        ReferenceFrame rf2 = new ReferenceFrame("Inna referencja");
        rf2.addBarrier(new SimpleLine(-0.5, 0.5, "linia lekko do dołu"));

        Perceptron p1 = new Perceptron();
        Perceptron p2 = new Perceptron();

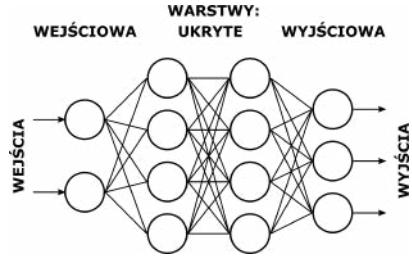
        cp.addAnswerer(rf1);
        cp.addAnswerer(p1);
        cp.addAnswerer(rf2);
        cp.addAnswerer(p2);
        cp.printResults();

        for (int i = 0; i < 5000; i++) {
            double[] d = new double[2];
            d[0] = rf1.genCoord();
            d[1] = rf1.genCoord();
            double a1 = rf1.answer(d), a2 = rf2.answer(d);
            p1.train(d, a1, 0.01);
            p2.train(d, a2, 0.01);
        }
        cp.printResults();
    }
}
```

# SIECI NEURONOWE WIELOPOZIOMOWE – PROBLEM UCZENIA

Proste jednokierunkowe sieci neuronowe (czyli takie bez sprzężenia zwrotnych, struktur imitujących pamięć itp.) mają strukturę warstwową. Wyróżnia się warstwy: wejściową, ukryte (jedna lub więcej) oraz wyjściową (Rysunek 4). Warstwa wejściowa służy kondycjonowaniu sygnałów przychodzących ze środowiska. Częstą operacją wykonywaną przez tę warstwę jest konwersja typów danych (np. konwersja liczb stałoprzecinkowych na zmiennoprzecinkowe w zakresie  $[-1, 1]$ ). Warstwa wyjściowa przeprowadza często proces odwrotny, czyli dostosowanie odpowiedzi do potrzeb elementów odbiorczych ze środowiska. Perceptrony wejściowe mają zwykle po jednym wejściu. Natomiast właściwe przetwarzanie odbywa się w warstwach ukrytych oraz warstwie wyjściowej.

Wiadomo jest, że liczba warstw sieci pozwala określić, jakiego stopnia złożoności może być dane zagadnienie. Dobrą ilustracją będzie poznana wcześniej płaszczyzna referencyjna. Prosta jednowarstwowa sieć potrafi podzielić płaszczyznę na dwie części linią prostą. Dwuwarstwowa potrafi rozwiązać problem typu XOR (trzy ciągłe obszary oddzielone dwiema prostymi od siebie). Dokładając kolejną warstwę, sieć będzie potrafiła wyodrębnić koło. Czterowarstwowa sieć rozpozna prawidłowo przestrzeń z kołem oraz drugim kołem w środku itp. (Rysunek 5). Dokładanie kolej-



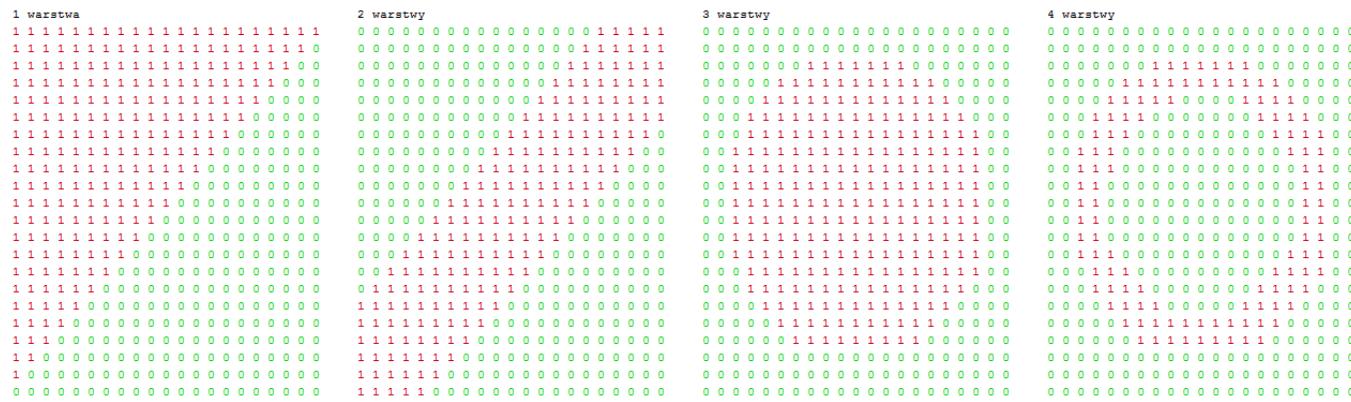
Rysunek 4. Wielowarstwowa sieć neuronowa

nych warstw pozwala rozwiązywać coraz bardziej skomplikowane problemy. Niestety, prosty algorytm uczenia poznany wcześniej nie wystarczy. Problem jest związany z tym, że dane wejściowe perceptronów ukrytych nie są bezpośrednio danymi wejściowymi, w oparciu o które wylicza się poprawki wag. Jednym z mechanizmów implementowanych w uczeniu wielowarstwowych sieci neuronowych jest mechanizm wstępnej propagacji błędów. Jest to skuteczny, ale trudny w implementacji algorytm uczenia, angażujący rachunek całkowy. Jego analiza wykracza poza ramy tego artykułu, w którym chcę przedstawić proste sposoby implementowania metod sztucznej inteligencji w swoich programach.

Proponuję całkowicie inne podejście do szkolenia sieci neuronowych. Zastanówmy się przez chwilę, jak działają zwierzęta. Zwykle przedstawiciele jednego gatunku żyą razem, w populacjach, stawiając czoła nieprzyjaznemu otoczeniu. Te silne i sprytne (czytaj: między innymi posiadające lepiej dopasowane do środowiska mózgi) przeżywają i po pewnym czasie mają potomstwo. Po wielu pokoleniach pewne własności zaczynają dominować w populacjach, stając się nowym standardem w gatunku. Gdyby tak hodować populację prostych organizmów-sieci neuronowych, których zadaniem byłoby rozwiązywanie danego zagadnienia, to stosując mechanizmy znane z teorii ewolucji, można by wyhodować sobie sieć najlepiej przystosowaną. Jednak aby móc tak zrobić, musimy poznać metodę implementowania mechanizmów ewolucyjnych.

# PODSTAWY PROGRAMOWANIA GENETYCZNEGO

Algorytmy genetyczne znajdują zastosowanie w sytuacjach, gdzie niemożliwe jest wyprowadzenie wzoru lub metody analitycznej do wyliczenia wyniku. Może być to związane z niewystarczającymi danymi testowymi (lub ich brakiem, jeśli środowisko docelowe jest zmienne), ograniczeniami aparatu matematycznego lub ograniczonymi zasobami pamięciowymi, mocy obliczeniowej lub czasu. Z drugiej strony szukanie rozwiązania metodą brute-force też okazuje się mało skuteczne bez wiedzy heurystycznej.



Rysunek 5. Możliwe stopnie skomplikowania przestrzeni problemu w zależności od liczby warstw sieci

W niniejszej metodzie obliczeniowej podstawą jest darwinowska teoria ewolucji. Zakłada ona trzy główne cele.

Jeśli symulacja spełnia poniższe założenia, z każdą populacją osobnik będą lepiej dopasowane do wymagań programu:

- » Dziedziczenie. Kolejne pokolenia reprezentują cechy swoich rodziców. Rodzice w wyniku reprodukcji przekazują zmieszane geny swoim dzieciom. Jeśli w bieżącym pokoleniu dany osobnik przeżyje wystarczająco długo, sam staje się jednym z rodziców i przekazuje swoje geny potomstwu.
- » Różnorodność. W obrębie populacji występuje zróżnicowanie w występowaniu danych cech. Bez różnorodności kolejne pokolenia będą jedynie kopią poprzednich, bez możliwości dostosowywania się do warunków środowiska przez mieszanie genów podczas reprodukcji.
- » Selekcja naturalna. Do etapu reprodukcji mogą dojść jedynie osobniki wystarczająco dobrze dopasowane (ang. *fitness*) do warunków środowiska. Jeśli dany osobnik jest słabo dostosowany, nie dożyje do reprodukcji, przez co słabe geny nie zostaną przekazane dalej.

Typowy program ewolucyjny składa się z następujących elementów:

- » Klasy osobnika, który posiada genotyp oraz odpowiadający mu fenotyp. Osobnik potrafi reprodukować z innym osobnikiem, żyć (czyli wykonywać swoje zadania) oraz wyliczać swój *fitness*, który będzie podstawą do określenia, czy osobnik przeżyje do etapu reprodukcji.
- » Klasy populacji osobników, która przeprowadzi okres „życia” każdego z osobników, wyliczy *fitness* każdego, uśmierci niedostosowane osobniki, a te dostosowane uszereguje. Na koniec wygeneruje nową populację jako reprodukcję ze starej. Algorytm dobierania partnerów do reprodukcji ma znaczenie przy szybkości zbiegania do uzyskania populacji najbardziej dostosowanych osobników.
- » Funkcji głównej, w której procesy z klasy populacji są wykonywane w pętli, aż do uzyskania zadowalających rezultatów.

Osobnik jest obdarzony fenotypem, czyli zestawem cech. Cechą może być wzrost, średnia waga, kolor oczu, długość kończyn, jedna z wag neuronu itp. Fenotyp ma swoje odzwierciedlenie w genotypie, czyli zestawie genów. Pojedynczy gen zwykle jest liczbą, ale może być też skomplikowaną strukturą danych. Często, w prostych przypadkach, genotyp jest jednocześnie fenotypem, np. wartości wag (genów) sieci neuronowej są po prostu kopiowane jako fenotyp do kodu wyliczającego odpowiedzi (tego, który „żyje” podczas symulacji). Podczas generowania pierwszej populacji każdemu osobnikowi przyporządkowywane są losowe wartości genów. Jest to z jednej strony największe możliwe zróżnicowanie cech w populacji. Z drugiej strony losowe geny zwykle oznaczają najgorsze dopasowanie do środowiska.

W kolejnych pokoleniach osobnik reprodukuje z innym osobnikiem. Najprostszy przypadek zakładający mieszanie genów to dwa różne osobniki (osobnik niekoniecznie reprodukuje sam ze sobą). Podczas procesu wybierany jest punkt siodłowy (ang: *pivot*), który określa, ile genów przekaże jeden osobnik, a ile drugi. Jednym z wariantów jest mieszanie z jednym punktem siodłowym, gdzie w pętli kopiowane są geny najpierw pierwszego osobnika (od początku wektora genów do punktu *pivot*), a następnie drugie (od punktu *pivot* do końca wektora genów). Podczas reprodukcji powinny sporadycznie wydarzyć się mutacje. Pomimo że mutacje kojarzą się z czymś złym, to mają dobryczynny wpływ na wynik symulacji. Przypuśćmy, że po wielu pokoleniach osobniki są w miarę

dostosowane do środowiska, ale nie mogą tej sytuacji dalej poprawić. Populacja osiągnęła tzw. minimum lokalne i ze względu na brak mutacji nigdy nie wystąpi taki gen, który mógłby wprowadzić populację z tego stanu. Są tylko kolejne mieszania podobnego zestawu genów. Mutacja pojedynczego genu może pozwolić na uzyskanie krytycznej kombinacji genów, ustawiającej populację na dalszy rozwój. Oczywiście mutacje nie powinny być zbyt częste.

Osobnik musi żyć. Życie zwykle oznacza wykonanie właściwej części programu. Jeśli osobnikiem jest sieć neuronowa, to żyjąc, przeprowadza obliczenia na treningowym zestawie danych. Jeśli osobnik reprezentuje tzw. problem plecakowy, żyjąc, próbuje wykonać określone zadania w kolejności wskazanej przez geny oraz obliczyć koszt tych operacji. Na koniec osobnik wylicza swoje dopasowanie do środowiska. Funkcja dopasowania może mieć różnych format wyniku, np. wartości ujemne, z zerem jako najlepszym dopasowaniem. Może być format procentowy lub liczb dodatnich, również malejących. Im większa liczba, tym gorsze dopasowanie. Wtedy *fitness* jest miarą niedopasowania.

### Przykład

Zaimplementowany zostanie prosty model ewolucyjny. Otóż niech każdy z osobników reprezentuje sobą pewien napis (String). Zadaniem symulacji będzie wyhodowanie populacji osobników z prawidłowym napisem. Ten prosty przykład nie tyle jest użyteczny co edukacyjny – bezpośrednio obrazuje omawiane wyżej zagadnienia.

Pierwszym krokiem jest zdefiniowanie klasy osobnika (Listing 8). Klasa *Creature* zawiera w sobie wszystkie mechanizmy, które żywego organizm mieć powinien. Genotyp osobnika jest jednocześnie jego fenotypem, co upraszcza kod mapujący geny na zachowanie podczas „życia”. Zdefiniowano kolejno dwa konstruktory. Pierwszy jest wywoływany wyłącznie na początku symulacji, gdzie każdy z osobników jest inicjalizowany losowymi wartościami genów. Wektor genów w tym przypadku jest reprezentowany przez klasę *String*, dla wygody operowania ciągami znaków. Domyslnie wektor genów dobrze jest reprezentować przy pomocy kontenera, np. *ArrayList*, lub po prostu tablicy.

Metoda *generateRandomGene* generuje losowy znak ASCII z przedziału  $32_{10}$  (spacja) do  $126_{10}$  (tylda, ostatni znak przed DELETE). Pozbywając się przedziału znaków sterujących, mamy możliwość zobaczyć zawartość genotypu w konsoli, w nienaruszonej formie. Genotyp jest tej samej długości co zdefiniowany napis docelowy - TARGET. To pole jest statyczne, przez co wszystkie osobniki będą miały do niego bezpośredni dostęp.

Drugi konstruktor odpowiada za reprodukcję. Pobiera przez referencje informacje o genach rodziców nowego osobnika. Na podstawie długości wektora genów wyznaczany jest punkt siodłowy. Następnie, w metodzie *reproduceFromTwoParents*, następuje utworzenie nowego wektora genów, zgodnie z wyznaczonym punktem siodłowym. Dodatkowo przeprowadzany jest test, czy ma zajść mutacja genu, czy nie. Jeśli tak, to wybierana jest pozycja mutacji oraz nowa wartość genu.

Kolejną istotną funkcją jest *live*. W niej osobnik „żyje”. Omawiany typ osobnika ma niewiele do wykonania w swoim krótkim życiu. Jego głównym zadaniem jest wyznaczenie, jak dobrze jest dopasowany do środowiska. Dopasowanie określone jest przez procentową zgodność genotypu/fenotypu ze wzorcem. Jeśli na danej pozycji znaki genotypu i wzorca są jednakowe, licznik jest inkrementowany, na końcu normalizowany do długości genotypu, przez co wynik jest w przedziale  $[0, 1]$ . Funkcja *getFitness* zwraca wyznaczoną wartość.

Skoro osobniki mają żyć w populacji, istnieje potrzeba ich sortowania ze względu na stopień dopasowania do środowiska.

# BIBLIOTEKI I NARZĘDZIA

Na początku posortowanej listy powinny znaleźć się osobniki najlepiej dopasowane, natomiast na końcu – te najgorzej dopasowane. Z tego względu Creature implementuje interfejs szablonowy Comparable<T>. Pozwala to na zaimplementowanie mechanizmów sortujących na poziomie klasy Creature.

**Listing 8. Klasa Creature implementująca zachowanie pojedynczego osobnika**

```
public class Creature implements Comparable<Creature> {
    private static final Random R = new Random();
    private final String genes;
    private static String TARGET = „Witaj swiecie w moim programie do symulacji zycia - Grzegorz Grzeda”;
    private static double mutationRate = 0.02;
    private double fitness;
    private boolean hasNotLivedYet = true;

    public Creature() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < TARGET.length(); i++) {
            sb.append(generateRandomGene());
        }
        genes = sb.toString();
    }

    public Creature(Creature parent1, Creature parent2) {
        int pivot = R.nextInt(parent1.genes.length() + 1);
        genes = reproduceFromTwoParents(parent1, parent2, pivot);
    }

    private String reproduceFromTwoParents(Creature parent1,
    Creature parent2, int pivot) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < parent1.genes.length(); i++) {
            if (i < pivot) {
                sb.append(parent1.genes.charAt(i));
            } else {
                sb.append(parent2.genes.charAt(i));
            }
        }
        if (R.nextDouble() < mutationRate) {
            sb.setCharAt(R.nextInt(parent1.genes.length()),
            generateRandomGene());
        }
        return sb.toString();
    }

    public static void setTARGET(String TARGET) {
        Creature.TARGET = TARGET;
    }

    public static void setMutationRate(double mutRate) {
        mutRate = (mutRate < 0.0) ? 0.0 : ((mutRate > 1.0) ? 1.0 :
        mutRate);
        Creature.mutationRate = mutRate;
    }

    public void live() {
        fitness = 0.0;
        for (int i = 0; i < genes.length(); i++) {
            if (genes.charAt(i) == TARGET.charAt(i)) {
                fitness += 1.0;
            }
        }
        fitness /= genes.length();
        hasNotLivedYet = false;
    }

    public double getFitness() {
        if (hasNotLivedYet) {
            live();
        }
        return fitness;
    }

    private char generateRandomGene() {
        int min = 32;
        int max = 127;
        char c = (char) (R.nextInt(max - min) + min);
        return c;
    }

    @Override
    public String toString() {
        if (hasNotLivedYet) {
            live();
        }
        return genes + " - fit: " + fitness;
    }

    @Override
    public int compareTo(Creature o) {
        if (fitness < o.fitness) {
            return -1;
        } else if (fitness > o.fitness) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

```
        return -Double.compare(fitness, o.fitness);
    }
}
```

Po zakodowaniu osobnika czas na populację osobników (Listing 9). W konstruktorze podawana jest liczba osobników przypadająca na każde pokolenie. W tym miejscu populacja jest uzupełniana osobnikami z losowymi wartościami genów. Funkcja odpowiadająca za „życie” populacji – live – w pętli przegląda listę osobników, kolejno je ożywiając. Na koniec sortowane są pod względem fitnessu. Funkcja getPercentFitted zwraca informację, ile procent osobników dopasowało się do środowiska. Informacja ta zwykle jest jednym z warunków ukończenia hodowli. Przeładowana funkcja toString zwraca sformatowany skrót na temat najlepszych pięciu osobników w danym pokoleniu.

**Listing 9. Klasa implementująca populację osobników**

```
public class Population {
    private static final Random R = new Random();
    private ArrayList<Creature> population = new ArrayList<>();
    private int counter = 0;

    public Population(int size) {
        for (int i = 0; i < size; i++) {
            population.add(new Creature());
        }
    }

    public void live() {
        for (Creature c : population) {
            c.live();
        }
        population.sort(null);
        counter++;
    }

    public void reproduce() {
        ArrayList<Integer> mateNumbers = new ArrayList<>();
        for (int i = 0; i < population.size() / 2; i++) {
            int n = (int) (population.get(i).getFitness() * 100);
            for (int j = 0; j < n; j++) {
                mateNumbers.add(i);
            }
        }
        ArrayList<Creature> newPopulation = new ArrayList<>();
        for (int i = 0; i < population.size(); i++) {
            int a = mateNumbers.get(R.nextInt(mateNumbers.size()));
            int b = mateNumbers.get(R.nextInt(mateNumbers.size()));
            newPopulation.add(new Creature(population.get(a),
            population.get(b)));
        }
        population = newPopulation;
    }

    public double getPercentFitted() {
        double cnt = 0;
        for (Creature c : population) {
            if (c.getFitness() > 0.999) {
                cnt += 1.0;
            }
        }
        return cnt / population.size();
    }

    @Override
    public String toString() {
        int size = (population.size() > 5) ? 5 : population.size();
        StringBuilder sb = new StringBuilder();
        sb.append("===== \n");
        sb.append("Generation: ").append(counter).append("\n");
        for (int i = 0; i < size; i++) {
            sb.append(population.get(i)).append("\n");
        }
        return sb.toString();
    }
}
```

Osobny komentarz należy się funkcji reproduce. W niej zachodzi proces doboru naturalnego oraz dobierania rodziców dla osobników przyszłego pokolenia. Funkcja ta realizuje prosty system promowania bardziej dostosowanych osobników na rzecz mniej dostosowanych. Na początku tworzona jest pusta lista numerów

osobników. Następnie w pętli przegląda się funkcje fitnessu każdego z osobników z pierwszej połowy populacji. Druga połowa (ta mniej dostosowana) „nie przeżyła” do etapu reprodukcji, a ich słabe geny przepadają. Dla każdego z osobników biorących udział w reprodukcji wstawiana jest taka liczba odpowiadająca jego numerowi, ile wynosi procentowy fitness. Jeśli np. populacja liczy 10 osobników, 5 pierwszych jest branych pod uwagę oraz mają fitness odpowiednio: 0.8, 0.7, 0.6, 0.6, 0.5, to lista numerów będzie następująca: 80 numerów „1”, 70 numerów „2”, 60 numerów „3”, 60 numerów „4” oraz 50 numerów „5”.

W kolejnym etapie następuje losowanie numerów osobników biorących udział w reprodukcji. Powyższy proces generowania numerów powoduje, że najlepiej dopasowany osobnik ma największą szansę na reprodukcję. To spowoduje, że jego geny zdominują następne pokolenie. W tym prostym przykładzie może się zdarzyć, że osobnik będzie reprodukował sam ze sobą. Jednak ze względu na występowanie mutacji sytuacja ta nie jest szkodliwa. Oczywiście proces losowania i reprodukcji wykonywany jest w pętli tyle razy, ile docelowo ma liczyć kolejne pokolenie.

Na koniec została główna funkcja programu (Listing 10). Po utworzeniu populacji 1000 osobników uruchomiona zostaje pętla, w której każde pokolenie żyje i reprodukuje. Co dziesięć pokoleń wyświetlane jest podsumowanie. Pętla kończy bieg w momencie, gdy ponad 10% populacji dostosowało się do środowiska, co w tym przypadku oznacza odtworzenie wzorcowego napisu.

#### Listing 10. Główna funkcja programu

```
public class SimpleGA {
    public static void main(String[] args) {
        Population p = new Population(1000);
        for (int i = 0; true; i++) {
            if (i % 10 == 0) {
                System.out.println(p);
            }
            p.live();
            p.reproduce();
            if (p.getPercentFitted() > 0.1) {
                break;
            }
        }
        p.live();
        System.out.println(p);
    }
}
```

W Listingu 11 zawarto skrót wyniku pracy zapisanego w konsoli. Zauważmy, jak samoczynnie i wręcz „niespodziewanie” populacja ewoluje z masy pseudolosowych znaków do poszukiwanego wzorca. Warto również zauważyć, że osiągnięcie możliwe dobrego wyniku

#### Listing 11. Wynik pracy symulatora

```
Generation: 0
e;@&L8c{E;fE4F"A!7Wu4y|g1gl3@EwZ6/DRPB ScP$?*ywKXldNa8->_!=0&*2RHx - fit: 0.0298
BHAm>HO<V#B^r0wn="6IU^0|6(NX%08C^Pr37(R@[{YQ4IEz'[2s]NFoj$umU_V0CY - fit: 0.0149
qcJ6<TQZ\").z{+f;k[ib0?yj]a5{Un~noa`eu'|W>V#De<rP-%'r^9,z_!MjBy9 - fit: 0.0
kpJb54Q<68LE:to"/W~|2FH9JNn\o~WF$,U{twnT4|<ss$}d2@Y#Nk@>G>7@vQL - fit: 0.0
0EB)>HjZ[q>UL6(>v&0Vdi)5134"GMqDyOD!>!=@Vbm@o~c:au.:ue@Wh3ZgjwBW~G - fit: 0.0

Generation: 10
j&uajdvdfoec1bV R*k1-8V/og]\rMs3x6X)yI4pBzyj@1|cf1Za q2?NQ'@5VGszVdr - fit: 0.1940
juajdvdfoec1bVdkp,]] P/~Ühz6H1Cd-MBjmtH-mj| OB<Z '8G1..,0EN]aGrPuM= - fit: 0.1940
%i*qS D1iig8o O ?vP#4V/og)a&z5NNo_mR4u~vR1eq^A1:0.3R 6Y>o*~uGrQ0 \ - fit: 0.1940
j&ajdD1iiNm~^O +@i-8W56z}sm1aydYZ7`')~}+pWjZ\1\w;a q2?NQ'@5VGszVdr - fit: 0.1791
jua" D1iic1bV! ?vU#4V/og]\rMs3x6X)yI41Vqj$;ysaUaCG16)9M;ENyuGrPuM= - fit: 0.1791

Generation: 20
ji*aS D1iicn0 HszoIF V/og]a&zeNNo:_y6u~vNj| DB1\w ,8Glz,0E<yuGrzVdr= - fit: 0.3731
%iuajjdD1iecnC HswoiF V/og]a&zeExo_~y6u~vNj| OB^iZ a q2?,g}@yuGrPuM= - fit: 0.3731
%iuia" D10ecnC HszoI-8pprogram1aydo_mR4u~}g:j:Jz\!]Z ,8GVz,0ENyuGrPuM= - fit: 0.3432
%iuajdv1ieNm~ Hszo7F V/og]a&zeCdt^gmtH-Nj| OB^iZ a GVz,0ENyuGrPuM= - fit: 0.3432
%i*ajdv1ieNm~ HszoI-8V/ogram1efr]h:y6u~vNj| \1\w;a $Nz,0ENyuGrPVdr - fit: 0.3432

Generation: 30
%iuaj D1iecmo HszoIF VrograRNeCd7-^gmtH-Nj| zy_la /8GVzzgENyuGrzVdr - fit: 0.4776
...
Generation: 140
fiuaj D1iecmo H zoim qrogramzeCdoGsymu?auj| zy_ia a Grz,g}@yuGrzedr - fit: 0.6716
...
Generation: 220
%iuaj swiec1e H moim qrogram1eCdo$symu~auj| zycia - Grz,g}@yuGrzedr - fit: 0.7462
...
Generation: 350
%iuaj swiec1e w moim qrogram1eCdo symulauj| zycia - Grzeg}ryuGrzeda - fit: 0.8358
...
Generation: 510
%iuaj swiec1e w moim qrogram1eCdo symulacji zycia - Grzeg}ryuGrzeda - fit: 0.8955
...
Generation: 1380
Witaj swiec1e w moim programie do symulacji zycia - Grzegorz Grzeda - fit: 1.0
```

zajęło około 500 pokoleń, natomiast wypracowanie dokładnego rozwiązania zajęło około 1300 pokoleń. To ujawnia problem minimum lokalnego. Po 500 pokoleniach zostały osobniki z dobrym zestawem genów. Spadła różnorodność między osobnikami, co spowodowało, że reprodukcja nawet z losowym punktem siodłowym nie przynosiła rezultatu. Potrzeba było wielu mutacji, aby ewolucja postępowała dalej. Współczynnik mutacji był ustalony na 0.02, co oznacza 2% szans na mutację pojedynczego genu przy każdej reprodukcji. Jeśli współczynnik ten byłby za duży, np. 15%, ewolucja mogłaby w ogóle nie postępować, ponieważ zbyt wiele losowych wartości zaburzyłyby naturalny proces przekazywania dobrych genów.

## PODSUMOWANIE

Dotarliśmy do końca tego artykułu. Poznaliśmy podstawy sieci neuronowych oraz algorytmów genetycznych. Śmiało możemy rozwijać przedstawione wyżej pomysły, dostosowując je do swoich potrzeb. W następnej części przedstawię połączenie algorytmów genetycznych oraz sieci neuronowych, tworząc jednolity mechanizm do rozwiązywania różnych złożonych problemów. Opowiem również o możliwych ścieżkach optymalizacji kodu oraz uczynienia go bardziej przenośnym i uniwersalnym.



### GRZEGORZ GRZEDA

grzegorz.grzeda@gmail.com

Elektronik i programista. Pasjonuje się projektowaniem zintegrowanych układów elektronicznych w strukturach FPGA oraz programowaniem mikrokontrolerów. Nie wstydzi się programowania małych procesorów 8-bitowych, nie obawia się programowania układów SoC. Zwolennik otwartego oprogramowania oraz systemów unixowych. Obecnie doskonali techniki zwinnego pisania oprogramowania oraz zgłębia zagadnienia związane ze sztuczną inteligencją. Relaksuje się jazdą motocyklem.

# Pakiet MyHDL

## Python w służbie sprzętu i układów FPGA

Zastosowania Pythona są niepoliczalne. Można dłużej wymieniać przykłady pakietów, bibliotek i innych przypadków użycia tego języka (a raczej powinniśmy mówić środowiska/ekosystemu). Nie powinno zatem dziwić, iż w Pythonie można także projektować rozwiązania dla sprzętu z generacją kodu dla języków VHDL lub Verilog.

Od dłuższego czasu dostępna jest biblioteka o nazwie MyHDL, która pozwala na translację kodu napisanego w Pythonie na kod przeznaczony m.in. dla układów FPGA. Co oczywiste, możemy też korzystać z wysokopoziomowych rozwiązań tego języka. Jednak po uruchomieniu programu z pakietem MyHDL jako wynik otrzymujemy kod we wspominanych już językach VHDL oraz Verilog, zawierający implementację naszego zadania.

W artykule pokażemy przykładowe rozwiązania opracowane w MyHDL oraz jego implementację na rzeczywistym układzie FPGA. Naszym głównym zadaniem będzie opracowanie trybu tekstuowego o 80 kolumnach i 30 wierszach. Tryb ten będzie zgodny z rozdzielcością 640x480, a częstotliwość odświeżania to 60 Hz. Natomiast główny kod implementujący generowanie obrazu naturalnie napiszemy w Pythonie. Wykorzystamy też płytę rozwojową Nexys 2, która choć ma już swoje lata, to w zupełności wystarczy do realizacji naszego zadania.

### BRAMKI I SUMATORY NA POCZĄTEK

Na początek przedstawimy bardzo proste przykłady odnoszące się do bramek logicznych oraz sumatora. Naturalnie nie chodzi o podstawy elektroniki cyfrowej, ale o fakt, w jaki sposób można korzystać z pakietu MyHDL.

Listing 1 to implementacja bramki logicznej AND. Zawiera on dwie najważniejsze funkcje. Pierwsza `and_gate` to funkcja główna, zgodnie z nazwą odpowiada za implementację bramki AND. Funkcja ta przyjmuje cztery parametry, gdzie `a` oraz `b` to wejście dla bramki AND, natomiast `c` to naturalnie wyjście. Mamy jeszcze sygnał zegarowy – `clk` – jest to także parametr wejściowy.

Nie musimy także określać, czy dany sygnał lub zmienna jest parametrem wejściowym, czy też wyjściowym. Podczas konwersji pakiet MyHDL wywnioskuje to samodzielnie na podstawie podanego kodu.

Właściwa implementacja bramki AND znajduje się w funkcji `logic`. Ważny jest atrybut przed nazwą funkcji. Znajduje się tam wyrażenie `@myhdl.always(clk.posedge)` oznaczające, że nasza bramka AND działa synchronicznie razem z narastającym sygnałem zegara. Sygnał zegara `clk` naturalnie definiujemy później, bowiem dopiero w projekcie dla układu FPGA, na poziomie Pythona podajemy tylko nazwę tego sygnału.

Właściwa treść implementacji funkcji bramki AND znajduje się w funkcji o nazwie `logic`, to tylko linia kodu:

```
c.next = a and b
```

Ważnym elementem jest pole `next`. Gdy chcemy do zmiennej, czy też sygnału, przypisać nową wartość (zmienna oraz sygnał to różne pojęcia, ale to jeszcze wyjaśnimy w kontekście naszego głównego przykładu), to należy użyć pola `next` w ten podany sposób, aby późniejsza synteza kodu VHDL do implementacji na układzie FPGA potraktowała w sposób właściwy sygnał `c`, tj. by stał się on sygnałem wyjściowym.

Należy zwrócić uwagę, że ostatnią czynnością, jaką wykonujemy w funkcji `and_gate`, jest zwrocenie referencji do funkcji `logic`, odpowiedzialnej za logikę operacji AND.

**Listing 1. Bramka logiczna „i” (tj. AND) w Pythonie i pakiecie MyHDL**

```
#!/usr/bin/python

import myhdl

def and_gate(a, b, c, clk):
    @myhdl.always(clk.posedge)
    def logic():
        c.next = a and b

    return logic

def convert_to_vhdl():
    a, b, c, clk = [myhdl.Signal(bool(0)) for i in range(4)]
    myhdl.toVHDL(and_gate, a, b, c, clk)

convert_to_vhdl()
```

Listing 1 zawiera jeszcze jedną bardzo ważną funkcję o nazwie `convert_to_vhdl`. Zgodnie z nazwą zadaniem tej funkcji jest wykonanie konwersji definicji bramki AND opisanej za pomocą Pythona na odpowiedni kod w VHDLu.

Funkcja ma, jak widać, tylko dwie linie kodu. Linia pierwsza tworzy sygnały za pomocą typu `Signal` i są to wartości standardowej logiki Boola, tj. takie, które przyjmują wartość zero lub jeden. Natomiast właściwą konwersję wykonuje druga linia kodu:

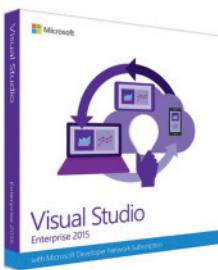
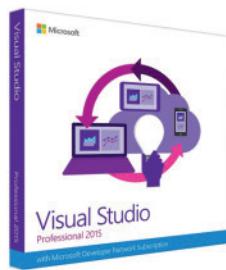
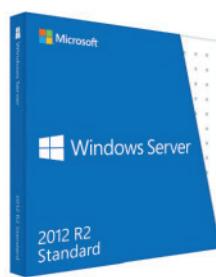
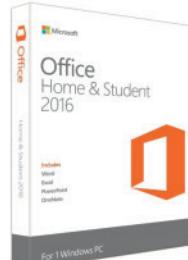
```
myhdl.toVHDL(and_gate, a, b, c, clk)
```

Metoda `toVHDL` wykona konwersję podanych obiektów do kodu VHDL. Obiekty poddawane konwersji w naszym przykładzie to funkcja `and_gate` oraz utworzone wcześniej sygnały. Nazwy sygnałów muszą się zgadzać z nazwami sygnałów w funkcji `and_gate`. Warto dodać, iż jeśli potrzebny jest kod w języku Verilog, to wystarczy użyć metody `toVerilog`.

W Listingu 2 przedstawiono kod utworzony na podstawie przykładu z bramką AND (usunięte zostały tylko puste linie). Jak widać,



**TTS Company rekomenduje oprogramowanie Microsoft ®**



**www.OprogramowanieKomputerowe.pl**

Microsoft Azure

Office 365

OneDrive

Więcej informacji: ☎ (22) 272 94 94 ✉ [sales@tts.com.pl](mailto:sales@tts.com.pl)

Sprzedaż



Dystrybucja



Import na zamówienie

tłumaczenie jest bezpośrednie i pisząc samodzielnie tego typu kod w VHDLu, efekt byłby praktycznie taki sam.

## Listing 2. Kod wygenerowany przez pakiet MyHDL na podstawie przykładu z Listingu 1

```
-- File: and_gate.vhd
-- Generated by MyHDL 0.9.0
-- Date: Sun Feb 19 14:34:10 2017

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

use work.pck_myhdl_090.all;

entity and_gate is
    port (
        a: in std_logic;
        b: in std_logic;
        c: out std_logic;
        clk: in std_logic
    );
end entity and_gate;

architecture MyHDL of and_gate is
begin
    AND_GATE_LOGIC: process (clk) is
    begin
        if rising_edge(clk) then
            c <= stdl(bool(a) and bool(b));
        end if;
    end process AND_GATE_LOGIC;
end architecture MyHDL;
```

## PRZYKŁAD Z PEŁNYM SUMATOREM

Kolejny przykład to realizacja pełnego sumatora, który sprowadza się do dwóch linii kodu z użyciem funkcji logicznych:

```
c.next = ((a ^ b) ^ oldcarry)
carry.next = (a and b) or (a and oldcarry) or (b and oldcarry)
```

Znak  $\wedge$  oznacza funkcję XOR. Podane linie bezpośrednio odnoszą się do implementacji sumatora. Wbrew pozorom reszta implementacji, jak widać w Listingu 3, jest podobna do naszej bramki AND, stosujemy identyczny atrybut `@always(clk.posedge)` oznaczający, iż układ sumatora działa przy narastającym zboczu sygnału zegarowego. W przeciwnieństwie do poprzedniego przykładu tym razem podczas importu pakietu MyHDL zastosowaliśmy wyrażenie:

```
from myhdl import *
```

Co naturalnie oznacza import symboli pakietu do aktualnego zasięgu zmiennych w Pythonie, więc nie trzeba elementów MyHDL poprzedzać nazwą pakietu. Bardziej istotnym elementem, jaki chcemy w tym miejscu omówić, są metody `test_sum_circuit` oraz `simulate`. Te dwie funkcje są wykorzystywane do symulacji działania obwodu, który projektujemy w MyHDLu. Pierwsza z funkcji tworzyinstancję obwodu oraz pomocniczą funkcję `clkgen` do generacji sygnału zegarowego. Tworzoną jest też funkcja `stimulus` odpowiedzialna za generację wartości dla sygnałów wejściowych, w naszym przypadku są to sygnały `a`, `b` oraz `oldcarry`. Sygnały te są generowane za pomocą metody `randrange` z pakietu `random`.

Wygenerowanie danych z symulacji do obejrzenia w programie GtkWave wykonuje funkcja `simulate`, a na Rysunku 1 przedstawiono przykładowe przebiegi, jakie zostały uzyskane za pomocą skryptu z Listingu 3. Podczas wywołania funkcji `simulate` określamy, przez ile nanosekund ma trwać symulacja – podaliśmy 1000, co oznacza, iż symulacja obejmuje jedną mikrosekundę.

## Listing 3. Sumator w pakiecie MyHDL wraz z funkcjami do testów

```
#!/usr/bin/python

import random
from myhdl import *

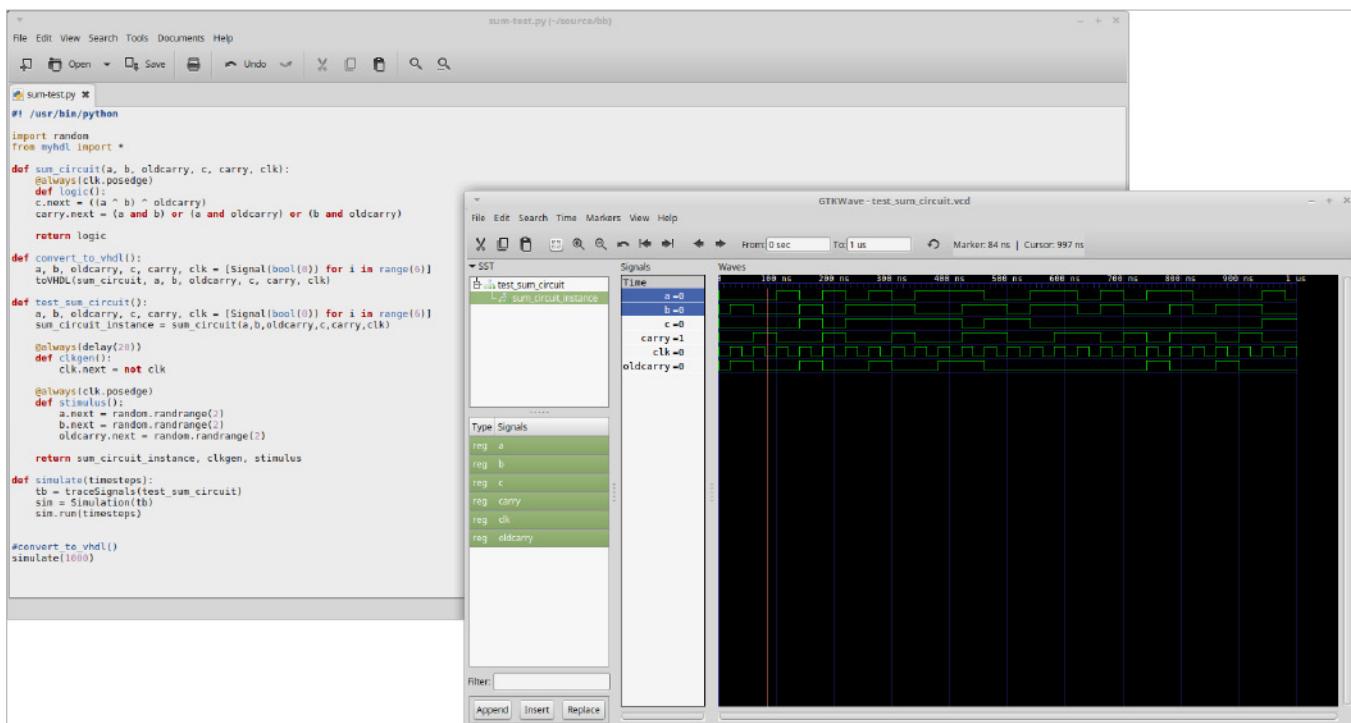
def sum_circuit(a, b, oldcarry, c, carry, clk):
    def logic():
        c.next = ((a ^ b) ^ oldcarry)
        carry.next = (a and b) or (a and oldcarry) or (b and oldcarry)
    return logic

def convert_to_vhdl():
    a, b, oldcarry, c, carry, clk = [Signal(bool(0)) for i in range(5)]
    toVHDL(sum_circuit, a, b, oldcarry, c, carry, clk)

def test_sum_circuit():
    a, b, oldcarry, c, carry, clk = [Signal(bool(0)) for i in range(5)]
    sum_circuit_instance = sum_circuit(a,b,oldcarry,c,carry,clk)
    @always(delay(20))
    def clkgen():
        clk.next = not clk
    @always(clk.posedge)
    def stimulus():
        a.next = random.randrange(2)
        b.next = random.randrange(2)
        oldcarry.next = random.randrange(2)
    return sum_circuit_instance, clkgen, stimulus

def simulate(timesteps):
    tb = traceSignals(test_sum_circuit)
    sim = Simulation(tb)
    sim.run(timesteps)

#convert to vhdl()
simulate(1000)
```



Rysunek 1. Przebiegi dla pełnego sumatora w programie GtkWave

```

@always(clk.posedge)
def logic():
    c.next = ((a ^ b) ^ oldcarry)
    carry.next = (a and b) or (a and oldcarry) or (b and
    oldcarry)

    return logic

def test_sum_circuit():
    a, b, oldcarry, c, carry, clk = [Signal(bool(0)) for i in
        range(6)]
    sum_circuit_instance = sum_circuit(a,b,oldcarry,c,carry,clk)

@always(delay(20))
def clkgen():
    clk.next = not clk

@always(clk.posedge)
def stimulus():
    a.next = random.randrange(2)
    b.next = random.randrange(2)
    oldcarry.next = random.randrange(2)

    return sum_circuit_instance, clkgen, stimulus

def simulate(timesteps):
    tb = traceSignals(test_sum_circuit)
    sim = Simulation(tb)
    sim.run(timesteps)

simulate(1000)

```

## Instalacja MyHDL

Pakiet MyHDL jest w całości napisany w języku Python. Toteż nie ma potrzeby, aby instalować dodatkowe biblioteki, aby uzyskać działającą instalację MyHDLa. Wystarczy sam Python dla systemów Linux, Windows bądź Mac OS. Z tego względu instalację możemy przeprowadzić, wydając polecenie:

```
pip install myhdl
```

Przy czym jeśli chcemy, aby MyHDL współpracował z narzędziami do tzw. ko-symulacji, np. Icarus, to niestety trzeba przeprowadzić dodatkową komplikację, której tutaj w racji ograniczonego miejsca nie będziemy omawiać.

Kompilacja samego pakietu MyHDL z kodu źródłowego naturalnie także jest możliwa i prosta. Jeśli wcześniej nie mieliśmy okazji tego przeprowadzić, to warto pokusić się o komplikację, a sprowadza się ona do ściagnięcia pliku źródłowego (ostatnia dostępna wersja to 0.9, zatem pełna nazwa pliku to: *myhdl-0.9.0.tar.gz*). Dekompresję przeprowadzamy w następujący sposób:

```
tar zxvf myhdl-0.9.0.tar.gz
```

Następnie przechodzimy do nowo utworzonego katalogu, wydając polecenie *cd myhdl-0.9.0*. Kompilację oraz zarazem instalację wykonamy, wydając polecenie:

```
python setup.py install
```

Możliwe jest też wskazanie katalogu parametrem *home*, gdzie pakiet MyHDL ma zostać zainstalowany:

```
python setup.py install --home=/home/user/dir
```

Najwygodniej jednak pozostać przy domyślnych katalogach instalacyjnych, ułatwi to korzystanie z pakietu MyHDL, choć wymagać będzie użycia polecenia *sudo*.

łyム pliku VHDL. I od razu możemy rozwijać tę wątpliwość: istotnie z poziomu pakietu MyHDL można takie rozwiązanie uzyskać.

Przypomnijmy główne założenie naszego zadania: chcemy stworzyć tryb znakowy o 80 kolumnach i 30 wierszach. Rozmiary zastosowanej czcionki to 8 na 16 pikseli. Łatwo, mnożąc  $80 \times 8 \times 30 \times 16$ , otrzymamy wartość 640 na 480 i taką rozdzielncość będzie tworzył nasz generator, przy odświeżaniu 60 Hz. Oznacza to, iż potrzebny będzie nam zegar 25 Mhz, aby zgodnie z wymogami standardu VGA uzyskać poprawny obraz. Dla uproszczenia nie będziemy zajmować się kolorami. Nasz generator będzie tworzyć zielone litery na czarnym tle (bardzo łatwo będzie można zmienić te kolory).

W Listingu 4 przedstawiono kod źródłowy odpowiadający za implementację trybu tekstowego VGA. Przy czym w listingu zaprezentowano tylko sam kod. Zawartość pamięci oraz definicja czcionki z racji wielkości zostały naturalnie usunięte. Razem z tymi definicjami wielkość pliku to około 110 kB, przy czym właściwa implementacja to funkcja o nazwie *vgatest* licząca około 80 linii kodu.

Przeglądając kod, widać, iż rozpoczynamy od definicji pamięci wideo *CONTENT\_BASE*, w której przechowywane są kody znaków przeznaczone do wyświetlenia na ekranie. Jednak niezbędna jest konwersja zawartości ekranu. W krotce *CONTENT\_BASE* umieszcza się zawartość, jaką chcemy wyświetlić na ekranie. Dla uproszczenia nie będziemy implementować pamięci RAM, bo chcemy na razie skupić się tylko na samej implementacji generacji sygnału VGA. Pierwotna postać krotki zawiera dwa typy danych: liczby całkowite oraz znaki tekstowe ujęte w apostrofy. Pętla *for* następująca po definicji zawartości ekranu tworzy nową krotkę *CONTENT*, która zawiera tylko i wyłącznie dane w postaci liczb, inaczej mówiąc – ujednolica typ danych, co jest niezbędne dla pakietu MyHDL.

Potrzebna jest także czcionka. Zakładamy, że będziemy mieli do dyspozycji 256 znaków o kodach od 0 do 255. Aby uprościć zadanie, możemy użyć czcionki np. z jądra Linux z pliku *linux/drivers/video/font-8x16.c*. Kod źródłowy nie będzie wymagał zbyt wielu zmian i bardzo łatwo dołączyć definicję czcionki do tworzonyego przez nas projektu. Ostatecznie definicję czcionki zawierać będzie krotka *FONT*. Jest to w uproszczeniu ciąg liczb, gdzie pojedynczy znak jest opisany przez szesnaście liczb.

### Listing 4. Implementacja trybu tekstowego

```

from myhdl import *
CONTENT_BASE = ( 0,0,0,0, ... i tak dalej -- zawartość pamięci
video )

CONTENT = ( )
idx=0
for l in CONTENT_BASE:
    if isinstance(l,str):
        CONTENT=CONTENT + tuple([ord(l)])
    else:
        CONTENT=CONTENT + tuple([l])

FONT = (
... definicje poszczególnych
... liter/znaków czcionki
/* 123 0x7b '{ */
0x00, /* 0000000 */
0x00, /* 0000000 */
0x0e, /* 00001110 */
0x18, /* 00011000 */
0x18, /* 00011000 */
0x18, /* 00011000 */
0x70, /* 01110000 */
0x18, /* 00011000 */
0x18, /* 00011000 */
0x18, /* 00011000 */

```

## TRYB TEKSTOWY

Nasze główne zadanie jest bardziej skomplikowane. Chcemy, tak jak napisaliśmy we wstępnie, opracować w Pythonie generator obrazu dla trybu tekstowego. Powstały plik VHDL po tłumaczeniu z Pythona zostanie użyty w projekcie dla układu Spartan 3. Oczekujemy, iż nie będzie konieczności wprowadzania zmian w powsta-

```

0x18, /* 00011000 */
0x0e, /* 00001110 */
0x00, /* 00000000 */
0x00, /* 00000000 */
0x00, /* 00000000 */
0x00, /* 00000000 */
... reszta definicji poszczególnych
... liter/znaków czcionki
)

def vgatext( clk25, redColor, greenColor, blueColor, horzSync,
vertSync):
    horzCnt = Signal(intbv(0)[10:])
    vertCnt = Signal(intbv(0)[10:])
    bits = Signal(intbv(0)[8:])
    row = 0
    col = 0
    addr = 0
    letcol = Signal(intbv(0)[8:])
    letter = 0

    @always_comb
    def addrCalc():
        col = (horzCnt - 144) // 8
        row = (vertCnt - 39) // 16
        letcol.next = (8 - ((horzCnt - 144) % 8))
        letrow = (vertCnt - 39) % 16
        addr = (row * 80) + col
        letter = CONTENT[int(addr)]
        bits.next=FONT[(int(letter)*16)+letrow]

    @always(clk25.posedge)
    def videoGen():
        redColor.next = 0
        greenColor.next = 0
        blueColor.next = 0
        if (horzCnt >= 144) and (horzCnt<784) and (vertCnt>=39) and (vertCnt<519):
            if (bits[letcol] == 1):
                greenColor.next = 7
        if (horzCnt >= 344) and (horzCnt<484) and (vertCnt>=239) and (vertCnt<319):
            redColor.next = 7
        if (horzCnt>0) and (horzCnt<97):
            horzSync.next = 0
        else:
            horzSync.next = 1
        if (vertCnt>0) and (vertCnt<3):
            vertSync.next = 0
        else:
            vertSync.next = 1
        horzCnt.next = horzCnt + 1
        if horzCnt==800:
            vertCnt.next = vertCnt + 1
            horzCnt.next = 0
        if vertCnt==521:
            vertCnt.next = 0
        return addrCalc, videoGen

def convert_to_vhdl():
    clk25 = Signal(bool(0))
    redColor = Signal(intbv(0)[3:])
    greenColor = Signal(intbv(0)[3:])
    blueColor = Signal(intbv(0)[2:])
    horzSync = Signal(bool(0))
    vertSync = Signal(bool(0))
    toVHDL(vgatext, clk25, redColor, greenColor, blueColor,
    horzSync, vertSync)

convert_to_vhdl()

```

Główne zadanie, tj. tworzenie obrazu, wykonuje funkcja `vgaText`. Na początku definiujemy potrzebne nam zmienne oraz sygnały. Przykładem może być licznik:

```
horzCnt = Signal(intbv(0)[10:])
```

zliczający poszczególny piksele obrazu w linii poziomej. Jest też, jak widać, sygnał, ponieważ będzie on wykorzystywany w dwóch funkcjach tworzących obraz. W tym momencie możemy podać proste wyjaśnienie, dlaczego nie możemy użyć zmiennej. Zmienne w przypadku języków VHDL, Verilog mają zasięg tylko lokalny i są ograniczone jedynie do procesu. Tymczasem sygnał ma charakter bardziej globalny, może być współdzielony przez kilka procesów. Dlatego liczniki pikseli są sygnałami.

Ogólnie znaczenie poszczególnych sygnałów i zmiennych jest następujące: sygnały `horzCnt` oraz `vertCnt` to główne liczniki poszczególnych pikseli, z których tworzony jest sygnał VGA. Sygnał `bits` to osiem bitów aktualnie przetwarzanego znaku (to powinno być sygnał, aby można było go użyć pomiędzy procesami, a proces w MyHDL to po prostu funkcja). Natomiast sygnał `letcol` to numer bitu, którym aktualnie się zajmujemy podczas tworzenia obrazu. Dodatkowe zmienne `row`, `col`, `addr` to odpowiednio numer wiersza, kolumny i adres w pamięci wideo. Ich wartości są generowane na podstawie głównych liczników pikseli.

Obecne dwie dodatkowe funkcje wykonują główne zadanie. Pierwsza z nich `addrCalc` jest odpowiedzialna za odczytanie informacji z pamięci wideo na podstawie wartości liczników pikseli, tj. sygnałów `horzCnt` oraz `vertCnt`. Poprzedzona jest atrybutem `@always_comb`, czyli jest to obiekt niezależny od sygnału zegarowego i działa niejako w trybie ciągłym.

Opis jej działania nie jest zbyt obszerny. Na podstawie liczników iteracji obliczamy, w którym wierszu i kolumnie się znajdujemy. Do sygnału `letcol` wpisujemy, którym bitem będziemy się zajmować. Następnie do zmiennej `letrow` umieszczamy obliczony numer wiersza ze znaku, jaki będzie nam potrzebny. Wysokość czcionki to 16 pikseli, dlatego wystarczy obliczyć resztę z dzielenia przez 16. Na podstawie wartości `row` i `col` obliczamy adres w pamięci wideo (zmienna `addr`). I możemy zapamiętać w zmiennej `letter` kod znaku, a następnie do sygnału `bits` wpisujemy osiem bitów reprezentujących wybrany fragment (wiersz) przetwarzanego znaku.

Za wygenerowanie odpowiednich sygnałów synchronizacji poziomej i odświeżania pionowego odpowiedzialna jest funkcja `videoGen`. Działa ona zgodnie z sygnałem zegarowym. Nie będziemy dokładnie omawiać, jak tworzyć obraz zgodny ze standardem VGA, powiemy tylko w tym momencie, iż trzeba zliczać piksele w liniach, a po dojściu do końca linii uruchamiamy sygnał synchronizacji poziomej i tak aż dojdziemy do ostatnich linii, po których trzeba aktywować sygnał synchronizacji pionowej. W zależności od użytej rozdzielczości zmieniają się wartości graniczne, ale dla nas najważniejsze są następujące linie kodu:

```
if (horzCnt >= 144) and (horzCnt<784) and (vertCnt>=39) and (vertCnt<519):
    if (bits[letcol] == 1):
        greenColor.next = 7
```

Jeśli warunek wartości w pierwszej linii kodu jest spełniony, oznacza to, iż znajdujemy się wewnętrz widzialnego obszaru obrazu i możemy z sygnału `bits` odczytać, czy dany bit określony przez sygnał `letcol` jest ustawiony na jeden. Jeśli tak, to zmieniamy wartość składowej koloru na 7, ponieważ mamy osiem dopusz-

czalnych wartości dla koloru zielonego, to siódemka jest wartością maksymalną (minimalna to zero). Pozostała część kodu videoGen sprowadza się do zwiększania i sprawdzania wartości liczników, aby, tak jak podaliśmy to wcześniej, w odpowiednich momentach aktywować sygnały synchronizacji.

Zanim przejdziemy do implementacji całości w pakiecie ISE, musimy jeszcze wygenerować plik VHDL, a zrobimy to jednym poleceniem:

```
python vgatest.py
```

Jeśli nasz kod został zapisany do pliku `vgatest.py`, powstanie plik `vgatest.vhd` oraz dodatkowy plik z biblioteką funkcji pomocniczych `pck_myhdl_090.vhd`. W projekcie ISE należy dodać wymienioną bibliotekę do projektu.

## ŁĄCZYMY WSZYSTKO W JEDNĄ CAŁOŚĆ

Użycie pakietu MyHDL, jak widać po podanych przykładach, nie sprawia trudności. Jednak oprócz MyHDL, aby wykorzystać wygenerowane pliki VHDL albo Verilog, potrzebne jest nam środowisko do pracy z układami FPGA. Zakładamy, że będzie to środowisko ISE firmy Xilinx (na Rysunku 2 przedstawiono nasz projekt po syntezie i utworzeniu pliku binarnego), ze względu na fakt, iż stosujemy układ Spartan 3.

Musimy założyć nowy projekt dla układu Spartan 3, a dokładniej dla układu w odmianie xc3s1200e. Nasz projekt naturalnie nie jest dedykowany do tej konkretnej wersji i może być używany praktycznie w dowolnym układzie FPGA, o ile jego zasoby będą wystarczające, aby pomieścić nasz projekt.

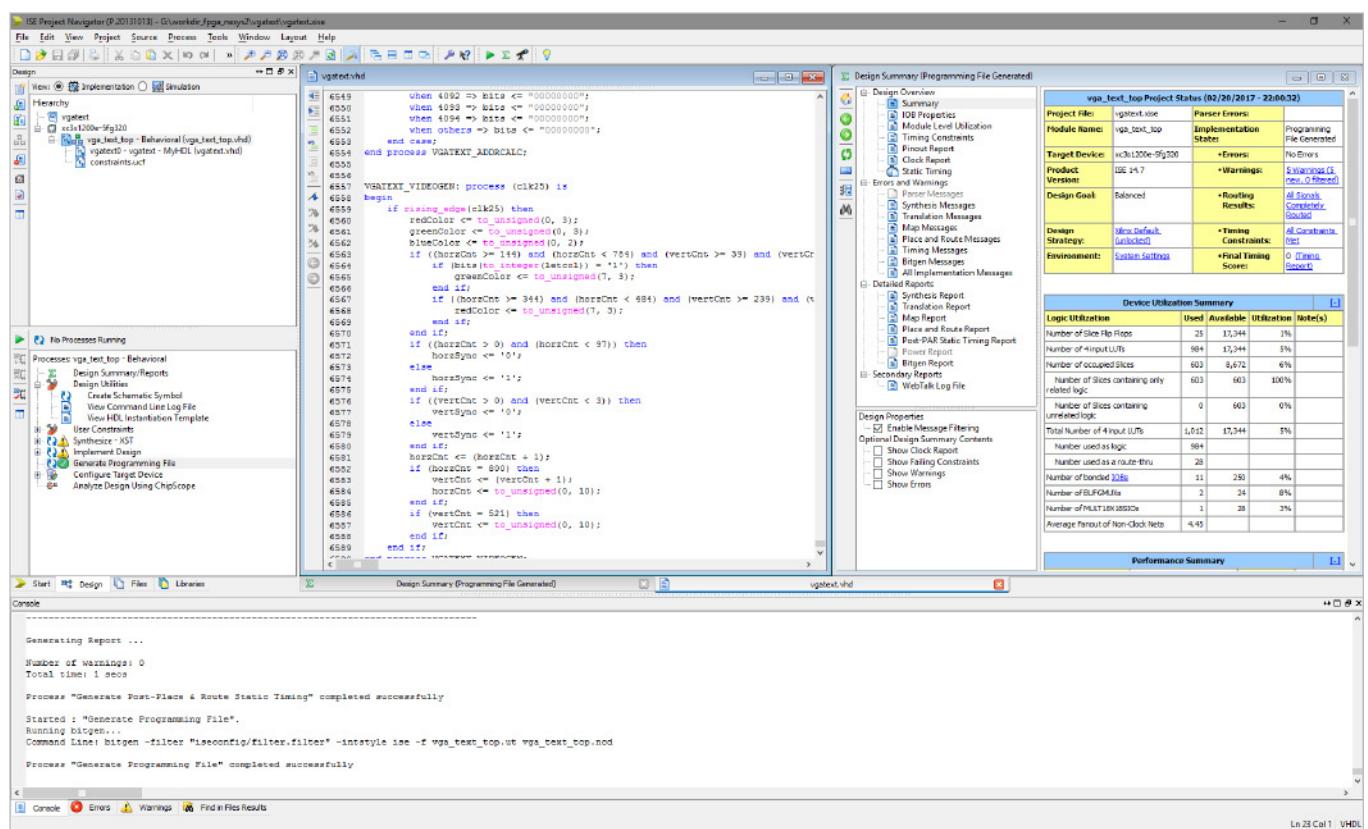
Pierwszy element, na jaki zwróciśmy uwagę, to plik UCF. Opisujemy w nim połączenia pomiędzy sygnałami zdefiniowanymi

w naszym projekcie a fizycznymi elementami, które są podłączone do układu FPGA. W naszym projekcie dotyczy to złącza VGA oraz sygnału zegara.

Postać pliku UCF dla płyty Nexys 2 dostosowanego do potrzeb naszego projektu została przedstawiona w Listingu 5. Definiujemy sygnał zegarowy `c1k`, który jest dostarczany przez złącze oznaczone jako `B8`, a także sygnały związane z generowaniem obrazu VGA. Podzielone zostały na kolory, np. złącza `R9`, `T8` oraz `R8` służą do kodowania koloru czerwonego. Mamy trzy złącza, a zatem są to trzy kolory, czyli mamy osiem odcienni czerwonego. Łącznie kolor jest kodowany za pomocą ośmiu złącz, ośmiu bitów, co pozwala uzyskać 256 kolorów. Opis kolorów scisłe zależy od sposobu, w jaki układ FPGA został zamontowany w danej płycie lub urządzeniu. Prezentowany projekt został oparty o płytę Nexys 2 i dlatego mamy takie rozwiązanie w sposobie podłączenia układu FPGA do złącza VGA. Do poprawnej obsługi sygnału obrazu mamy jeszcze sygnały `horzSync` oraz `vertSync`.

**Listing 5. Postać pliku UCF dla naszego projektu**

```
NET "clk50" LOC= "B8"; # Bank = 0 , Pin name = IP_L13P_0/GCLK8
, Type = GCLK , Sch name = GCLK0
NET "redColor<0>" LOC= "R9"; # Bank = 2 , Pin name = IO/D5 ,
Type = DUAL , Sch name = RED0
NET "redColor<1>" LOC= "T8"; # Bank = 2 , Pin name = IO_L10N_2
, Type = I/O , Sch name = RED1
NET "redColor<2>" LOC= "R8"; # Bank = 2 , Pin name = IO_L10P_2
, Type = I/O , Sch name = RED2
NET "greenColor<0>" LOC= "N8"; # Bank = 2 , Pin name = IO_L09N_2
, Type = I/O , Sch name = GRN0
NET "greenColor<1>" LOC= "P8"; # Bank = 2 , Pin name = IO_L09P_2
, Type = I/O , Sch name = GRN1
NET "greenColor<2>" LOC= "P6"; # Bank = 2 , Pin name = IO_L05N_2
, Type = I/O , Sch name = GRN2
NET "blueColor<0>" LOC= "U5"; # Bank = 2 , Pin name = IO/VREF_2
```



Rysunek 2. Projekt w programie ISE 14.7 z naszym generatorem trybu tekstowego

```
, Type = VREF , Sch name = BLU1
NET "blueColor<1>" LOC= "U4"; # Bank = 2 , Pin name = IO_
L03P_2/DOUT/BUSY , Type = DUAL , Sch name = BLU2

NET "horzSync" LOC= "T4" ; # Bank = 2 , Pin name = IO_L03N_2/
MOSI/CSI_B , Type = DUAL , Sch name = HSYNC
NET "vertSync" LOC= "U3" ; # Bank = 2 , Pin name = IO_L01P_2/
CSO_B , Type = DUAL , Sch name = VSYNC
```

Aby połączyć projekt w jedną całość, korzystamy z pliku *vga\_top.vhdl*. Treść tego pliku została zaprezentowana w Listingu 6. Plik ten bezpośrednio połączy sygnały z naszej implementacji generatora obrazu VGA do fizycznego złącza opisanego w pliku UCF. Dlatego w sekcji entity w definicji port podajemy te same nazwy sygnałów, jakie zostały określone w pliku UCF.

Nim przejdziemy do omówienia obiektu odpowiedzialnego za generację obrazu, musimy jeszcze utworzyć proces tworzący sygnał zegarowy o częstotliwości 25 Mhz. W Listingu 6 sekcja odpowiedzialna za to zadanie rozpoczyna się od linii proces (*clk50*). Ponieważ zastosowana przez nas płyta Nexys 2 domyślnie generuje sygnał 50 Mhz, to wystarczy aktywować sygnał *clk25* co drugie zdarzenie zegarowe, aby uzyskać sygnał zegarowy o częstotliwości 25 Mhz.

Komponent o nazwie *vgateText* reprezentuje obiekt naszego generatora obrazu utworzonego w Pythonie. Aby osadzić ten komponent, tworzymy instancję naszego generatora (o nazwie *vgateText0*) za pomocą następującej linii kodu:

```
vgateText0 : vgateText port map( clk25 => clk25, redColor =>
redColor, greenColor => greenColor, blueColor => blueColor,
horzSync => horzSync, vertSync => vertSync);
```

Wyrażenie typu *clk25 => clk25* oznacza połączenie sygnału z parametrem komponentu *vgateText*. W pliku UCF oraz w pliku *vga\_top.vhdl* celowo zdefiniowano sygnały o tych samych nazwach, aby uzyskać lepszą czytelność kodu i opisów sygnałów.

#### Listing 6. Plik główny naszego projektu

```
entity vga_text_top is
  Port ( clk50 : in STD_LOGIC;
  redColor : out unsigned(2 downto 0);
  greenColor : out unsigned(2 downto 0);
  blueColor : out unsigned(1 downto 0);
  horzSync : out std_logic;
  vertSync : out std_logic);
end vga_text_top;

architecture Behavioral of vga_text_top is

  signal clk25 : std_logic;
  component vgateText
    port ( clk25: in std_logic;
    redColor: out unsigned(2 downto 0);
    greenColor: out unsigned(2 downto 0);
    blueColor: out unsigned(1 downto 0);
    horzSync: out std_logic;
```

```
    vertSync: out std_logic
  );
end component ;
begin
  process (clk50)
begin
  if clk50'event and clk50='1' then
    if (clk25 = '0') then
      clk25 <= '1';
    else
      clk25 <= '0';
    end if;
  end if;
end process;
vgateText0 : vgateText port map( clk25 => clk25, redColor =>
redColor, greenColor => greenColor, blueColor => blueColor,
horzSync => horzSync, vertSync => vertSync);
end Behavioral;
```

## PODSUMOWANIE

Pakiet MyHDL to kolejny przykład ukazujący, że Python jest uniwersalnym językiem programowania. Istotnie mogliśmy napisać kod, który nie różni się w żaden zasadniczy sposób od typowego programu w Pythonie. Jedynym elementem są atrybuty, ale i te można spotkać w wielu pakietach dla Pythona. W ramach naszego mini projektu możemy dodać jeszcze kilka funkcji, np. sprzętową obsługę kurSORA. Przydałyby się dodatkowe atrybuty pozwalające na określenia koloru tła oraz koloru czcionki, najlepiej oddzielnie dla każdego znaku.

Trzeba też podkreślić, że MyHDL, choć stosuje język wysokiego poziomu, nie zwalnia nas z konieczności implementacji czasem bardzo niskopoziomowych funkcji. Nie jest to narzędzie, które zastąpi np. dostępny pakiet Vivado HLS, gdzie pisane programy w języku C są tłumaczone do Veriloga lub VHDLa. Z drugiej jednak strony pakiet Vivado HLS nie pozwoli nam na tak proste utworzenie sterownika VGA, ponieważ jak dotąd nie oferuje dostępu do sygnału zegara. Widać w tym momencie, iż są to rozwiązania dedykowane do różnych obszarów zastosowań i tak należy je traktować. Dlatego tradycyjnie na koniec jak zawsze zachęcamy do własnych poszukiwań i implementacji sprzętowego kurSORA do naszego trybu tekstopiowego.

## W sieci:

Pakiet MyHDL:

<http://www.myhdl.org/>

Najważniejsza strona o języku/środowisku Python:

<http://www.python.org/>

Zestawienie kilku metod generacji kodu HDL:

<https://github.com/cfelton/alt.hdl>

Chisel, podobne rozwiązanie jak MyHDL ale dla języka Scala:

<https://chisel.eecs.berkeley.edu/>



**MAREK SAWERWAIN**

Autor, pracownik naukowy Uniwersytetu Zielonogórskiego, na co dzień zajmuje się teorią kwantowych języków programowania, ale także tworzeniem oprogramowania dla systemów Windows oraz Linux. Zainteresowania: teoria języków programowania oraz dobra literatura.

# Let us bring you home

[www.semihalf.com](http://www.semihalf.com)

Semihalf creates software for advanced solutions in the areas of operating systems, virtualization, networking and storage. We make software which is tightly coupled with the underlying hardware to achieve maximum system capacity for running at scale. We offer a dynamic and open yet principled work environment with exceptional engineering challenges. Our partners and customers are semiconductor industry leaders mainly from Silicon Valley.

Join us for a deep dive into the latest microprocessor technologies and high performance software development.

Currently we are looking for

## Kernel Hacker

(Krakow)

### Your challenges

You will be responsible for bringing up new hardware and creating low-level software running on multicore processors. You will be hacking on operating system kernel (BSD, Linux), writing device drivers and optimizing for the best performance results. Your code will often be submitted to the open source repositories.

### Requirements

- ④ Fluency in C code development and debugging
- ④ Low level coding experience (Linux or BSD kernel, bootloaders)
- ④ x86 or ARM assembly experience
- ④ Handling of standard shell utilities and tools like GCC, GDB, GIT, DTrace

### Benefits



Flexible working hours aligned to your individual preferences



Small teams (2-6 persons) with real influence on projects



Individual yearly training budget



Unique employee profit sharing programme



Social package, medical health care, multisport card



Chillout room, game console

# Co robi Twój program i czemu tak wolno?

## Rzecz o mierzeniu wydajności programów

Znajdowanie problemów z wydajnością programów wydaje się bardziej sztuką, niż nauką, ale to nie jest prawda. W tym artykule postaram się przybliżyć ilościowe, empiryczne podejście do tego skomplikowanego zagadnienia.

### ŚRODOWISKO PRACY I PRZEDMIOT POMIARU

W artykule przedstawiono analizę krótkiego programu napisanego w C++ w środowisku GNU/Linux z użyciem narzędzia perf. Dodatkowo wyniki pomiarów będą przetwarzane skryptami shell-owymi Bash-a i AWK. Programy kompilowane są z opcją -O3, tj. z agresywną optymalizacją. Wszystkie pomiary są wykonane na procesorze Core i5-4460 3.2 GHz.

W Listingu 1 przedstawiono program będący przedmiotem badań.

**Listing 1. Program będący przedmiotem analizy wydajności**

```
#include <list>
#include <cstdlib>
#include <iostream>
#include <numeric>

template < class T >
void init(T & container,
           unsigned size)
{
    for (unsigned i = 0; i < size; i++)
        container.push_back(rand() -
            RAND_MAX / 2);
}

template < class T >
__attribute__((noinline))
long test_sum(T & container,
               unsigned repetitions)
{
    long sum = 0;

    for (unsigned i = 0; i < repetitions;
         i++)
        sum =
            std::accumulate(container.
                            begin(),
                            container.end(),
                            sum);

    return sum;
}

int main(int argc, char *argv[])
{
    if (argc != 3)
        return 1;

    unsigned size = atoi(argv[1]);
    unsigned repetitions = atoi(argv[2]);

    std::list< long >list;
    init(list, size);

    long sum =
        test_sum(list, repetitions);

    std::cout << sum << std::endl;

    return 0;
}
```

Program z Listingu 1 produkuje sumę liczb całkowitych umieszczonych w klasie kolekcji: liście list<long>.

Przykładowe uruchomienie programu daje takie rezultaty:

```
$ time ./sum 1000000 1000
1419548128300
real 0m2.486s
user 0m2.488s
sys 0m0.000s
```

W dalszej części artykułu postaram się w sposób systematyczny i przy użyciu odpowiedniej metodologii odpowiedzieć na pytanie, co zrobić, żeby ten program działał szybciej. Doświadczeni czytelnicy zapewne znają z góry odpowiedź na to pytanie, ale celem tego artykułu jest dojście do tezy i jej udowodnienie za pomocą liczb.

### KILKA SŁÓW O POMIARACH

Parafrując Marka Twaina, należy stwierdzić, że na świecie istnieją kłamstwa, parszywe kłamstwa, statystyki oraz pomiary wydajności, czyli znajdują się one w hierarchii rzeczy podnych na samym dniu, zaraz po statystyce. W istocie wszystkie pomiary wydajności programów opierają się na wybiórczo stosowanej statystyce i dla tego wyniki pomiarów bywają trudne w interpretacji.

Podstawowe wyzwanie polega na tym, czy dany pomiar można uogólnić, tzn. czy można założyć, że ten sam program na innej maszynie będzie zachowywał się podobnie. Żeby nabyć takiej pewności, należy posłużyć się wiedzą z zakresu architektury komputerów i pamięć o kilku fundamentalnych zasadach pomiarów komputerowych:

- » Wszystkie parametry pomiaru oprócz jednego powinny być takie same, tzw. klauzula *ceteris paribus*.
- » Wyniki pomiaru służą do porównywania z innymi wynikami, ale tylko identycznego pomiaru.
- » Pomiary programów nie przystają do siebie, jeśli programy znacząco się różnią.

### BHP OPTYMALIZACJI

Zanim programista zabierze się za ulepszanie programu, należy zadać sobie kilka podstawowych pytań:

1. Skąd wiadomo, że istnieje problem z wydajnością programu?
2. Czy kiedykolwiek program zachowywał się inaczej?
3. Co ostatnio uległo zmianie?
4. Czy spadek wydajności jest mierzalny, np. jako czas wykonania?
5. Czy ten sam problem widać u innych?

Ta lista pytań nosi nazwę *Problem Statement Method*. Pytania wydają się trywialne, ale spróbujmy się pochylić nad pytaniem 1. Czas

wykonania programu wynosi 2.4 sekundy. Czy to dużo, czy mało? Skąd wiadomo, że można lepiej? Można zadać to pytanie w jeszcze inny sposób:

### Kiedy należy uznać, że optymalizacja jest skończona?

Jest to fundamentalne pytanie, które musi sobie zadać każdy inżynier oprogramowania, być może zanim nawet przystąpi do pracy. Odpowiedź na to pytanie wymaga wyznaczenia górnego pułapu wydajności dla zadanego problemu. Nasz problem to sumowanie dużej ilości liczb. Skąd wiadomo, jak szybko może to zrobić procesor Core i5-4460? Istnieją co najmniej dwa źródła wiedzy:

- » Benchmarki
- » Dokumentacja techniczna

Pierwsze dostarcza nam wyników realistycznych, ale być może obarczonych błędem, ponieważ benchmark może mierzyć nie do końca to samo. Drugie źródło pozwala określić teoretyczny pułap, który bywa niemożliwy do osiągnięcia, ale stanowi pewien punkt odniesienia. Dodatkowo wynik teoretyczny powinien być nie gorszy od benchmarku.

Oprócz oszacowań należy upewnić się, że nasz program rzeczywiście jest ograniczony przez procesor, a nie przez inne elementy systemu, takie jak:

- » operacje wejścia-wyjścia
- » obsługa pamięci wirtualnej (np. swap)

Jest to częścią tzw. *Workload Characterization Method*:

1. Co powoduje obciążenie systemu? Program, sieć, dysk...
2. Skąd się to bierze? Np. konkretna ścieżka w kodzie.
3. Jak scharakteryzować obciążenie: pasmo [MB/s], obciążenie CPU [%], IOPS (operacje na sekundę)...
4. Jak obciążenie zmienia się w czasie?

Dla naszego programu oczekwaną odpowiedzią jest: system obciążony jest przez program, który zużywa 100% procesora, i miarą obciążenia jest czas wykonania programu.

## WYNIK TEORETYCZNY

Nowoczesne procesory potrafią wykonywać kilka jednocześnie operacji na cykl. Mikroarchitektura Haswell ma 4 jednostki stałoprzecinkowe (ALU) i dwie 256-bitowe jednostki wektorowe, z których każda może wykonać 4 jednocześnie dodawania. Przyjmijmy konserwatywnie 4 operacje sumowania na cykl. Dla takiego wyniku nasz program, pomijając czas na inicjalizację, powinien działać:

$$T = \frac{N}{f * ops} = \frac{1000 * 10^6}{3.2 * 10^9 * 4} = 0.078125[s]$$

gdzie:

- »  $T$  – całkowity czas wykonania programu
- »  $N$  – całkowita ilość operacji programu
- »  $f$  – taktowanie procesora
- »  $ops$  – ilość operacji na cykl

Jest to bardzo optymistyczne założenie, pomijające kilka istotnych czynników:

- » narzucona wykonanie innych instrukcji, np. pętli
- » zdolność dostarczenia do procesora wystarczającej ilości danych

## BENCHMARK

Za benchmark może posłużyć nam prosty program z Listingu 2, który sumuje tablicę z użyciem prostej pętli.

### Listing 2. Program służący jako benchmark

```
#define SIZE 1000000
long a[SIZE] = {[0 ... SIZE - 1] = 1};
long scalar;
int main(void)
{
    unsigned k;
    unsigned i;
    for (k = 0; k < 1000; k++)
        for (i = 0; i < SIZE; i++)
            scalar += a[i];
    return 0;
}
```

Zanim uruchomimy program, należy się jedna nota ostrzegawcza.

### Nigdy nie należy zakładać taktowania procesora z góry, bo jest ono ustalane dynamicznie!

Jeśli w obliczeniach bazujemy na taktowaniu, to musimy być bardzo ostrożni w określaniu, z jakim taktowaniem został wykonany nasz program. Najlepiej zawsze uruchamiać go polecienniem `perf stat`:

```
$ perf stat ./benchmark
Performance counter stats for './benchmark':
      370,054708  task-clock (msec)          # 0,999 CPUs utilized
                  2 context-switches          # 0,005 K/sec
                  0 cpu-migrations           # 0,000 K/sec
                 166 page-faults             # 0,449 K/sec
  1204104345  cycles                   # 3,254 GHz
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
  2002019315  instructions            # 1,66 insns per cycle
  500401770   branches                # 1352,237 M/sec
     8058 branch-misses             # 0,00% of all branches
                                         0,370518934 seconds time elapsed
```

Rubryka `cycles` podaje oszacowaną częstotliwość taktowania, a ostatnia linijka, tj. `time elapsed`, to czas spędzony przez program na działaniu, czyli odpowiednik rubryki `user` polecenia `time`.

Polecenie `perf stat` pozwala nam na sprawdzenie pozostałych zaleceń BHP:

- » rubryka `task-clock` informuje, że program zużywa 99.9% CPU
- » rubryki `context-switches` i `cpu-migrations` wskazują, że proces nie był przerywany, co w skrajnych przypadkach mogłoby wpływać na wynik
- » rubryka `page faults` nie wskazuje na problem z pamięcią wirtualną, a można to zweryfikować poprzez upewnienie się, że stronnicowanie zachodzi głównie podczas inicjalizacji programu.

Upewniwszy się, że system wykonał nasz program zgodnie z założeniami, można przystąpić do obliczeń:

$$ops = \frac{N}{f * T} = \frac{1000 * 10^6}{3.2 * 10^9 * 0.37} \approx 0.8$$

Podobne obliczenia przeprowadźmy dla programu z Listingu 1 za pomocą `perf stat` i przy pilnowaniu częstotliwości:

$$T = 0.29 \Rightarrow ops = \frac{1000 * 10^6}{3.2 * 10^9 * 0.29} \approx 0.1^t.$$

Oznacza to, że benchmark uzyskał wynik 8 operacji dodawania średnio na 10 cykli. Nasz program daje za to zawrotny wynik jednego dodawania na 10 cykli! Uzbrojeni w powyższe szacunki możemy wstępnie opisać zagadnienie wg metody *Problem Statement*:

1. Skąd wiadomo, że jest problem? Istnieje problem z wydajnością, ponieważ: program wykonuje 0.1 dodawania na cykl, gdy nasz benchmark wykonał 0.8 dodawania, a teoretycznie można nawet wykonać 4 lub 8 z użyciem instrukcji wektorowych.
2. Czy rezultaty się powtarzają? Program daje przewidywalne rezultaty: 0.1 dla listy przy 3.2 GHz
3. Co uległo zmianie? Na razie nic, bo zaczynamy od zera.
4. Czy problem jest mierzalny? Spadek wydajności jest mierzalny wg kilku metryk (czas, operacje na sekundę) z ostrożnym uwzględnieniem innych czynników:
  - » obciążenia procesora
  - » obciążenia systemu
  - » częstotliwości taktowania
5. Czy problem widać u innych? Ten sam problem widać u innych i jest on szeroko opisany w literaturze. My natomiast udajemy, że widzimy go po raz pierwszy po to, aby przekonać niedowiarków i nauczyć się samodzielnego rozwiązywania podobnych problemów.

## SKRZYNEK Z NARZĘDZIAMI DO OPTYMALIZACJI

W dalszej części artykułu będziemy analizować nasz program za pomocą tzw. liczników wydajności (*performance counters*), które są udostępniane przez Linuksa za pomocą narzędzia `perf`. Domyślnym licznikiem jest `cycles`, czyli cykle procesora. Używając tego licznika, można wstępnie zanalizować program i poszukać wąskiego gardła:

```
$ perf record ./sum 10000000 1000
...
$ perf report --stdio
...
# Samples: 10K of event 'cycles:pp'
# Event count (approx.): 8327909671
#
# Overhead  Command Shared Object      Symbol
# .....     .....   .....           .....
#
97.96%    sum      sum             [...] test_sum<std ...
  0.47%    sum      libc-2.23.so    [...] _int_malloc
  0.27%    sum      libc-2.23.so    [...] _int_free
  0.26%    sum      libc-2.23.so    [...] malloc
```

Wydruk z polecenia `perf report` pokazuje 98% czasu spędzonego w funkcji-szablonie `test_sum()`. Gdyby nie atrybut `noinline` dodany do szablonu, zobaczylibyśmy jedynie funkcję `main()`, ponieważ szablon uległby rozwinięciu w kod wewnętrz `main()`. Wyróżnienie części kodu, gdy kompilator rozwija funkcje (*inlining*) i usuwa zbędne symbole, to często spotykany, praktyczny problem. W pierwszym podejściu warto selektywnie wyłączyć możliwość rozwijania niektórych funkcji. Całkowite wyłączenie tej optymalizacji (`-fno-inline-functions`) jest ryzykowne, bo może znacząco spowolnić program i w efekcie wyprodukować nirealistyczne rezultaty.

Nie jest żadnym zaskoczeniem, że funkcja sumująca znalazła się na pierwszym miejscu. W niektórych przypadkach posługiwanie się samym licznikiem cykli wystarcza do odkrycia, gdzie jest wąskie gardło, jeśli programista nie wie, gdzie się ono znajduje. W naszym przypadku problem jest nieco inny. Z góry wiadomo, że program spędza czas na przechodzeniu przez elementy listy i sumowaniu ich (o ile dobrze wykonamy pomiar), ale nie wiadomo, jaka jest przyczyna tego, że program działa poniżej oczekiwani. Żeby dociec przyczyny, należy użyć bardziej wyszukanych metryk, niż tylko cykle:

```
$ perf list
List of pre-defined events (to be used in -e):
branch-instructions OR branches          [Hardware event]
branch-misses                           [Hardware event]
bus-cycles                             [Hardware event]
cache-misses                           [Hardware event]
cache-references                      [Hardware event]
cpu-cycles OR cycles                   [Hardware event]
instructions                           [Hardware event]
ref-cycles                            [Hardware event]
...
L1-dcache-load-misses                 [Hardware cache event]
L1-dcache-loads                        [Hardware cache event]
L1-dcache-stores                       [Hardware cache event]
L1-icache-load-misses                 [Hardware cache event]
LLC-load-misses                        [Hardware cache event]
LLC-loads                             [Hardware cache event]
LLC-store-misses                      [Hardware cache event]
LLC-stores                            [Hardware cache event]
```

Oto niektóre z ciekawszych zdarzeń, które `perf` może śledzić:

- » `cycles` – cykle procesora wg aktualnego zegara CPU (ilość cykli na sekundę może się zmieniać)
- » `instructions` – instrukcje zakończone (*retired*)
- » `branches` – instrukcje warunkowe
- » `branch-misses` – błędnie wykonane instrukcje warunkowe – kosztują kilkanaście cykli oraz stracony czas na wykonanie błędnej ścieżki kodu
- » `cache-references` – odwołania do `cache`, tutaj jest to `cache` ostatniego poziomu (LLC – *Last Level Cache*), odwołanie do `cache` L3 zajmuje około 40 cykli
- » `cache-misses` – odwołania do RAM spowodowane brakującym wpisem w `cache` (poza LLC jest już tylko RAM), odwołanie do RAM to 200 lub więcej cykli procesora
- » `L1-dcache-loads` – odwołanie do `cache` danych L1, latencja 4 cykle dla Intel'a, lepiej mają tylko rejestr z zerową latencją
- » `L1-dcache-load-misses` – brak wpisu w L1, oznacza odwołanie do L2 (12 cykli), L3 lub do RAM

Niestety, liczniki dostarczone przez producentów procesorów nie mogą być rozpatrywane jako precyzyjne instrumenty, a raczej jako statystyki. Dzieje się tak z kilku powodów:

### 1. Precyzyjne znaczenie metryk zależy od konkretnej mikroarchitektury!

Haswell może inaczej zliczać zdarzenia typu `cache-miss`, niż Broadwell albo Skylake. Procesory ARM, POWER i inne też mają liczniki wydajności i każdy z nich może zachowywać się w inny sposób. Nie jest to jednak powód do rozpaczy, bo liczniki stanowią ważne źródło informacji, jeśli odczyty liczników porównujemy z tymi samymi odczytami z innych programów, czyli tak samo, jak z pomiarami wydajności. Przykład: można zliczyć zdarzenia `cache-miss` dla badanego programu i dla benchmarku i porównać rząd wielkości.

## 2. Polecenie perf record samo działa na zasadzie statystycznej.

perf próbuje wartości liczników z częstością rzędu kilku kHz. Można dostroić do innej wartości, pamiętając, że im gęstsze próbkowanie, tym większy narzut. Doświadczeni użytkownicy twierdzą, że narzut 5% jest tolerowany. Wyznaczenie narzutu pozostawiam czytelnikowi do samodzielnego wykonania.

## 3. Współczesne procesory wykonują instrukcje poza kolejnością (Out Of Order).

Jest to prawda dla prawie wszystkich procesorów, za wyjątkiem niektórych energooszczędnego jednostek na urządzenie mobilne, np. Cortex A53, Cortex A7, niektóre Atomy. Wykonanie *out of order* oznacza, że czas oczekiwania na zdarzenie, np. cache-miss, może być poświęcony na wykonanie innej pracy, o ile taka jest dostępna dla procesora. Inaczej mówiąc, niewielkie opóźnienia (np. odwołanie do cache L1 lub L2) wcale nie powodują wstrzymania pracy procesora, bo zazwyczaj można je „ukryć”, wykonując inne instrukcje.

## 4. Procesory superskalarne posiadają wiele jednostek wykonawczych (Wide-issue superscalar).

Superskalarność oznacza, że procesor może wykonywać kilka instrukcji jednocześnie w jednym cyklu. Nazywa się to w literaturze *Instruction-Level Parallelism*, ILP. W wydajnych procesorach mamy 4-6 niezależnych potoków instrukcji (*instruction pipeline*).

Podsumowując te wszystkie zjawiska, nie można rozpatrywać działania programu jako procesu sekwencyjnego z „przerwami” w postaci zdarzeń typu cache-miss lub branch-miss, ale jako **strumień instrukcji, który może być większy lub mniejszy**. Całkowita przepustowość procesora, określona teoretycznie na około 4 instrukcje na cykl (wyjaśnione poniżej), jest zmienna w czasie, a zadanie optymalizacyjne polega na maksymalizacji przepustowości, czyli **maksymalizacji ILP**.

Wszystkie zjawiska opisane powyżej zobrazowane są po części na Rysunku 1 i Rysunku 2. W szczególności należy zwrócić uwagę na 4-krotny dekoder instrukcji na Rysunku 1, który w jednym cyklu może zdekodować 4 rozkazy, oraz na 8 tzw. portów (*execution ports*) na Rysunku 2. Każdy port może wykonać jedną z wyszczególnionych pod nim operacji, np. ALU wykonuje operacje arytmetyczne i logiczne na liczbach całkowitych.

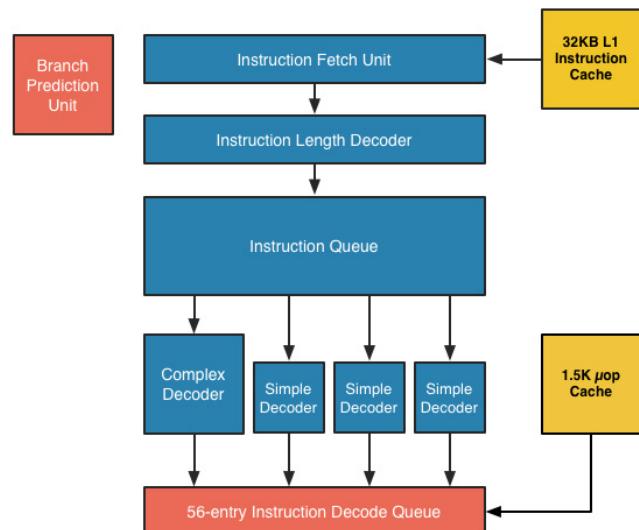
## UŻYTECZNE METRYKI

Największą siłą liczników wydajnościowych jest to, że można analizować kilka z nich, jednocześnie konstruując syntetyczne metryki. Tworzenie złożonych metryk pozwala na częściowe obejście ograniczeń wymienionych powyżej. Oto kilka najbardziej użytecznych:

$$\begin{aligned} IPC &= \frac{\text{instructions}}{\text{cycles}} \\ CPI &= \frac{\text{cycles}}{\text{instructions}} = IPC^{-1} \\ L1MissRate &= \frac{L1Miss * 100}{L1Hit} \\ MPKI &= \frac{L1Miss * 1000}{\text{instructions}} \\ L3MissRate &= \frac{L3Miss}{L3Hit} \end{aligned}$$

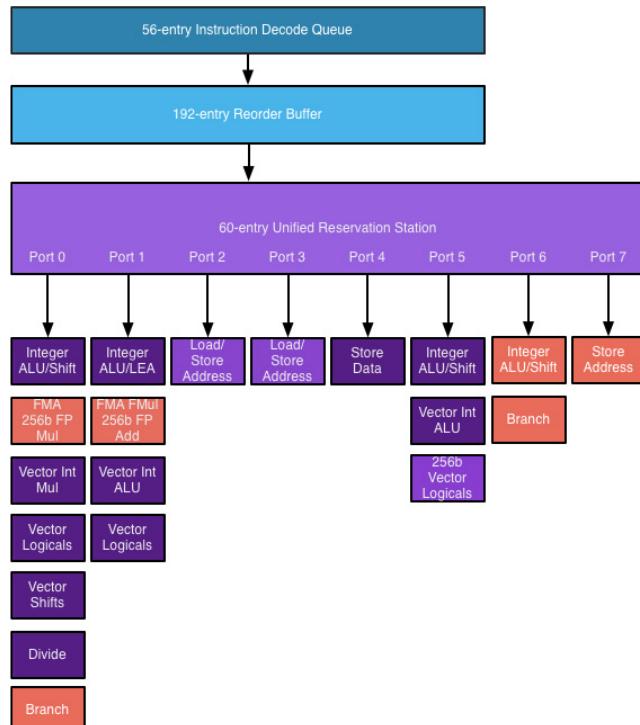
**IPC**, *instructions per cycle*, to syntetyczna metryka mierząca średnią ilość instrukcji na cykl, czyli *de facto* ILP, którego zwiększenie stanowi cel optymalizacji. Praktyczną regułą jest oczekiwanie, że IPC będzie powyżej 1, przy czym konkretna wartość zależy od charakteru programu, gdyż programy mają swoje „naturalne ILP”, czyli najczęściej poziom wewnętrznej współbieżności programu wynika z jego struktury, z algorytmu, i tego ograniczenia procesor nie może pokonać. Dla programów numerycznych, w tym dla naszego, trywialnego, IPC jest zazwyczaj bardzo wysokie (może być większe niż 10). W dalszej części artykułu będziemy stosować nazwę IPC w odniesieniu do metryki, a nazwę ILP w odniesieniu do własności samego programu.

## Intel Haswell Front End



Rysunek 1. Pierwsza część potoku instrukcji w mikroarchitekturze Haswell, tzw. Font-End. Źródło: Anandtech

## Intel Haswell Execution Engine



Rysunek 2. Część wykonawcza potoku instrukcji w mikroarchitekturze Haswell, tzw. Back-End. Źródło: Anandtech

rakteru programu, gdyż programy mają swoje „naturalne ILP”, czyli najczęściej poziom wewnętrznej współbieżności programu wynika z jego struktury, z algorytmu, i tego ograniczenia procesor nie może pokonać. Dla programów numerycznych, w tym dla naszego, trywialnego, IPC jest zazwyczaj bardzo wysokie (może być większe niż 10). W dalszej części artykułu będziemy stosować nazwę IPC w odniesieniu do metryki, a nazwę ILP w odniesieniu do własności samego programu.

**CPI**, *clocks per instruction*, to odwrotność IPC, czasami bardziej przydatna do kalkulacji, gdy inne wartości też mierzmy względem instrukcji. Oczekujemy, że CPI będzie mniejsze od 1 dla krytycznych fragmentów kodu.

**L1MissRate** wskazuje, ile odwołań do pamięci wymaga sięgnięcia „głębiej” do *cache*. To zdarzenie może mieć wielorakie konsekwencje, gdyż brakujące dane mogą znajdować się na wyższych poziomach *cache* (L2, L3) albo nawet w pamięci RAM. Optymalizacja użycia pamięci powinna dążyć do tego, żeby większość odwołań nie wykrazała poza L1. Metrykę najlepiej wyrazić w procentach, stąd w liczniku 100. Trudno oszacować bezwzględny wpływ tej metryki na wykonanie programu, ale wartości powyżej 10% powinny zwrócić uwagę programisty.

**MPKI**, *Miss Per Kilo Instructions*, to metryka wskazująca na częstotliwość występowania zdarzeń typu *L1-dcache-miss* w programie. Dla wygody, żeby nie używać ułamków, można liczyć ilość zdarzeń na 1000 instrukcji, czyli *de facto* promile. Im mniej program ma tych promili, tym bardziej można na nim polegać. Podobnie jak w przypadku *L1-MISS RATE*, wartość MPKI należy interpretować w jakimś kontekście, np. w porównaniu z innym programem, ale program „wskazujący na spożycie” z wartością 100 promili i więcej należy poddać „kontroli drogowej”.

**L3MissRate** – procent odwołań do pamięci *cache* L3, który kończy się odwołaniem do RAM. Próg ostrożnościowy można ustalić na 10%. Metryka ta jest powiązana z *L1MissRate*, bo każde zdarzenie *L3Miss* generuje także *L1Miss*.

## Uwaga

Zawsze należy pamiętać o tym, że zadanie optymalizacji powinno się zaczynać od empirycznie wykrytych wąskich gardel (*hot-spots*), które pokazuje np. najprostsza metryka *cycles*.

Nie ma sensu skupiać się na fragmentach programu, które nie są wąskim gardłem, a przyspieszanie takiego kodu daje minimalny zysk. Na przykład, jeśli program spędza 40% czasu funkcji A i 10% czasu w funkcji B, to 2-krotne przyspieszenie każdej z funkcji daje:

$$\text{Program Speedup} = \frac{1}{1 - \frac{\text{Overhead}}{\text{Speedup}_e}}$$

$$\text{Program Speedup}_A = \frac{1}{1 - \frac{\text{Overhead}_A}{\text{Speedup}_A}} = \frac{1}{1 - \frac{0.4}{2}} = \frac{1}{0.8} = 1.25$$

$$\text{Program Speedup}_B = \frac{1}{1 - \frac{0.1}{2}} = \frac{1}{0.95} \approx 1.05$$

## DZIEL I RZĄDŹ

Jedną z klasycznych metod podchodzenia do problemu optymalizacji jest metoda USE, czyli *Utilization, Saturation, Errors*. Podejście polega na tym, że identyfikujemy kluczowe zasoby systemu i dla każdego z nich obliczamy:

- » Utylizację – stopień zużycia względem maksimum
- » Saturację – ilość pracy opóźnionej, czekającej na wykonanie
- » Błędy – sytuacje wyjątkowe mające wpływ na wydajność

W przypadku badania programu, który zużywa 100% procesora przez określony czas, można próbować podzielić podsystemy procesora i rozpatrzyć każdy zasób osobno. Na przykład:

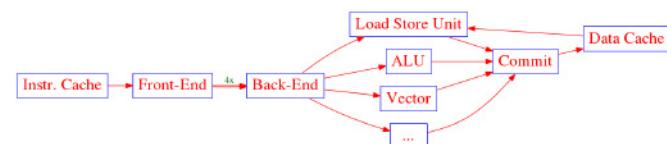
- » *Front-End*, czyli pobieranie i dekodowanie instrukcji
- » *Back-End*, czyli wykonywanie zdekodowanych wcześniej instrukcji, dodatkowo można go dzielić na:
  - » Rdzeń (jednostki wykonawcze, ALU itd.)
  - » Pamięć (Cache):
    - » Latencję
    - » Przepustowość
- » BPU, *Branch Prediction Unit*, czyli jednostkę odpowiedzialną za spekulacyjne wykonanie instrukcji warunkowych i skoków

Dla każdego z tych zasobów można wyznaczyć osobną metrykę zużycia i być może także sytuacji wyjątkowych.

Sposób rozwiązywania złożonych problemów przez podział na podproblemy nazywa się metodą „Dziel i rządź” (*Divide and conquer*) i jest dobrze znany adeptom informatyki. Podział procesora na podsystemy pozwala skonstruować drzewo decyzji i skupić się na jednym aspekcie działania programu na raz.

W dalszej części programu skupimy się na dwóch zasobach:

- » Rdzeń – wykonanie operacji arytmetycznych
- » Pamięć – pobieranie danych z pamięci w celu „nakarmienia” rdzenia danymi
  - » IOPS (Latencja)
  - » Pasmo (Przepustowość) w GB/s



Rysunek 3. Schemat przepływu danych wewnętrznych procesora dla programu z Listingu 1

## ZBIERANIE METRYK

Program perf potrafi działać w dwóch trybach:

- » zliczanie zdarzeń z całego programu
- » szacunkowe zliczanie zdarzeń dla symboli (funkcji, metod) za pomocą próbkowania

Obie metody obarczone są błędami, z tym że błędy wynikają z czegoś innego.

**Zliczanie** działa na całym programie, w związku z tym wprowadza „szum” w postaci zdarzeń z inicjalizacji programu. Im bardziej skomplikowany program, tym więcej „szumu”. Wstępnią fazę inicjalizacji można pominąć, ale jeśli interesuje nas tylko jedna funkcja na raz, to metoda ta może okazać się nie dość dokładna. Z drugiej strony zliczanie daje nam ujęcie globalne, pozwalając scharakteryzować cały program.

**Próbkowanie** pozwala zebrać statystykę dla każdej funkcji lub metody oddzielnie (o ile nie została rozwinięta przez kompilator), ale jest to obarczone błędem próbkowania. Błąd ten trzeba minimalizować w praktyce przez zapewnienie, że program działa dostatecznie długo. Ważną rzeczą jest, żeby obciążenie programu było podczas zbierania metryk stałe i jednorodne, tj. program powinien cały czas wykonywać ten sam rodzaj pracy, co jest szczególnie ważne dla skomplikowanych aplikacji, np. serwerowych.

Zacznijmy od zebrania metryki IPC dla programu z Listingu 1:

```
$ perf record -e \{cycles,instructions\} ./sum 1000000 1000
...
```

Warto uruchomić perf report z opcją -n, żeby upewnić się, że ilość zebranych próbek jest wystarczająca:

```
$ perf report -n --stdio
...
# Overhead Samples Command Shared Object Symbol
...
97.98% 10358 sum sum [...] test_sum<std::
...
...
```

Otrzymaliśmy ponad 10000 próbek dla naszego symbolu, co powinno wystarczyć. Przystępujemy zatem do liczenia metryki:

```
$ perf report --stdio --group --show-total-period --percent-limit 0.1 | tee report.txt
# Samples: 21K of event 'anon group { cycles, instructions }'
# Event count (approx.): 13296774659
#
#      Overhead          Period   Command Shared Object   Symbol
# .....,.
#    97.98%  90.71%  8699011996  4007811158  sum      sum     [.] test_sum
#    0.45%   2.23%  38144426   98366818  sum      libc-2.23.so  [.] _int_malloc
...
```

Kilkum ważnym opcjom do perf report należy się wyjaśnienie:

- » --group włącza grupowanie zdarzeń – rozpatrywanie ich razem
- » --show-total-period włącza estymację ilości zdarzeń
- » --percent-limit <n> filzuje symbole z niskim udziałem zdarzeń podanym w procentach
- » --stdio wyłącza tryb interaktywny i drukuje na konsoli, co można przekierować do pliku
- » -n, --show-nr-samples wyświetla ilość zebranych próbek, im więcej, tym dokładniej.

Interesuje nas symbol `test_sum` i kolumna `Period`, która zawiera zebrane statystyki dla liczników `cycles` i `instructions`. Żeby obliczyć IPC, należy przygotować sobie odpowiedni arkusz kalkulacyjny albo użyć skryptu AWK wywołanego na zachowanym raporcie z pliku `report.txt`:

```
$ awk 'BEGIN {print "IPC SYMBOL"} !/#/ && $4 {print $4/$3,$8}
      ./report.txt | column -t
IPC           SYMBOL
0.46072      test_sum<std::cxx11::list ...
2.5788       _int_malloc
...'
```

Widać wyraźnie, że IPC dla różnych funkcji znacząco się różni. Dla porównania polecenie `perf stat` zliczy nam metrykę w sposób następujący:

```
perf stat -e cycles,instructions ./sum 1000000 1000
14195481283000

Performance counter stats for './sum 1000000 1000':
 8370174890      cycles
 4409400468      instructions      #  0,53  insns per cycle
 2,485884005 seconds time elapsed
```

Jak widać, metryka IPC liczona przez `perf stat`, czyli 0.53, ma wartość inną od metryki IPC dla funkcji `test_sum`, która wynosi 0.46. Funkcja sumująca zajmuje ponad 90% cykli, ale pozostałe 10% zaburzają wyniki pomiaru, prowadząc do znacznego błędu przy użyciu metody zliczającej za pomocą `perf stat`.

W dalszej części programu ograniczymy się do pokazywania tabelki z metrykami, pomijając skrypty i wydruki z konsoli. Czytelnicy dla własnych potrzeb mogą z łatwością zaadaptować skrypt AWK albo użyć arkusza kalkulacyjnego.

Nadszedł czas na zebranie metryk dotyczących pamięci i próbę oszacowania stopnia zużycia zasobów. Jako górny pułap zużycia uznamy wyniki z programu wzorcowego z Listingu 2.

PROGRAM	FUNKCJA	IPC	L1MissRate	MPKI	ADDS/CYCLE
sum	test_sum	0.46	25	125	0.1
benchmark	main	1.5	25	62	0.8

Tabela 1. Metryki programów z Listingu 1 i Listingu 2

W Tabeli 1 zestawiono metryki programów wzorcowego i testowego.

Zanim zaczniemy po kolej analizować te liczby, przypomnijmy, jak należy je odczytywać: porównujemy metryki z kilku programów > znajdujemy metrykę, która stanowi wyróżnik > zwracamy uwagę na wszystkie progi ostrożnościowe > dopasowujemy do naszego uproszczonego modelu procesora > wybieramy obszar działania procesora, na którym należy się skupić

Metryka IPC funkcji `test_sum()` wskazuje na to, że rdzeń procesora się „nudzi”, choć jego potencjał w sumowaniu liczb może być dużo większy. Funkcja `main()` dodaje 0.8 liczby na cykl i ma IPC 1.5, więc ewidentnie istnieje potencjał do poprawy od strony rdzenia. Należy szukać przyczyny, dlaczego rdzeń nie może działać wydajniej.

Metryka `L1MissRate` dla funkcji `test_sum()` wynosi 25%, czyli przekracza próg ostrożnościowy. Ta metryka ma identyczną wartość dla benchmarku, więc nie stanowi wyróżnika.

Metryka MPKI wskazuje, że 1 na 8 instrukcji generuje zdarzenie `L1-dcache-miss: 1000` promili / 125 = 8. Ta sama metryka dla benchmarku jest poniżej progu i 2 razy mniejsza. Różnica w wartości tej metryki sugeruje problemy z pamięcią. Gdyby MPKI była podobna, jak w programie wzorcowym, podejrzenie padłyby na rdzeń. W tym przypadku, patrząc na uproszczony model procesora z Rysunku 3, można wyciągnąć wniosek, że rdzeń jest niewykorzystany ze względu na zbyt wolno napływające dane.

Warto zapamiętać, że używanie każdej z tych metryk z osobna jest niewystarczające do wnioskowania na temat źródła problemów z wydajnością. Na przykład można sobie wyobrazić program, który ma wysoki `L1MissRate`, ale niski MPKI, w związku z tym dostęp do pamięci nie powinien być wąskim gardłem, jeśli wykonywany jest względnie rzadko. Ten sam problem dotyczy IPC. Niewielka wartość tej metryki nie wskazuje na źródło opóźnień. Może to być niekoniecznie pamięć, ale np. zła spekulacja, która w tym artykule została wymieniona tylko hasłowo. Dopiero zestawienie ze sobą kilku metryk i porównanie z innym, znany programem, pozwala na identyfikację problemu z dużą dozą pewności.

Rozstrzygnięcie, czy w tym przypadku ograniczeniem jest pasmo pamięci, czy latencja, może nastąpić nie-wprost: pasmo procesora jest wystarczające na wykonania co najmniej 0.8 dodawania/cykl, są więc dwie możliwości wyjaśnienia przyczyny wolnego działania programu z Listingu 1:

- » ograniczeniem jest latencja, czyli program jest tak skonstruowany, że głównie czeka na dane
- » jest potencjalny problem z pasmem, ale program „marnuje” pasmo na coś innego niż liczby, gdyż potencjalnie mógłby załadować dużo więcej liczb z pamięci

## U ŹRÓDŁA PROBLEMÓW

Na koniec analizy spójrzmy na fragment kodu źródłowego funkcji `test_sum()` poleconiem `perf annotate`:

```
:           _Self&
:           operator++() _GLIBCXX_NOEXCEPT
:           {
:               _M_node = _M_node->_M_next;
99.22 :           400ca4:    mov    (%rdx),%rdx
```

Przez 99% cykli wewnętrz funkcji wykonywana jest jedna instrukcja `mov`. Z kodu źródłowego widać, że jest to iterator listy. Najwyraźniej oczekiwane na załadowanie kolejnego elementu listy blo-

kuje cały program. Jest to zjawisko dobrze znane, co sugerowałem wcześniej, ale specjalnie odwlekałem ten moment, żeby najpierw opisać zagadnienie ilościowo. W praktyce programistycznej natrafimy zapewne na mnóstwo trywialnych programów, gdzie trudno będzie określić przyczynę problemu na podstawie jednej linijki kodu, więc warto przyswoić sobie metody analizy ilościowej.

Skoro źródłem problemu okazało się nieefektywne wykorzystanie pamięci *cache* przez listę, należy spróbować użyć struktury danych, która posiada lepsze właściwości. Podpowiedź leży w programie wzorcowym, który używała zwykłej tablicy. Idąc z duchem czasu, użyjmy klasy *vector*:

### Listing 3. Fragment programu z Listingu 1 z użyciem klasy vector

```
std::vector<long> list;
init(list, size);
long sum =
    test_sum(list, repetitions);
```

Spróbujmy teraz potwierdzić hipotezę, zbierając metryki dla nowej wersji programu:

PROGRAM	FUNKCJA	IPC	L1MissRate	MPKI	ADDS/CYCLE
sum	test_sum<list>	0.46	25	125	0.1
benchmark	main	1.5	25	62	0.8
sum2	test_sum<vector>	1.9	25	50	0.8

Tabela 2. Metryki dla programu wzorcowego (benchmark) i testowego z listą i wektorem

Udało się! Ogólny wynik funkcji *test<vector>* jest identyczny z benchmarkiem i wynosi 0.8 dodawania na cykl. Co jeszcze uległo zmianie?

- » IPC zwiększyła się, zbliżając się do wartości z benchmarku
- » MPKI jest mniejsze i porównywalne z benchmarkiem

Nasze podejrzenie, że winny był podsystem pamięci, sprawdziło się, gdyż wydajność programu wzrosła wraz z metryką MPKI. Jak wytłumaczyć jednak fakt, że benchmark nie działa lepiej, choć ma lepsze metryki?

Jednym z możliwych wytłumaczeń jest to, że programy różnią się nieznacznie wygenerowanym kodem, co jednak nie wpływa na szybkość działania programu. Musi istnieć jakieś dodatkowe wyjaśnienie, którego należy szukać w innych metrykach.

Nadal podejrzewamy, że ograniczeniem jest pamięć, dlatego że metryka *L1MissRate* przekracza próg ostrożnościowy. Używając metody „Dziel i rządź”, poszukamy innych metryk dotyczących pamięci. Podsystem pamięci składa się z 3 poziomów *cache* i z RAM. Dotychczas rozważaliśmy tylko *cache L1*, teraz można sięgnąć do RAM i „odkurzyć” metrykę *L3MissRate*.

Żeby poszukać dowodu na ograniczenie przez pamięć, można spróbować uruchomić program z różnymi rozmiarami tablicy i obliczyć metryki ponownie:

Rozmiar tablicy	IPC	L3MissRate	ADDS/CYCLE
1 000	3.2	0	1.36
20 000	3.1	0	1.36
200 000	3.0	0.2	1.25
700 000	2.7	20	1.11
900 000	1.97	45	0.82

Tabela 3. Metryki dla funkcji *test\_sum<vector>* dla różnych wielkości tablicy

W Tabeli 3 pokazano metryki wydajnościowe funkcji *test\_sum<vector>* w zależności od rozmiaru tablicy. Wyraźnie widać, że dla małych rozmiarów tablic osiągi są wyższe, ale sięgają maksymalnie 1.36 dodawania na cykl. W Tabeli 3 uwzględniono nową metrykę: *L3MissRate*, jako że metryka dla *cache* poziomu 1 nie stanowi dla testów wyróżnika, gdy zbiór danych nie mieści się w *cache L1* (32 kB).

## CZEMU TAK WOLNO?

Podsumowując dotychczasowe obserwacje, spróbujmy skonfrontować wyniki eksperymentalne z teoretycznymi szacunkami i wyciągnąć następującą tezę: zadanie sumowania tablicy jest ograniczone głównie przez dostępną przepustowość pamięci, przy czym wielkość ta rośnie, gdy zbiór danych maleje, co jest naturalne dla pamięci hierarchicznej (z wieloma poziomami *cache*). Przy tak prostym przetwarzaniu danych, czyli przy niewielkiej ilości obliczeń w stosunku do operacji na pamięci, głównym zadaniem programisty jest maksymalne wykorzystanie dostępnego pasma i linijkę *cache*, co naturalnie wiąże się z użyciem tablic. Zastosowanie struktur danych przyjaznych *cache* pozwala dodatkowo kompilatorowi na wygenerowanie instrukcji wektorowych (tzw. wektoryzacja pętli), co programista często otrzymuje jako „prezent” od kompilatora w nagrodę za użycie tablic.

Okazuje się, że to, co według programisty jest „pracą”, tj. dodawanie liczb, stanowi niewielkie obciążenie dla procesora w porównaniu z „wysiłkiem” związanym z transferem danych z pamięci. Intuicja programisty jest narzędziem nieocenionym, ale w przypadku zadania polegającego na szacowaniu wydajności wyniki często są sprzeczne z intuicją i zawsze należy szukać oparcia w eksperymentach i metrykach. Z drugiej strony metryki wymagają umiejętności doboru i interpretacji, co wskazuje na konieczność użycia solidnie ugruntowanej metodologii.

## W sieci

Strona Brendana Gregg'a poświęcona monitorowaniu wydajności w Linuksie:

- ▶ <http://www.brendangregg.com/linuxperf.html>

Metodologie znajdowania problemów z wydajnością:

- ▶ <http://www.brendangregg.com/methodology.html>

Jeden z lepszych opisów mikroarchitektury Haswell:

- ▶ <http://www.anandtech.com/show/6355/intels-haswell-architecture>



### MACIEJ CZEKAJ

mjc@semihalf.com

Programista systemów wbudowanych w krakowskiej firmie Semihalf, poprzednio w Motoroli. Zajmuje się w systemami wbudowanymi Linux/ARM, stosami sieciowymi i wirtualizacją. Współtworzył m.in. aplikację do zarządzania siecią SDN na bazie DPDK: Contrail Virtual Router.

# Najlepsze od



Odwiedź nas na:  
**IT.PWN.PL**

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

# CMS as a Service, czyli Umbraco w chmurze!

Jeszcze jakiś czas temu mówienie o czymkolwiek w chmurze świadczyło o byciu na czasie. Był to swoisty buzzword sprzedawczy, który miał mówić nam o klientom i partnerom biznesowym, że wiemy, co dzieje się w branży i podążamy nieustannie za jej rozwojem. Obecnie mamy do czynienia z sytuacjami, w których niemalże połowa Internetu przestaje funkcjonować i zostaje sparaliżowana, gdy „skromną” wpadkę zalicza jeden z kilku wiodących dostawców rozwiązań chmurowych. Uzależniamy się od chmury. Nie jest to jednak nic złego. Grunt to kontrolować to uzależnienie i czerpać z niego jak największe korzyści.

**U**mbraco miałem już okazję napisać kilka słów w poprzednim wydaniu magazynu „Programista”. Nie ulega wątpliwości, iż jest to bardzo potężne narzędzie/framework, które w odpowiednich programistycznych rękach może przysłużyć się do powstawania niesamowitych rozwiązań IT. Jeśli jednak jeszcze nie wiecie nic na jego temat – nie przejmujcie się – Umbraco Cloud to swego rodzaju nowy twór, który kierowany jest również do osób nie posiadających doświadczenia z jego prekursorem.

Tytułem małego wprowadzenia jednak: Umbraco to w pełni funkcjonalny opensource'owy system do zarządzania treścią stworzony w technologii ASP.NET. Dostępny jest za darmo, na licencji MIT i może być dowolnie dystrybuowany, rozszerzany i modyfikowany na potrzeby dowolnych rozwiązań i scenariuszy. To, co odróżnia go od całej gamy innych CMSów dostępnych na rynku (wg Wikipedii jest to niespełna 200 rozwiązań zbudowanych w różnych technologiach), to właśnie nowy model sprzedawczy SaaS (ang. *Software as a Service*). Usługa Umbraco Cloud to zupełnie nowość na rynku systemów do zarządzania treścią, opakowująca doświadczenia programistyczne w otoczkę administracyjną, możliwą do zarządzania za pomocą przycisków i przełączników – nawet przez zupełnych nowicjuszy!

## UAAS – UMBRACO CLOUD

Początkowa nazwa usługi – Umbraco as a Service – w swojej skrótowej wersji UaaS nie brzmiała nazbyt dobrze w komunikacji z anglojęzycznymi kontrahentami :). Stąd też decyzja o jej zmianie na Umbraco Cloud, czyli Umbraco w chmurze. Dokładnie to, czym jest usługa sama w sobie. Serwis powstał w wyniku chęci zapewnienia możliwości wykorzystywania Umbraco, okrykniętego wielokrotnie najlepszym systemem Open Source do zarządzania treścią w technologii .NET, przez osoby nietechniczne oraz niedoświadczone w kwestiach chociażby zarządzania serwerami. Nie każdy przecież jest w stanie wybrać, zakupić, a następnie skonfigurować środowisko hostingowe pod aplikację czy witrynę, po czym również wrzucić i uruchomić na nim nawet pudełkowy system do zarządzania treścią. To coś, na czym zarabia spora rzesza „programistów”, korzystających z braków umiejętności wśród osób chcących być po prostu widocznymi w sieci.

Nie dotyczy to jednak tylko osób nietechnicznych. W świecie dającym do automatyzacji na każdym kroku zdjęcie chociażby jednego z zadań ciążących na programistach na różnych szczeblach do-

świadczenie jest sporym ułatwieniem ich pracy. Nie wszyscy nawet bardziej doświadczeni programiści mają doświadczenie i wiedzę w zakresie procesów Continuous Integration czy Continuous Delivery. Czyż nie byłoby to sporym usprawnieniem, gdyby wszelkie kwestie administracyjne, integracyjne oraz wdrożeniowe zostały przejęte przez kogoś lub coś mającego o tym pojęcie, np. serwis wykorzystujący do tego sprawdzone mechanizmy dostępne w architekturze chmurowej? A co, jeśli to samo narzędzie zadba również o automatyczne aktualizacje CMSa oraz umożliwi nam transfer treści pomiędzy środowiskami w dowolnym momencie, w którym będziemy tego potrzebowali? Brzmi interesująco, prawda?

## JAK TO JEST ZROBIONE

Jakkolwiek skomplikowanie by to nie brzmiało, nie ma w tym żadnej magii. Przywołując słowa, które w trakcie swojego słynnego wystąpienia na Startup School w roku 2008 zabrzmiły z ust Davida Heinemeiera Hanssona, twórcy m.in. Basecampa czy frameworka Ruby on Rails – „It's not rocket surgery”.



Rysunek 1. Schemat architektury Umbraco Cloud

Umbraco Cloud działa z wykorzystaniem architektury Microsoft Azure i WebApps dostępnych w usłudze Azure App Service. Na bazie doświadczeń programistów tworzących samo narzędzie skonfigurowane i „wyżyłowane” środowisko dostępne jest dla każdej aplikacji stworzonej w panelu administracyjnym. Do każdej aplikacji dołączona jest baza danych Azure SQL oraz repozytorium Git, o którym nieco więcej wspomnę w dalszej części artykułu. Czy

# DLACZEGO WARTO ZOSTAĆ CERTYFIKOWANYM DEVELOPEREM UMBRACO?

- ✓ Jest to w pełni funkcjonalny, OpenSource'owy CMS w technologii .NET
- ✓ Skoncentrowany na edytorach i ich zadowoleniu z użytkowania systemu, ale również uwielbiany przez developerów
- ✓ Ponad 400 000 aktywnych instalacji na całym Świecie, w tym witryny takich firm jak Microsoft i Carlsberg
- ✓ Niesamowicie łatwy w rozszerzeniu o dowolne, własne komponenty bądź logikę, prosty do zintegrowania z własnym designem



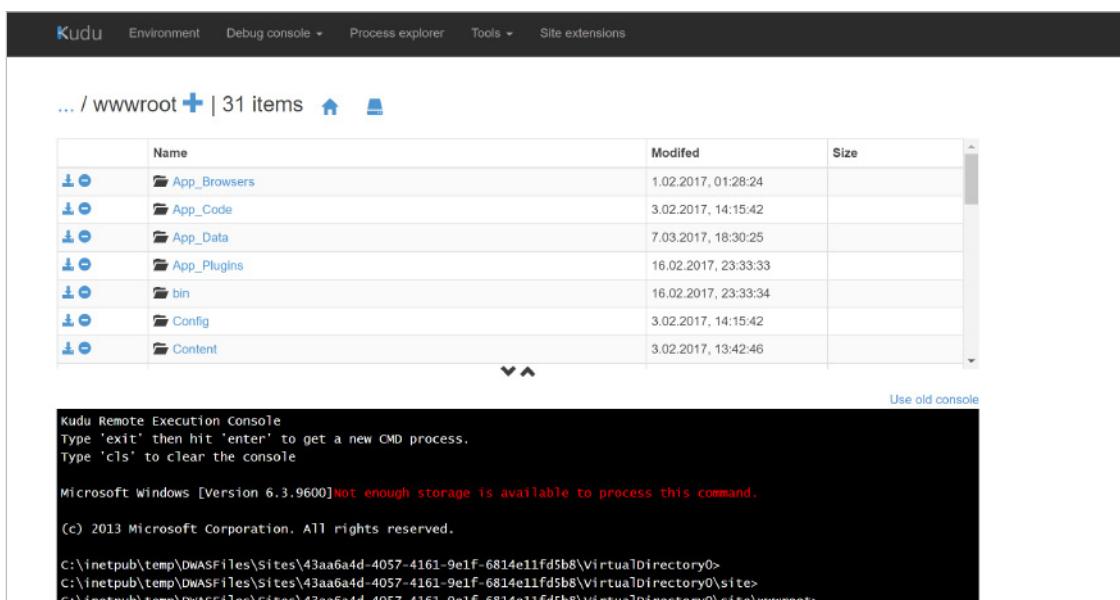
Oficjalny tytuł "Umbraco Certified" daje gwarancję, iż osiągnąłeś określony poziom umiejętności i wiedzy z zakresu tworzenia rozwiązań z wykorzystaniem Umbraco. Ukończenie kursów zatem, nie tylko czyni Twoją pracę bardziej efektywną i przyjemną, ale również odróżnia Cię od Twoich konkurentów i otwiera drogę na nowe możliwości.

## SZKOLENIA DOSTĘPNE W POLSCE

- Umbraco Fundamentals
- Umbraco, Mvc i Visual Studio
- Integracja aplikacji z Umbraco
- Rozszerzanie panelu administracyjnego z wykorzystaniem AngularJS



Aby dowiedzieć się więcej bądź zarejestrować się na szkolenia, skontaktuj się z nami na [info@thecogworks.pl](mailto:info@thecogworks.pl) lub odwiedź stronę [www.umbraco.com/training](http://www.umbraco.com/training).



Rysunek 2. Aplikacja KUDU umożliwiająca m.in. przegląd plików w obrębie witryny oraz wykorzystywanie konsoli na serwerze, na którym zainstalowana jest aplikacja

musimy wiedzieć więcej na temat architektury? Nie, gdyż w chwili obecnej nie mamy na to zupełnie wpływu (co jest zarówno pozytywnym aspektem, jak i jedną z wad całego ekosystemu).

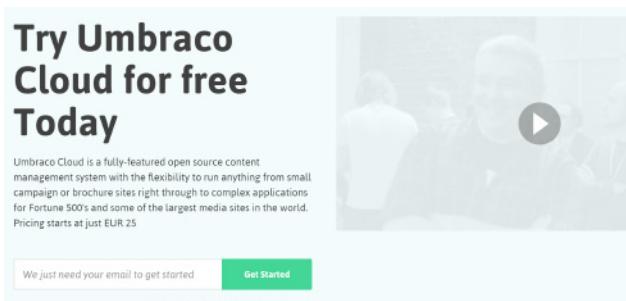
Dostęp do systemu plików dostępny jest za pomocą KUDU – webowego narzędzia, które umożliwia m.in.:

- » przeglądanie i edycję plików w obrębie aplikacji,
- » wywoływanie poleceń z konsoli,
- » korzystanie z PowerShella,
- » monitoring działających na serwerze procesów wraz z dostępem do informacji nt. środowiska i zdarzeń mających na nim miejsce.

W obrębie KUDU udostępnione zostało również RESTowe API, dzięki któremu wyżej opisywane dane mogą być udostępniane i przetwarzane przez dowolne narzędzia i aplikacje, dla których umożliwimy dostęp do KUDU (Rysunek 2).

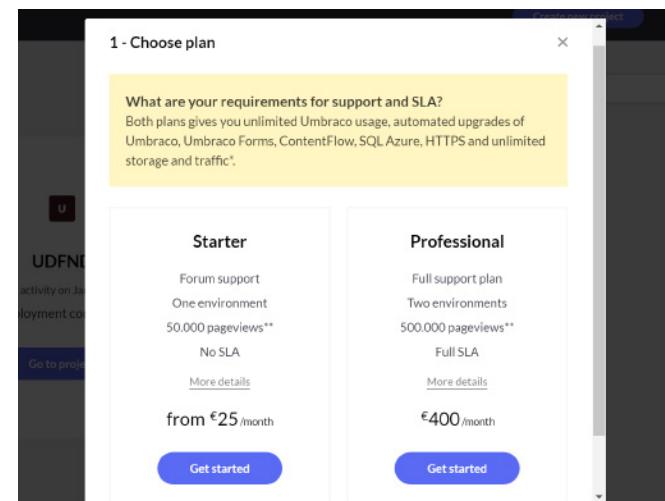
## JAK ZACZĄĆ

To proste. Wystarczy odwiedzić witrynę <http://umbraco.com/>, podać adres e-mail, na który chcemy zarejestrować konto, oraz wypełnić podstawowe dane personalne takie jak nazwa użytkownika czy hasło. Pierwszy projekt w Umbraco Cloud oferuje nam bezpłatny, 14-dniowy okres próbny. Zarówno w jego trakcie, jak i po jego zakończeniu zawsze możemy przerwać subskrypcję i nie będziemy w żaden sposób obciążeni jakimkolwiek kosztami, o ile nie uruchomimy kolejnych projektów, za które musimy zapłacić miesięcznie kwotę min. 25 EUR per środowisko.



Rysunek 3. Ekran rejestracji do wersji trial Umbraco Cloud (<http://umbraco.com>)

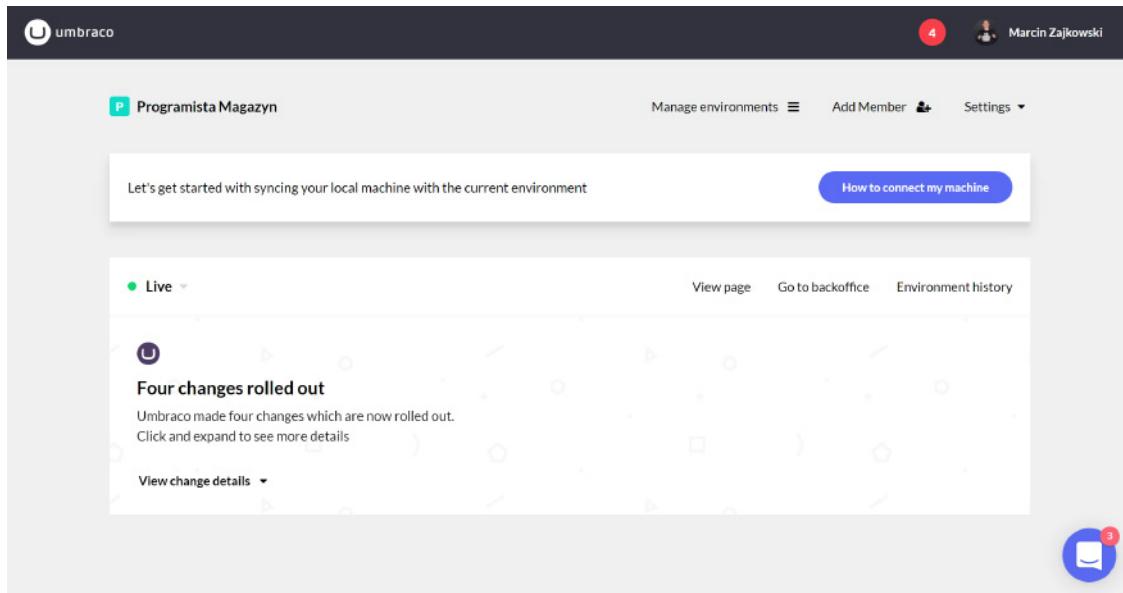
Po zalogowaniu jesteśmy skierowani do ekranu głównego aplikacji, na którym widnieje listing utworzonych przez nas projektów z możliwością przeskakiwania do ich szczegółów oraz konfiguracji. Z poziomu każdego ekranu mamy także możliwość utworzenia nowego projektu.



Rysunek 4. Wybór planu dla nowotworzonego projektu

Tworząc go, w pierwszym kroku podajemy jego nazwę oraz wskazujemy tzw. *Baseline* (opcjonalnie). Umbraco Cloud daje nam możliwość tworzenia podprojektów uzależnionych od projektu bazowego. Mając wiele witryn korzystających z tego samego typu danych, bądź funkcjonalności, staje się to doskonałą opcją do zapanowania nad wspólnie dostępnym kodem, który może być aktualizowany globalnie – raz dla wszystkich dziedziczących go aplikacji. Projekty bazowe wykorzystywane są m.in. przez firmę Carlsberg, która jest w tej chwili największą marką, której witryny działają na Umbraco Cloud. Firma posiada kilkadziesiąt witryn w kilkunastu marketach językowych opartych o jeden, wspólnie dostęgowy projekt bazowy.

Po zatwierdzeniu chcemy utworzenia nowego projektu w ciągu kilkunastu sekund, maksymalnie kilku minut, otrzymujemy skonfigurowane i gotowe środowisko *Live*, na którym docelowo funkcjonować będzie nasza produkcyjna witryna.

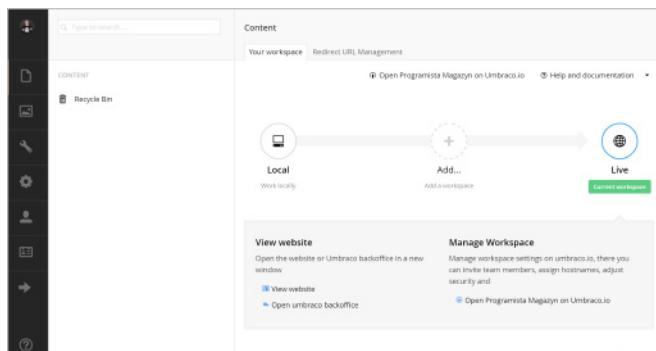


Rysunek 5. Ekran główny projektu w UC

Z poziomu tego widoku mamy możliwość zarządzania pełną konfiguracją naszego projektu. Możemy utworzyć dodatkowe środowisko developerskie, które może być przez nas wykorzystywane na cele prac programistycznych przed uruchomieniem projektu bądź już przy jego rozwoju. Co ciekawe, środowisko możemy usuwać i tworzyć w dowolnym momencie i równie dynamicznie jesteśmy za nie rozliczani – płacimy tylko wtedy, kiedy jest nam ono potrzebne i jest rzeczywiście użytkowane. W wielu projektach nie będzie ono nawet konieczne, gdyż już w tym momencie mamy dostęp do naszej standardowej instancji Umbraco (*Live*), w której możemy już skonfigurować naszą witrynę oraz udostępnić ją światu.

Do projektu możemy dołączać nowe osoby, dając im tym samym dostęp do zarządzania nim w określonych dla nadawanej im roli przywilejach i zakresie.

Nasza witryna jest gotowa do dalszej pracy. Możemy zalogować się do panelu administracyjnego Umbraco (wykorzystując dane konta tworzonego podczas rejestracji). Jego ekran startowy nieco odbiega od standardowego. Głównym ekranem jest tutaj wgląd w informacje na temat przepływu danych pomiędzy instancjami witryny wraz z możliwością przełączania się pomiędzy trybami pracy w obrębie każdej z nich.



Rysunek 6. Widok startowy panelu administracyjnego nowego projektu Umbraco Cloud

## PRACA NA LOKALNYCH ŹRÓDŁACH

Istnieje kilka sposobów na rozpoczęcie pracy w środowisku lokalnym z witryną, którą właśnie uruchomiliśmy w chmurze.

The diagram is titled 'Start working with your website locally' and explains that it makes changes locally and easily sends them to UC. It shows three options: 'Use Visual Studio' (with an infinity icon), 'Use Grunt or Gulp' (with a monitor icon), and 'Connect with git' (with a GitHub icon).

Rysunek 7. Sposoby na pracę lokalną z projektem Umbraco Cloud

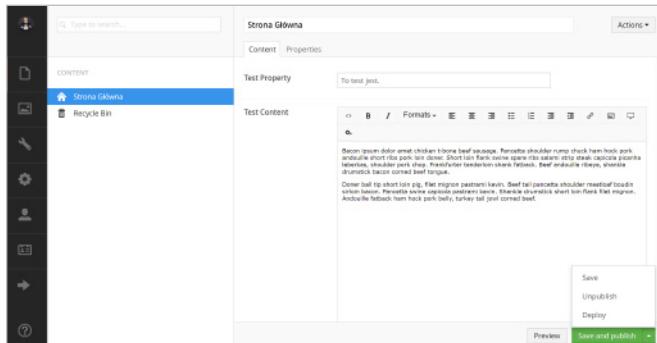
Bez względu na to, czy jesteśmy power userami Visual Studio czy też frontend developerami chcącymi jedynie namieszać w stylach bądź skryptach witryny, każdy ma stosunkowo komfortową opcję na pracę ze źródłami projektu UC. Zawsze również mamy możliwość wpięcia się do repozytorium Git tworzonego na potrzeby każdego ze środowisk.

Repozytorium to różni się nieco od typowych repozytoriów projektowych ASP.NET bądź dowolnych projektów tworzonych w Visual Studio. Nie zawiera bowiem żadnych plików projektu VS, a jedynie wynikową witrynę (i to wraz z artefaktami!), która jest już skompilowaną aplikacją webową uruchamianą docelowo na serwerze. Powoduje to początkowy szok i niedowierzanie doświadczeni programistów, ale jest to jedno z niewielu rozwiązań pozwalających na zachowanie spójności i sprawnego przepływu danych pomiędzy instancjami naszych witryn. W przypadku złożonych solucji i rozwiązań to, o co musimy zadbać, to to, aby w tym właśnie repozytorium lądowała jedynie nasza skompilowana wersja witryny. Jej końcowy rezultat. Cała magia prowadząca do jego powstania może być przechowywana w oddzielnym repozytorium. Nie stanowi to żadnego problemu.

W przypadku kilku z naszych projektów wykorzystywaliśmy build server (TeamCity) do tego, aby przetwarzał kod źródłowy z naszego repozytorium i w postaci paczki wdrożeniowej „pushował” go do docelowego repozytorium Umbraco Cloud, celem jego wyłapania po stronie serwisu. Nie ma tutaj rzeczy niemożliwych! Proces ten, wraz z przykładową konfiguracją, doskonale opisał na swoim blogu jeden z developerów z Umbraco HQ – Sebastian Janssen: <https://cultiv.nl/blog/visual-studio-and-umbraco-as-a-service/>.

## TRANSFER ZMIAN ORAZ PRZEPŁYW CONTENTU

Celem zobrazowania prostoty całego procesu przesyłu danych pomiędzy środowiskami musimy utworzyć przykładowe treści, wykorzystując do tego środowisko lokalne bądź developerskie. Po dodaniu przykładowego typu danych oraz wypełnienia go przykładową treścią od razu po jego zapisie w opcjach dostępnych w dokumencie pojawiła się opcja *Deploy*.



Rysunek 8. Opcja transferu treści do kolejnego środowiska aplikacji

Po wyborze opcji dane są zbierane w kolejkę i możliwe do przekazania „dalej”. Informacje o tym dostajemy również wyświetlona na startowym ekranie naszego panelu administracyjnego.

Nie jest to jednak możliwe do czasu, gdy nasza produkcyjna bądź „ kolejna” instancja aplikacji nie posiada aktualnego kodu źródłowego. Informację o tego typu zmianach jak np. nowe typy dokumentów zawarte są w panelu konfiguracji aplikacji Umbraco Cloud.

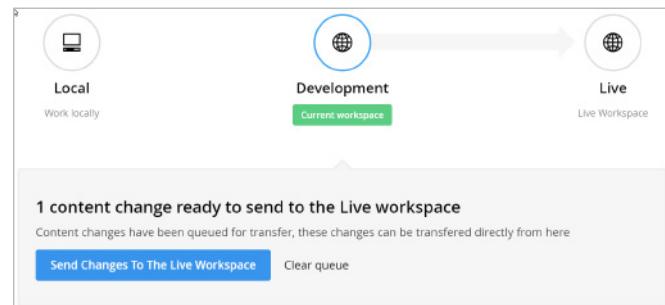


Rysunek 9. Informacja o zmianach w obrębie źródeł aplikacji

Po kliknięciu przycisku *Deploy changes to Live* zmiany są natychmiastowo przesyłane na produkcyjną wersję aplikacji. Jest to nic innego niż commit w Git, który zostaje wypchnięty do repozytorium przypisanego do wersji aplikacji *Live*.

Zmiany konfiguracyjne mamy już w środowisku *Live*, możemy zająć się transferem treści. Informacje o elementach możliwych do

przesłania mamy na ekranie startowym panelu administracyjnego Umbraco.



Rysunek 10. Informacja o możliwym transferze zmian do produkcyjnej wersji aplikacji

Wystarczy, iż wywołamy akcję przyciskiem i potwierdzimy ją w kolejnym kroku, po rewizji tego, co zostanie przesłane. Voila! Zmiany, które wprowadziliśmy na wersji developerskiej witryny, zostały wdrożone na jej produkcyjnej instancji. Bez żadnego podłączania się do serwerów, wykonywania zaawansowanych operacji tudzież nerwów i stresów zwykle z tym związanych.

## PODSUMOWANIE

Rozwiązań oparte na architekturze w chmurze nie są już niczym unikalnym. Startupy prześcigają się w sprzedaży, a czasem nawet agresywnym wciskaniu narzędzi sprzedawanych w modelu *Software as a Service*. Odnalezienie w tym stosie oprogramowania, które jest użyteczne, możliwe do rozszerzania – wygodne dla developerów oraz względnie tanie w utrzymaniu – jest bardzo trudne.

Może z uwagi na to, iż Umbraco przede wszystkim pozostaje darmowym systemem z otwartym kodem źródłowym, który jedynie w postaci usługi Umbraco Cloud sprzedaje mechanizmy wspomagające procesy zarządzania wdrożeniami oraz utrzymaniem aplikacji, czyni go wyjątkowym w tym towarzystwie. Bowiem za kwotę niższą niż zakup tego samego typu zasobów bezpośrednio w usłudze Microsoft Azure mamy możliwość stworzenia i zarządzania aplikacją internetową ASP.NET MVC, w której bazę i swoisty framework stanowi najlepszy na rynku system do zarządzania treścią w technologii .NET, który dodatkowo posiada ogólnodostępny kod źródłowy. Czego chcieć więcej? Jeśli naszym celem jest stworzenie aplikacji internetowej, znamy podstawy C# i ASP.NET lub chcemy zapoznać się z tą technologią – warto spróbować. To nic nie kosztuje (przez 14 dni :)).

## W sieci:

- Oficjalna dokumentacja Umbraco Cloud: <https://our.umbraco.org/documentation/Umbraco-Cloud/>
- Informacje nt. KUDU: <https://azure.microsoft.com/nl-nl/resources/videos/what-is-kudu-with-david-ebbo/>
- Umbraco Cloud FAQ: <https://umbraco.com/products/umbraco-cloud/umbraco-cloud-faq/>
- Szkolenia certyfikacyjne Umbraco w Polsce: <https://certyfikacijaumbraco.evenea.pl/>



### MARCIN ZAJKOWSKI

[marcin.zajkowski@thecogworks.com](mailto:marcin.zajkowski@thecogworks.com)

Certyfikowany trener i pierwszy w Polsce Umbraco Certified Master. Na co dzień pracuje jako Senior Umbraco Developer w The Cogworks, organizuje spotkania grup społeczności w Polsce oraz ewangelizuje Umbraco, prowadząc szkolenia i prezentacje na jego temat podczas wszelkiego rodzaju wydarzeń IT.



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)

# Microsoft Bot Framework

## Tworzenie inteligentnych kanałów komunikacyjnych

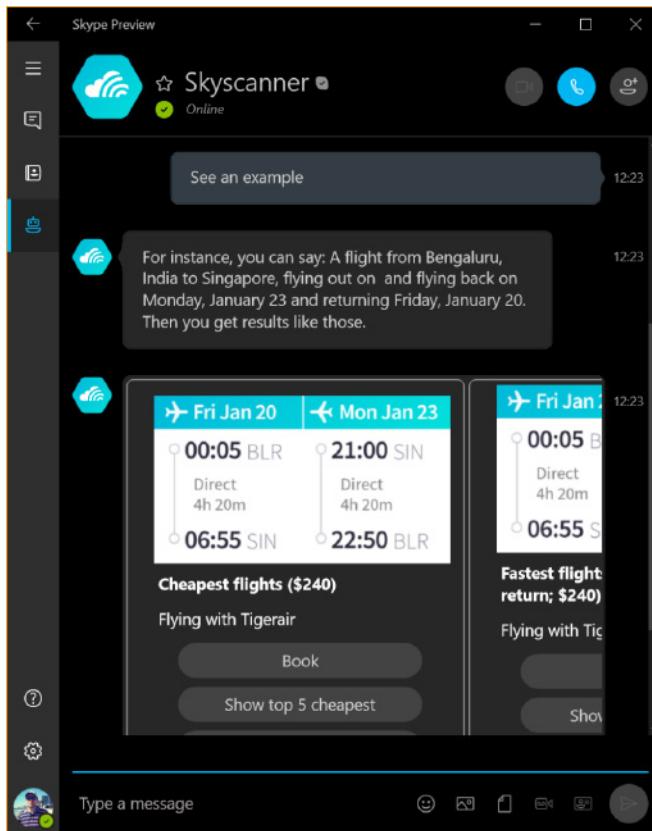
Nie da się ukryć, że dzięki wynikom wielu lat prac badawczych nad sztuczną inteligencją oraz zaawansowanymi metodami przetwarzania sygnałów i obrazów jesteśmy teraz beneficjentami nowoczesnych kanałów komunikacyjnych. Przystaje już dziwić fakt, że urządzenia mobilne mogą być sterowane głosem, a odczytanie swojej lokalizacji czy wyszukanie informacji realizuje się po wypowiedzeniu odpowiedniej komendy poprzedzonej frazą „Hey, Siri”. Z kolei zarezerwowanie biletu lotniczego można wykonać za pomocą kilku wiadomości wysłanych za pośrednictwem Skype czy Messengera do odpowiedniego bota. W tym artykule zostanie przedstawione, w jaki sposób utworzyć tego typu kanał komunikacyjny w oparciu o Microsoft Bot Framework.

### WPROWADZENIE

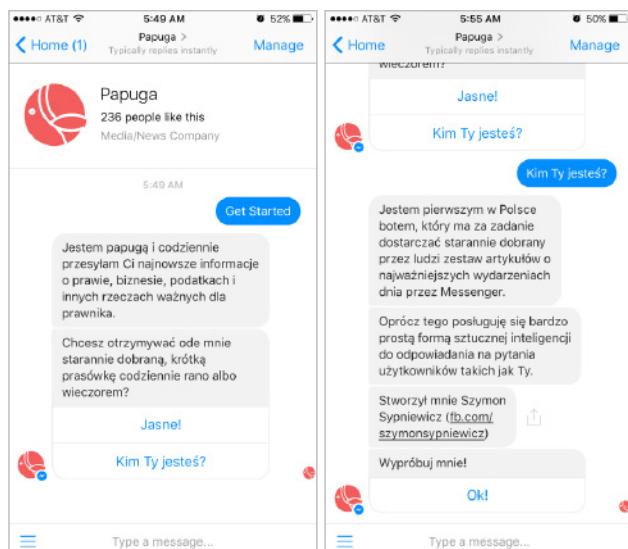
Formalnie bot jest definiowany jako aplikacja, która realizuje czynności w sposób automatyczny. Pierwsze boty można było spotkać w czasach popularności usługi Internet Relay Chat (IRC). Boty „pilnowały” zazwyczaj porządku na kanale i usuwały z niego użytko-

kowników posługujących się wulgarnym słownictwem. Wraz z rozwojem technologii oraz upowszechnianiem się technik sztucznej inteligencji dzisiejsze boty, podobnie jak i inne aplikacje, charakteryzują się zaawansowaną funkcjonalnością. Oprócz rozpoznawania brzydkich wyrazów boty rozumieją język naturalny, co umożliwia im prowadzenie dyskusji z użytkownikami.

Programowanie botów staje się coraz bardziej popularne. Już teraz można spotkać kilkudziesiąt tego typu aplikacji, chociaż dla Skype czy Facebook Messenger. Na Rysunku 1 zilustrowano przykładowego bota Skyscanner w aplikacji Skype Preview (Windows 10). Natomiast na Rysunku 2 przedstawiłem kilka ilustracji polskiego bota Papuga w aplikacji Facebook Messenger na iOS 10.



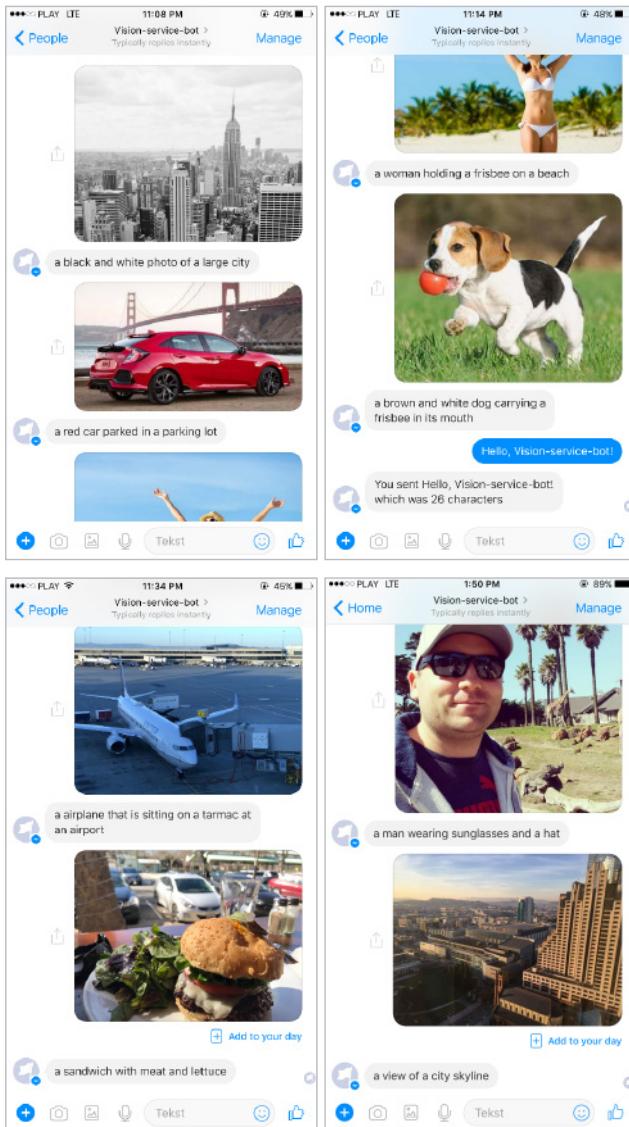
Rysunek 1. Przykład działania nowoczesnego bota Skyscanner do wyszukiwania biletów lotniczych



Rysunek 2. Bot Papuga w trakcie działania w aplikacji Facebook Messenger na platformie iOS 10

W tym artykule pokażę techniki programowania inteligentnych botów z wykorzystaniem Microsoft Bot Framework, Visual Studio

oraz chmury Azure. Chmura będzie nam potrzebna do publikowania bota, a ponieważ zagadnienia tworzenia i publikacji aplikacji w Azure nie były prezentowane na łamach „Programisty”, to artykuł rozpoczynam od szczegółowego zilustrowania kolejnych etapów tworzenia aplikacji webowej w Azure. Następnie omówię szablon aplikacji *Bot\_Application*, który umożliwia bardzo szybkie zapoznanie się z podstawowymi elementami tworzenia botów w oparciu o Microsoft Bot Framework. W ostatnim etapie uzupełnię funkcjonalność bota o rozpoznawanie zawartości obrazów. Przykłady ostatecznej funkcjonalności bota, którego implementację opiszę w tym artykule, zilustrowano na Rysunku 3.

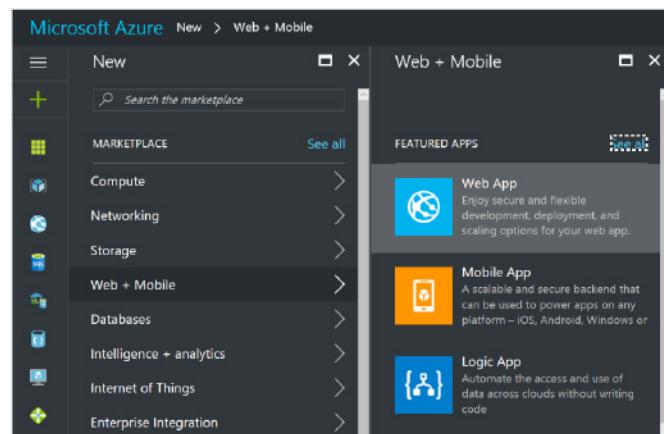


Rysunek 3. Przykład działania bota *Vision-service-bot* na urządzeniu mobilnym z systemem iOS 10

## APLIKACJA AZURE

Wykorzystanie chmury Azure wymaga subskrypcji. Dostępne opcje zostały już opisane na łamach „Programisty”. Jednak najprostszym sposobem jest utworzenie bezpłatnego konta Azure. Takie konto uprawnia do 30-dniowego dostępu do Azure z kredytem w kwocie €170 lub \$200 (w zależności od lokalizacji). W tym artykule będę korzystał z próbowej subskrypcji Azure Pass, w której kwoty usług będą podane w USD.

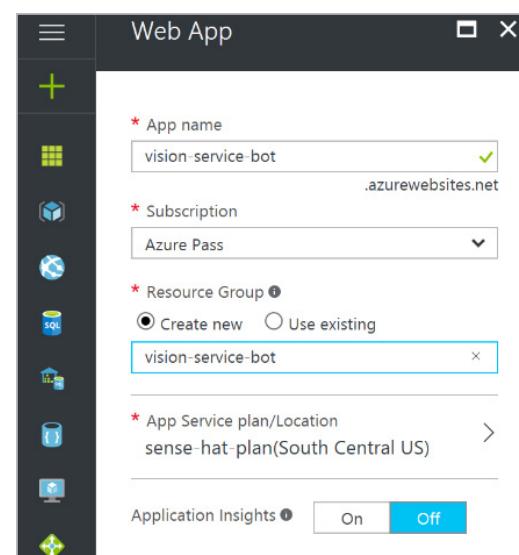
Po zalogowaniu do Azure (<https://portal.azure.com>) wystarczy kliknąć ikonę , która znajduje się w lewym górnym rogu UI portalu Azure. W efekcie nastąpi uaktywnienie menu Marketplace, gdzie z listy dostępnych opcji rozwijamy węzeł Web + Mobile, a następnie klikamy Web App (Rysunek 4). Spowoduje to uaktywnienie odpowiedniego kreatora, który umożliwia skonfigurowanie nazwy aplikacji, zasobów oraz planu App Service (Rysunek 5).



Rysunek 4. Tworzenie serwisu w Azure portal

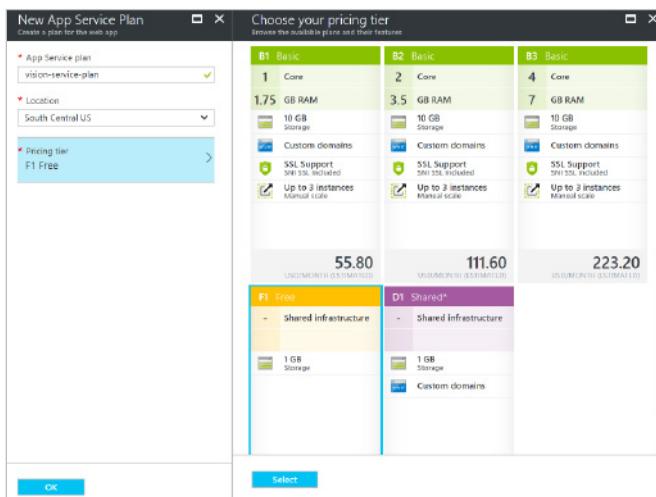
W tym przykładzie zmieniłem nazwę aplikacji na *vision-service-bot*, a także utworzyłem nową grupę zasobów o tej samej nazwie. Grupa zasobów ułatwia zarządzanie poszczególnymi elementami Azure. Jest to szczególnie przydatne w sytuacji, gdy zarządzamy dużą liczbą usług w Azure.

Jeśli chodzi o App Service plan, to determinuje on fizyczne zasoby sprzętowe oraz opcje konfiguracyjne, które będą dostępne dla naszej aplikacji. Jeśli nie mamy utworzonego planu App Service, możemy go utworzyć, klikając na opcję *App Service plan/Location* w kreatorze Web App z Rysunku 5. Spowoduje to uaktywnienie listy dostępnych planów, na której wystarczy kliknąć przycisk z etykietą *Create New*. Uzyskamy wówczas możliwość ustalenia nazwy planu, jak również poziomu cenowego (Pricing tier). Na potrzeby tego artykułu utworzyłem plan o nazwie *vision-service-plan*, a Pricing tier ustawiłem na F1 Free (Rysunek 6).



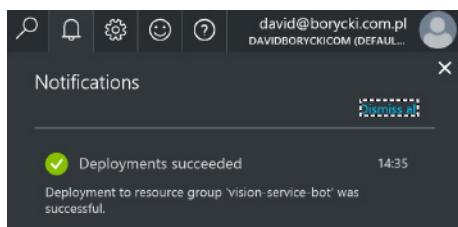
Rysunek 5. Konfiguracja aplikacji webowej

# PROGRAMOWANIE APLIKACJI WEBOWYCH



Rysunek 6. Konfiguracja App Service Plan

Po utworzeniu lub wybraniu App Service plan, w kreatorze z Rysunku 5 wystarczy kliknąć przycisk *Create*. Spowoduje to rozpoczęcie procesu tworzenia aplikacji webowej, co potrwa nie dłużej niż kilkanaście sekund. O zakończeniu tego procesu zostaniemy poinformowani za pomocą notyfikacji Azure. Te notyfikacje są prezentowane w prawym górnym rogu UI portalu Azure (Rysunek 7).

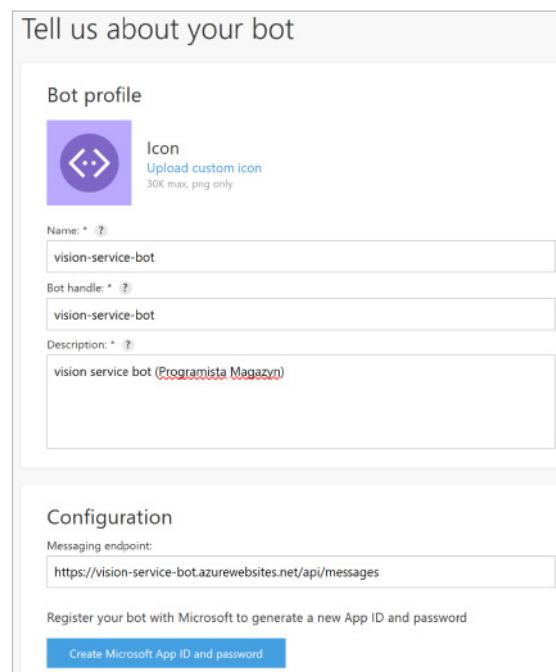


Rysunek 7. Fragment obszaru notyfikacji portalu Azure

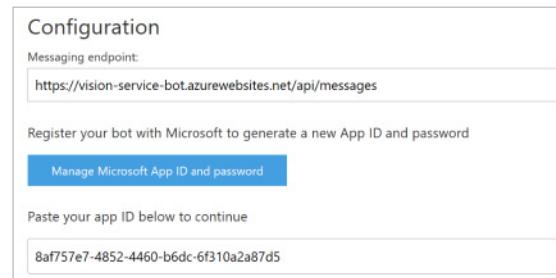
Mamy już przygotowane środowisko hostingowe dla bota. W kolejnym kroku należy go zarejestrować, a następnie zaimplementować.

## REJESTRACJA

Aby zarejestrować bota, wystarczy skorzystać z witryny: <https://dev.botframework.com/bots/new>, gdzie po zalogowaniu pojawi się kreator pokazany na Rysunku 8. W tym kreatorze konfigurujemy nazwę bota, jego uchwyty, opis oraz adres serwisu REST (messaging endpoint). W tym przykładzie nazwę oraz uchwyty ustawiłem na *vision-service-bot*. Natomiast w polu *messaging endpoint* wpisałem adres URL aplikacji, którą utworzyłem w poprzednim podrozdziale (<https://vision-service-bot.azurewebsites.net>) uzupełnioną o /api/messages. Ten ostatni fragment URL, jak się za chwilę okaże, wynika z domyślnej konfiguracji Web API szablonu aplikacji *Bot\_Application* dla Visual Studio. Domyślnie ten szablon wykorzystuje mechanizm autoryzacji z Microsoft Bot Framework. W związku z tym, aby móc nawiązać komunikację z punktem końcowym bota, potrzebne będą dwa łańcuchy: identyfikator Microsoft App oraz hasło. Obie te wartości można wygenerować podczas rejestracji bota, klikając przycisk z etykietą *Create Microsoft App Id and password*. Spowoduje to uaktywnienie kolejnego okna przeglądarki prezentującego identyfikator aplikacji oraz przycisk umożliwiający wygenerowanie hasła. Po zanotowaniu hasła zamykamy to okno z wykorzystaniem dedykowanego przycisku. W efekcie odpowiednie pola kreatora rejestracji bota zostaną automatycznie uzupełnione, jak to zilustrowano na Rysunku 9.

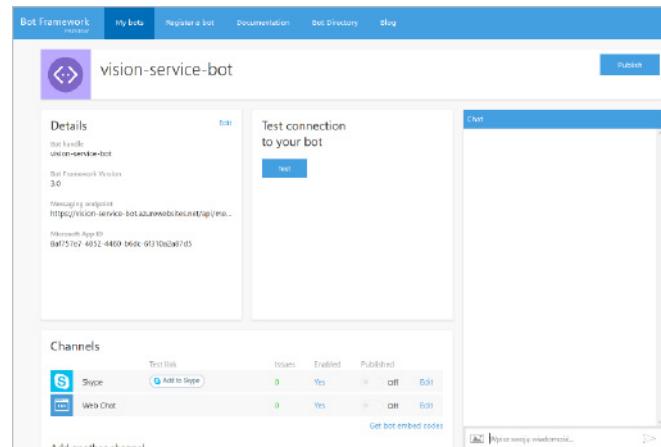


Rysunek 8. Rejestracja bota



Rysunek 9. Konfiguracja punktu końcowego bota

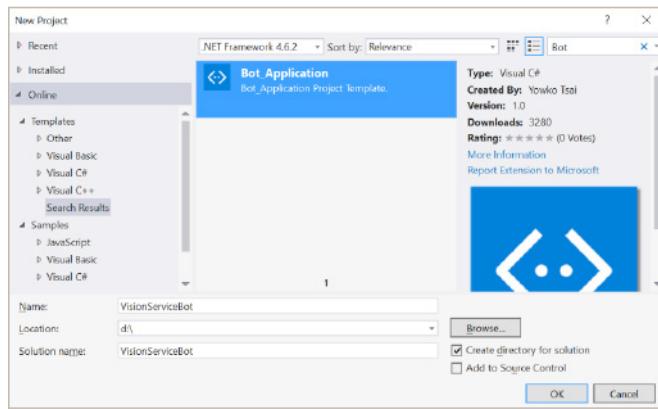
Po skonfigurowaniu wymaganych parametrów trzeba jeszcze zaakceptować warunki serwisu, a następnie potwierdzić rejestrację, klikając przycisk z etykietą *Register*. W efekcie nastąpi utworzenie bota i automatyczne przekierowanie do zakładki *My Bots* serwisu Microsoft Bot Framework (Rysunek 10). Mamy tam możliwość przetestowania komunikacji z botem, jego publikacji, jak również jego opublikowania oraz utworzenia kanałów dla różnych aplikacji, np. Skype czy Facebook Messenger.



Rysunek 10. Parametry vision-service-bota w portalu Microsoft Bot Framework

## IMPLEMENTACJA

Od strony konfiguracyjnej mamy już wszystko gotowe. Przejdźmy teraz do zaimplementowania bota. Do tego celu można wykorzystać gotowy szablon projektu *Bot\_Application*. Jest on dostępny dla Visual Studio 2015. Aby znaleźć ten szablon, wystarczy w kreatorze tworzenia nowego projektu w Visual Studio przejść na zakładkę *Online*, a następnie w polu wyszukiwania wpisać „Bot”. Po chwili ukaże się lista znalezionych wyników, która powinna zawierać wyłącznie jeden element: *Bot\_Application* (Rysunek 11). Dalsze kroki wyglądają standardowo. Mianowicie wystarczy jedynie zmienić nazwę projektu i wskazać folder, w którym zostanie on zapisany. Po kliknięciu przycisku z etykietą *OK* szablon projektu zostanie pobrany. W pewnym momencie zostaniemy poproszeni o akceptację instalacji dodatku do Visual Studio 2015, po czym zostanie utworzony nowy projekt aplikacji.



Rysunek 11. Parametry vision-service-bot'a w portal Microsoft Bot Framework

Utworzony projekt ma strukturę charakterystyczną dla aplikacji ASP.NET MVC i zawiera jedną statyczną stronę *default.htm*. Prezentuje ona jedynie widok powitalny.

Zasadniczym elementem projektu jest kontroler Web API, zaimplementowany w ramach klasy *MessagesController* (Listing 1). Widzimy, że klasa ta dziedziczy po *System.Web.Http.ApiController*, a jej deklaracja jest udekorowana atrybutem *BotAuthentication* z przestrzeni nazw *Microsoft.Bot.Connector*. *BotAuthentication* jest elementem pakietu NuGet o nazwie *Microsoft.Bot.Builder* (<https://github.com/Microsoft/BotBuilder>), który jest również określany mianem Microsoft Bot Builder SDK. Chociaż w tym przypadku wykorzystałem gotowy szablon, to dla istniejących aplikacji lepszym rozwiązaniem wydaje się uzupełnienie projektu aplikacji o Microsoft Bot Builder SDK.

**Listing 1. Domyślny kontroler Web API aplikacji VisionServiceBot**

```
[BotAuthentication]
public class MessagesController : ApiController
{
    /// <summary>
    /// POST: api/Messages
    /// Receive a message from a user and reply to it
    /// </summary>
    public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
    {
        if (activity.Type == ActivityTypes.Message)
        {
            ConnectorClient connector = new ConnectorClient(
                new Uri(activity.ServiceUrl));
```

```
// calculate something for us to return
int length = (activity.Text ?? string.Empty).Length;

// return our reply to the user
Activity reply = activity.CreateReply(
    $"You sent {activity.Text} which was {length} characters");

await connector.Conversations.ReplyToActivityAsync(reply);
}
else
{
    HandleSystemMessage(activity);
}

var response = Request.CreateResponse(HttpStatusCode.OK);

return response;
}

private Activity HandleSystemMessage(Activity message)
{
    if (message.Type == ActivityTypes.DeleteUserData)
    {
        // Implement user deletion here
        // If we handle user deletion, return a real message
    }
    else if (message.Type == ActivityTypes.ConversationUpdate)
    {
        // Handle conversation state changes, like members
        // being added and removed. Use Activity.MembersAdded
        // and Activity.MembersRemoved and
        // Activity.Action for info. Not available in all channels
    }
    else if (message.Type == ActivityTypes.ContactRelationUpdate)
    {
        // Handle add/remove from contact lists
        // Activity.From + Activity.Action represent what happened
    }
    else if (message.Type == ActivityTypes.Typing)
    {
        // Handle knowing that the user is typing
    }
    else if (message.Type == ActivityTypes.Ping)
    {
    }

    return null;
}
```

Na definicję klasy *MessagesController* składa się jedna publiczna funkcja implementująca obsługę metody POST, a także jedna prywatna metoda *HandleSystemMessage*. Żądania typu POST są właśnie zapytaniemami użytkowników wysyłanymi do bota. Są one reprezentowane przez instancje klasy *Activity*. W szczególności klasa posiada pola takie jak *Text* czy *ServiceUrl*. Pierwsze z nich zawiera tekst wiadomości otrzymanej od użytkownika. Z kolei *ServiceUrl*, jak wskazuje na to nazwa, zawiera adres URL punktu końcowego serwisu bota. Jest on wykorzystywany do nawiązania połączenia z klientem w oparciu o klasę *ConnectorClient*. Służy ona również do wysyłania odpowiedzi do użytkowników.

W przypadku domyślnej implementacji odpowiedź ma postać `$"You sent {activity.Text} which was {length} characters"`, gdzie `{activity.Text}` to wiadomość otrzymana od użytkownika, a `length` określa liczbę znaków tej wiadomości. Tak skonstruowana odpowiedź jest następnie wysyłana z wykorzystaniem metody *ReplyToActivityAsync*.

Taka domyślna obsługa zapytań dotyczy wyłącznie wiadomości od użytkownika. Dodatkowo w ramach instancji klasy *Activity* zgłoszane są również inne zdarzenia, takie jak chociażby informacja o tym, że użytkownik pisze wiadomość. Tego typu zdarzenia mogą być obsłużone poprzez rozszerzenie definicji prywatnej metody *HandleSystemMessage*.

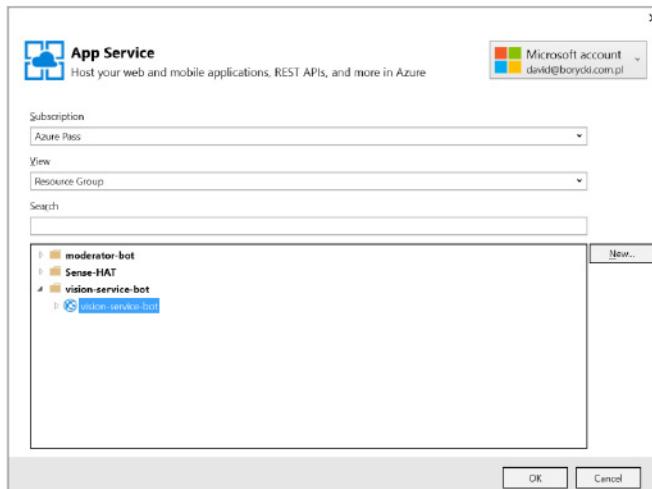
## PUBLIKACJA

Aplikacja utworzona w oparciu o szablon *Bot\_Application* jest w zasadzie gotowa do publikacji. Jedyne, co wcześniej należy zrobić, to w pliku *web.config* wpisać identyfikator bota oraz hasło (Listing 2). Te wartości uzyskaliśmy podczas jego rejestracji. Alternatywnie dane uwierzytelniające można również podać jako argumenty atrybutu *BotAuthorization*.

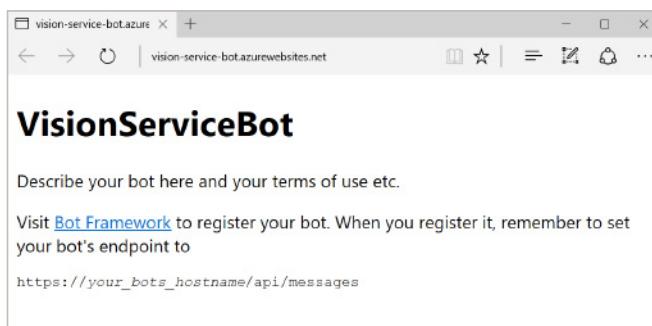
**Listing 2. Fragment pliku web.config do konfiguracji autoryzacji**

```
<appSettings>
  <!-- update these with your BotId, Microsoft App Id and your
  Microsoft App Password-->
  <add key="BotId" value="vision-service-bot" />
  <add key="MicrosoftAppId"
    value="8af757e7-4852-4460-b6dc-6f310a2a87d5" />
  <add key="MicrosoftAppPassword"
    value="<TYPE_YOUR_PASSWORD_HERE>" />
</appSettings>
```

Po zapisaniu zmian można opublikować aplikację w Azure. Przebiega to niemal automatycznie za pomocą kreatora *Publish*. Aktywuje się go za pomocą opcji *Publish...* z menu kontekstowego projektu *VisionServiceBot*. Następnie w kreatorze *Publish* wystarczy wybrać opcję *Microsoft Azure App Service*, a następnie wybrać odpowiednią aplikację. W tym przypadku jest nią *vision-service-bot*. Po kliknięciu przycisku *OK* pojawi się kolejny ekran, w którym można kliknąć przycisk *Publish*. Po chwili aplikacja zostanie opublikowana. W przeglądarce zostanie wyświetlony statyczny widok z Rysunku 13.



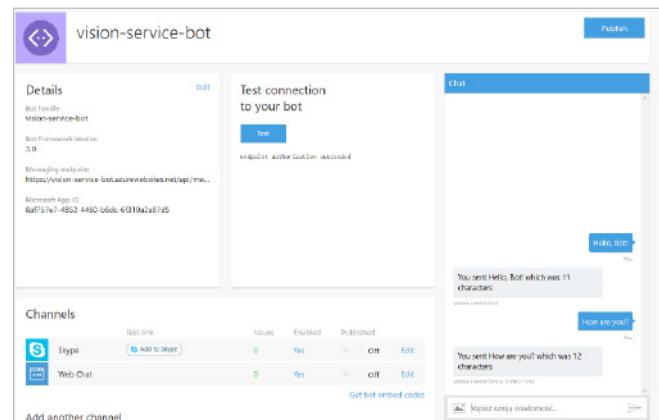
Rysunek 12. Publikacja aplikacji w Azure



Rysunek 13. Domyślny widok aplikacji vision-service-bot

Aby przetestować działanie bota, można skorzystać z emulatorka (<https://github.com/Microsoft/BotFramework-Emulator#downlo>

ad) lub z ekranu odpowiednich narzędzi dostępnych w serwisie Microsoft Bot Framework (Rysunek 10). Aby przetestować połączenie z botem, wystarczy tam kliknąć przycisk z etykietą *Test*. Natomiast wiadomości można wysyłać za pomocą okna czatu. Na Rysunku 14 zilustrowano uzyskane w ten sposób przykładowe odpowiedzi domyślnego bota.



Rysunek 14. Testowanie bota

## SZTUCZNA INTELIGENCJA

Aby rozszerzyć domyślną funkcjonalność utworzonego bota, zaimplementowałem dodatkową klasę *VisionServiceHelper* (Listing 3). Wykorzystuje ona funkcjonalność analizy obrazów dostarczaną w ramach Microsoft Cognitive Services (MCS). MCS opisywałem już na łamach „Programisty” (3/2016) i podobnie jak w tamtym artykule – również i tutaj skorzystam z klasy *VisionServiceClient*, która jest dostarczana w ramach pakietu NuGet: *Microsoft.ProjectOxford.Vision*.

Aby korzystać z klasy *VisionServiceClient*, wystarczy zarejestrować konto w serwisie MCS, a następnie przekazać uzyskany w ten sposób klucz do konstruktora klasy *VisionServiceClient*. Po utworzeniu instancji klasy *VisionServiceClient* analizuję obraz w oparciu o metodę *DescribeAsync*. Posiada ona dwie wersje. Pierwsza, z której skorzystałem w tym przykładzie, umożliwia nam przekazanie URL obrazu. Z kolei druga wersja przyjmuje w argumencie tablicę bajtów, reprezentującą obraz. Niezależnie od wybranego sposobu przekazania obrazu metoda *DescribeAsync* zwraca instancję obiektu typu *AnalysisResult*, przechowującego charakterystykę obrazu. *AnalysisResult* jest również wynikiem działania innych metod udostępnianych przez klasę *VisionServiceClient*. Z tego powodu konkretna zawartość pól klasy *AnalysisResult* zależy od użytej metody. W przypadku *DescribeAsync* opis obrazu można odczytać z właściwości *Captions* klasy *Description*. Na potrzeby tego przykładu odczytuję pierwszą etykietę opisującą zawartość wskazanego obrazu. Jest ona następnie zwrotnie zwracana jako wynik działania metody *GetDescription* klasy *VisionServiceHelper*.

W kolejnym kroku zmodyfikowałem definicję metody *Post* klasy *MessagesController* zgodnie z Listingiem 4. Najpierw sprawdzam, czy wiadomość posiada jakieś załączniki. W tym celu odczytuję właściwość *Attachments* instancji klasy *Activity*. Z kolekcji dostępnych załączników wybieram tylko ten pierwszy, a następnie analizuję go z wykorzystaniem metod sztucznej inteligencji z MCS.

Po zapisaniu tych zmian ponownie opublikowałem bota, a następnie wysłałem do niego kilka obrazów. Uzyskane wyniki przedstawiłem na Rysunku 15.

**Listing 3. Definicja klasy VisionServiceHelper**

```
public static class VisionServiceHelper
{
    private static VisionServiceClient visionServiceClient =
        new VisionServiceClient("<YOUR_KEY_Goes_HERE>");

    public async static Task<string> GetDescription(string url)
    {
        var result = "No features were detected";

        try
        {
            var analysisResult = await visionServiceClient.
                DescribeAsync(url);

            result = analysisResult.Description.Captions.
                FirstOrDefault().Text;
        }
        catch(Exception ex)
        {
            result = $"{result} ({ex.Message})";
        }

        return result;
    }
}
```

**Listing 4. Zmodyfikowana obsługa zapytań typu POST bota**

```
public async Task<HttpResponseMessage> Post([FromBody]Activity
    activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        ConnectorClient connector = new ConnectorClient(
            new Uri(activity.ServiceUrl));

        var replyString = string.Empty;

        // Check if user send some attachments
        if (activity.Attachments != null)
        {
            // If so, try to process the first one
            replyString = await ProcessAttachment(activity.Attachments.
                FirstOrDefault());
        }
        else
        {
            // Default processing...
            //

            // calculate something for us to return
            var length = (activity.Text ?? string.Empty).Length;

            // return our reply to the user
            replyString = $"You sent {activity.Text} which was
                {length} characters";
        }

        var reply = activity.CreateReply(replyString);

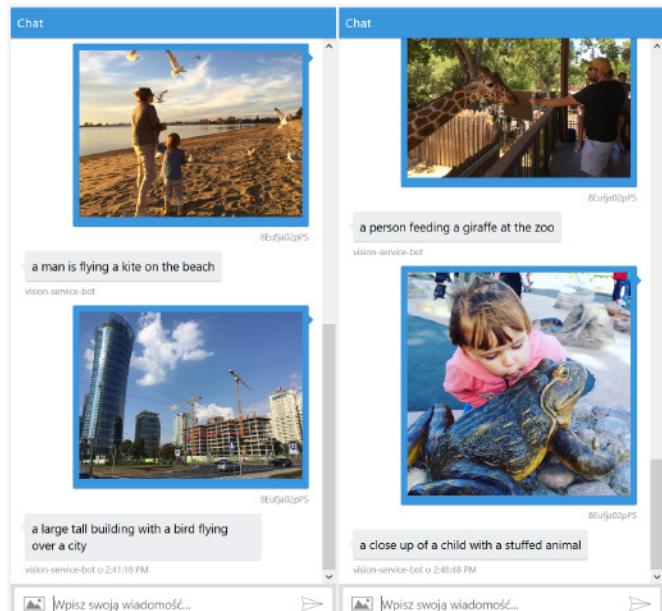
        await connector.Conversations.ReplyToActivityAsync(reply);
    }
    else
    {
        HandleSystemMessage(activity);
    }

    var response = Request.CreateResponse(HttpStatusCode.OK);

    return response;
}
private async Task<string> ProcessAttachment(Attachment attachment)
{
    var replyString = string.Empty;
```

```
if(attachment != null)
{
    // Check the content type
    if (attachment.ContentType.Contains("image"))
    {
        // Get image description
        replyString = await VisionServiceHelper.
            GetDescription(attachment.ContentUrl);
    }
    else
    {
        replyString = "Attachment format is not supported at this time";
    }
}

return replyString;
}
```



Rysunek 15. Przykładowe wyniki działania bota vision-service-bot

## KANAŁY KOMUNIKACYJNE

Zaimplementowany bot może być następnie skonfigurowany do współpracy z różnymi aplikacjami, takimi jak Skype, Facebook Messenger czy Slack. Ja zdecydowałem się skonfigurować mojego bota do pracy z Facebook Messenger. W tym celu wykonałem procedurę opisaną w dokumentacji Microsoft Bot Framework (<https://facebook.botframework.com/Dev>). Po jej ukończeniu uzyskałem efekt pokazany wcześniej na Rysunku 3.

## PODSUMOWANIE

Celem tego artykułu było przedstawienie wybranych aspektów tworzenia nowoczesnych kanałów komunikacyjnych w oparciu o Microsoft Bot Framework. Zasadnicza funkcjonalność takich botów zawiera się w klasie kontrolera Web API. Z tego powodu Microsoft Bot Framework może być bardzo szybko zastosowany w istniejących już aplikacjach ASP.NET MVC lub ASP.NET Core.

**DAWID BORYCKI**

Doktor fizyki. Adiunkt w Instytucie Chemii Fizycznej Polskiej Akademii Nauk. Zajmuje się projektowaniem oraz implementacją algorytmów cyfrowej analizy obrazów i sterowania prototypowymi urządzeniami do obrazowania biomedycznego. Współautor wielu książek o programowaniu i międzynarodowych zgłoszeń patentowych.

# Jak rozproszyć swoją web aplikację?

W tym artykule pokażę, jak wykorzystać dostępne dla Pythona narzędzia, aby maksymalnie rozproszyć aplikację Django na wiele mniejszych i tańszych maszyn, korzystając na przykład z budżetowych serwerów dedykowanych lub kolkowanych serwerów "refurbished". Równie dobrze będziemy mogli wykorzystać wirtualne maszyny od sprawdzonego usługodawcy.

## BAZA DANYCH

Prawie każda aplikacja potrzebuje jakiegoś magazynu danych. Zwykle podczas tworzenia aplikacji webowych pierwsze, co przychodzi nam do głowy jako baza, to MySQL. Posiada on wiele narzędzi pozwalających na wykorzystanie go w wygodny sposób, zwłaszcza w Pythonie. Jednak skalowanie go i tworzenie klastra może być nie lada problemem dla początkujących w tej dziedzinie administratorów. Co więcej – jedna, centralna baza danych prawie zawsze będzie słabym ogniwem naszej infrastruktury. Jeśli nawet nie pod względem wydajności, to pod względem redundancji i dostępu do niej. Mamy kilka możliwości, aby poradzić sobie z tym problemem.

### Redis i bazy NoSQL

Pierwszym podejściem jest przejście na bazy typu NoSQL (np. Redis), które korzystają z zupełnie innego podejścia do przechowywania i odczytywania danych. Wspomniany Redis pozwala zapisywać pary klucz-wartość przy jednoczesnym łatwym skalowaniu. Nie udostępnia żadnych mechanizmów znanych z baz SQL, takich jak tabele, JOIN'y i całej masy innych, potencjalnie wygodnych udogodnień. Na początku może to wydawać się dość dużym utrapieniem – nie mamy żadnych struktur danych i przy złym podejściu może to doprowadzić do dużego bałaganu. Z drugiej strony, jeśli zapewnmy sobie choć trochę dyscypliny podczas pisania kodu, możemy z tego stworzyć całkiem elastyczne narzędzie. Jak zatem poradzić sobie z brakiem tabel? Ano tak – bazy Key-Value potrafią bardzo szybko odnajdywać wartości poszczególnych kluczy. Zamiast definiować tabele, możemy po prostu stworzyć całą serię kluczy z odpowiednimi nazwami, w których będą przechowywane wartości. Na przykład obiekty klasy User, która ma pola `first_name`, `last_name`, `date_of_birth`, możemy zapisać tak:

User:1:first_name	Jan
User:1:last_name	Nowak
User:1:date_of_birth	1979
User:2:first_name	Karol
User:2:last_name	Kowalski
User:2:date_of_birth	2001
...	

Tabela 1. Przykładowe wartości kluczy

Nie widać tutaj konkretnej struktury tabel, jednak przeglądając wszystkie klucze zaczynające się od `User:2:`, możemy przeczytać wszystkie pola danego obiektu. Dodatkowo Redis pozwala w bardzo przyjemny sposób przeszukiwać swój zbiór danych przez

dodanie znaków takich jak \* i ? w celu pobrania zakresu kluczy. Aby użyć redisa będziemy potrzebowali sam serwer oraz moduł Pythona:

```
sudo apt-get install redis-server
sudo pip install redis
```

Spróbujmy wykonać powyższy kod w Pythonie:

#### Listing 1. Przykład użycia Redis w Pythonie

```
import redis
conn = redis.Redis('localhost')
conn.set('User:1:first_name', 'Karol')
conn.set('User:1:first_name', 'Kowalski')
conn.set('User:2:first_name', 'Jan')
conn.set('User:2:first_name', 'Nowak')
```

Następnie wypiszmy imię użytkownika o ID=2:

```
print(conn.get('User:2:first_name'))
```

I wypiszmy imiona wszystkich użytkowników, korzystając z metody keys z odpowiednim filtrem:

```
for key in conn.keys('User:*:first_name'):
    print(conn.get(key))
```

No dobrze – mamy pisać własny ORM<sup>1</sup> do zarządzania danymi w Redisie? Nie! Wiele frameworków pozwala użyć Redis'a właśnie w ten sposób. Dzięki temu nie musimy się martwić o to, jak zarządzać danymi. Z drugiej strony musimy się poważnie zastanowić, czy takie rozwiązanie będzie dla nas wydajne. Wszystko oczywiście zależy od tego, co w danej chwili potrzebujemy. Inaczej będzie wyglądało obciążenie bazy dla dużego serwisu, który jedynie pokazuje produkty, które są mało powiązanymi ze sobą danymi, a inaczej dla aplikacji analizującej zachowania użytkowników w takim sklepie, gdzie znajduje się dużo powiązań pomiędzy tabelami.

Warto popatrzeć tutaj na inną zaletę takiego rozwiązania – jeżeli nasz kod będzie odpowiednio przygotowany, to migracja struktury danych nie będzie dużym problemem, przez który trzeba wyłączyć pół serwisu. Co, jeśli w Redisie przy niektórych osobach pojawi się pole `User:ID:credit_card_number`? MySQL musiałby znać domyślne wartości dla każdego wpisu i wymagałby migracji tabel. Niektóre ORM'y po czymś takim nie uruchomiliby naszej aplikacji, ponieważ klasa odwzorowująca tabelę w bazie różniłaby się pólami. W przypadku umiejętnego wykorzystania NoSQL nie mamy tego problemu. Jedyne, co musimy zrobić, to mieć to na uwadze

1. Object-relational mapping – mechanizm pozwalający na odwzorowanie struktury tabel w bazie danych na klasy w naszej aplikacji. Wykorzystanie tego typu narzędzi znacznie przyspiesza pisanie kodu i odpisywanie bazy danych, zwiększać przy tym bezpieczeństwo, np. poprzez eliminowanie punktów potencjalnie podatnych na SQL injection.

przy pobieraniu danych z bazy i liczyć się z brakiem poszczególnych pól (np. przez ustawienie wartości domyślnej w kodzie).

Popatrzmy na przykładową, bardzo prostą implementację takiego prostego „ORM”:

#### **Listing 2. Wykorzystanie bazy NoSQL (np. Redis) do zapisywania obiektów w Pythonie**

```
import redis
import uuid

class Model(object):
    _conn = None
    _model_name = 'not_defined'
    _id = None

    def __init__(self, id):
        # Tym konstruktorem „pobieramy” z bazy istniejący
        # obiekt. Tak naprawdę jakiekolwiek dane zostaną
        # pobrane dopiero przy wywołaniu metody get
        self._id = id
        self._conn = redis.Redis('localhost')

    def __init__(self):
        # Tym konstruktorem tworzymy nowy obiekt. Wygeneruje on
        # losowy ID dla obiektu
        self._id = str(uuid.uuid4())
        self._conn = redis.Redis('localhost')

    def get(self, field, default):
        # Jak w powyższym opisie - każde pole składa się
        # z nazwy modelu, ID obiektu i nazwy pola, oddzielonych
        # dwukropkami
        value = self._conn.get(self._model_name + ':' + self._id + ':'
                               + field)

        # Jesli Redis nie zwrocił zadnej wartości (pole nie ma),
        # to zwracamy domyslna, podana przez programista
        if value == None:
            return default
        else:
            return value

    def set(self, field, value):
        self._conn.set(self._model_name + ':' + self._id + ':' + field, value)

    @classmethod
    def object_get(cls, id):
        return cls(id)
```

Jest to klasa pozwalająca na ustawienie i odczyt dowolnego pola z modelem danych (metody `set` i `get`) oraz pobranie obiektu z bazy danych. Aby utworzyć nowy model danych, musimy jeszcze stworzyć klasę odpowiadającą temu modelowi:

```
class User(Model):
    _model_name = 'User'
```

i gotowe. Możemy dalej dowolnie zarządzać danymi w obrębie takich obiektów. Wszystkie zmiany będą zapisane w Redisie, do którego się połączylismy w konstruktorze klasy `Model`:

#### **Listing 3. Wykorzystanie prostego ORM do zapisania informacji o użytkownikach**

```
u1 = User()
u1.set('first_name', 'Jan')
u1.set('last_name', 'Kowalski')

u2 = User()
u2.set('first_name', 'Jan')
u2.set('last_name', 'Kowalski')

u1.get('first_name', 'not known')
u1.get('middle_name', 'not known')
```

Pierwszy `get` powinien zwrócić poprawne imię Jan. Drugi `get` powinien zwrócić string `not known`, ponieważ nie ustawiliśmy takiego

pola. Powyższy przykład jest bardzo prosty. Można by go oczywiście rozbudować o kolejne metody, odpowiednio przeładować wbudowaną metodę `__getattribute__` i dopisać wyszukiwanie obiektów w bazie. Pozostaje wtedy tylko pytanie, kiedy warto sięgnąć po gotowe rozwiązanie, posiadające te wszystkie cechy.

Niezależnie od naszego celu powyższy przykład jest bardzo odporny na migrowanie danych. Jeśli nasza aplikacja jest rozproszona po wielu komputerach i jej zaktualizowana część aplikacji zacznie używać pola o nazwie `middle_name`, to nic nie powinno się dziać ze starą wersją. Nie musimy martwić się też o migrację

#### **Rozproszone samej bazy danych – Cassandra**

Innym podejściem jest rozproszenie bazy danych na wiele serwerów i odpowiednie przygotowanie samej struktury danych. O ile MySQL może nie być najlepszym przykładem takiego rozwiązania, to baza danych Cassandra już tak. Jest to oprogramowanie stworzone przez Facebook'a, które zostało od samego początku zaprojektowane jako bardzo skalowalne, tak aby podobać wyzwaniom, jakie stawia uruchomienie tak dużego serwisu jak Facebook.

#### **Każdemu po troszkę – osobna baza dla każdej części serwisu**

Innym sposobem na rozproszenie i zdecentralizowanie bazy jest określenie, jakie dane są potrzebne poszczególnym fragmentom naszej aplikacji. Na przykład aplikacja obsługująca sklep internetowy nie potrzebuje dostępu do systemu newsów, a ten nie potrzebuje dostępu do oprogramowania analizującego ruch na naszych stronach. Idąc tą ścieżką, możemy mocno rozdzielić poszczególne funkcjonalności systemu oraz samą bazę.

Co jednak, gdy nie da się tego do końca rozdzielić? Możemy pójść w kierunku architektury mikroserwisów, do których ten artykuł jest wstępem. Na przykład, jeśli w każdym zakątku naszego systemu będziemy potrzebowali informacji o kontach użytkowników, to zapewnijmy sobie dostęp do tych informacji przez dodatkowe API dedykowanego serwisu. Z jednej strony dodajemy nową warstwę pośredniczącą w dostępne do bazy, co może pogorszyć wydajność, lecz z drugiej strony ta warstwa może pełnić rolę `cache`, które znacznie ograniczy odczyty z bazy danych.

Zwykle informacja o tym, czy konto istnieje i czy użytkownik jest poprawnie zautoryzowany przez jego klucz sesji, jest potrzebna natychmiast. Jeśli nasz serwis obsługujący konta użytkowników pozwoli się w łatwy sposób zduplikować i zapewnić `cache`, to znacząco odciążymy dzięki temu bazę. Z drugiej strony będziemy mieli pewne opóźnienia w aktualizacji takiej bazy, właśnie przez `cache`. Warto jednak rozważyć ten koszt, gdyż tworzenie nowego konta lub zmiana hasła są dość rzadkimi operacjami, a użytkownikowi można pokazać jedynie komunikat „Twoje konto zostanie aktywowane w ciągu X minut”.

## **PROSTA APLIKACJA W DJANGO**

Aby pokazać narzędzia, dzięki którym możemy uruchomić aplikację na więcej niż jednym komputerze, będziemy potrzebowali jakieś przykładowej aplikacji. W poprzednim wydaniu „Programisty” pokazaliśmy, jak zbudować prosty serwis oparty o Django. W bardzo dużym skrócie przypomnę tutaj, jak taką aplikację stworzyć i jak dodać do niej model danych pozwalający przechowywać informacje o kontach użytkowników. Przyda się to nam w dalszej części do ob-

slugi wysyłania maili. Na początek musimy uruchomić nowy projekt Django, a w nim aplikację zarządzającą kontami, którą rozbudujemy:

```
django-admin startproject mywebsite
cd mywebsite
./manage.py startapp account
```

Aplikacja account będzie odpowiedzialna za obsługę kont użytkowników. Aby ją włączyć, musimy dodać jej nazwę w pliku mywebsite/settings.py do listy INSTALLED\_APPS:

```
INSTALLED_APPS = [
    'account',
    'django.contrib.admin',
    ...
]
```

Dodajmy routing<sup>2</sup> do poszczególnych aplikacji w pliku mywebsite/urls.py:

**Listing 4. Główny plik urls.py łączący poszczególne adresy URI z aplikacjami w Django**

```
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    # Domyslny wpis umożliwiający wykorzystanie wbudowanego panelu
    # administratora
    url(r'^admin/', admin.site.urls),

    # Dolaczony plik urls.py z aplikacji account
    url(r'^account/', include('account.urls')),
]
```

Dodajmy do tego jeszcze plik urls.py w aplikacji account, który połączy odpowiednie funkcje generujące poszczególne strony:

**Listing 5. Plik urls.py naszej aplikacji łączący poszczególne URI z konkretnymi funkcjami do ich obsługi**

```
from django.conf.urls import url
from django.contrib import admin

import account.views

urlpatterns = [
    url(r'^register/$', account.views.register, name='account_register'),
    url(r'^login/$', account.views.login, name='account_login'),
]
```

Django będzie sklejało razem wyrażenie regularne ^/account (główny plik mywebsite/urls.py) z jedną z dwóch możliwości (register/\$ lub login/\$) z dołączonego pliku (account/urls.py). Z tego powodu nie należy w dołączonym pliku zaczynać URI od ^, a jedynie zadbać o końcowy \$.

Na koniec pozostało nam jeszcze opisać model danych w pliku account/models.py, który w najprostszy sposób opisuje konto użytkownika:

**Listing 6. Model danych dla konta użytkownika – Django ORM**

```
from __future__ import unicode_literals

from django.db import models

class Account(models.Model):
    first_name = models.CharField(max_length=40)
    last_name = models.CharField(max_length=40)
    email = models.CharField(max_length=40)
    active = models.BooleanField(default=False, help_text='Is
    account activated through email')
    activation_code = models.CharField(max_length=60, default='')
```

```
blank=True, null=True, help_text='Code used to activate
account with email')

password_hash = models.CharField(max_length=60, help_
text='sha512 of password + salt')
```

oraz dodać dwie funkcje obsługujące powyższe URI w pliku account/views.py:

**Listing 7. Funkcje w pliku views.py obsługujące logowanie i rejestrację użytkownika**

```
from django.shortcuts import render, redirect, HttpResponseRedirect
from account.models import Account

import random
import hashlib

def register(request):
    if request.method == POST:
        # Wypada w tym miejscu dodac walidacje danych
        new_user = Account()
        new_user.first_name = request.POST['first_name']
        new_user.last_name = request.POST['last_name']
        new_user.email = request.POST['email']
        new_user.password_hash = str(hashlib.sha512(request.
POST['password']).hexdigest())
        new_user.active = False
        new_user.activation_code = str(hashlib.sha1(
            str(random.random())).hexdigest())

        # Nastepnie zwrocmy przekierowanie do strony z informacją o mailu
        return redirect('account_created')
    else:
        # Tutaj powinnismy wygenerowac formularz,
        # np. korzystajac z szablonow Django
        return render('registration_form.html')

def login(request):
    return HttpResponseRedirect('Not implemented')
```

Czego tutaj brakuje? Poza oczywistą oczywistością, jaką są formularze i cała otoczka wizualna strony zapisana w szablonach (ang. *templates*), o których można poczytać na stronach projektu Django<sup>3</sup>, potrzebujemy trochę rozbudować logikę związaną z rejestracją.

W większości serwisów aktywacja konta odbywa się przez wysłanie wiadomości email z linkiem aktywującym. Pierwszą z potrzebnych rzeczy będzie dołożenie kolejnej funkcji do account/views.py, która będzie wywoływana takim linkiem i będzie ustawiała poprawnie flagę active w danym obiekcie konta:

**Listing 8. Funkcja odpowiedzialna za aktywowanie konta poprzez link wysłany emailem**

```
def activate(request, code):
    account = Account.objects.get(activation_code=code)
    account.active = True
    account.save()
    return HttpResponseRedirect('home')
```

Pierwsza instrukcja w tej funkcji pobiera z bazy obiekt Account, który ma przypisany odpowiedni klucz aktywacyjny, podany w parametrze. Następnie możemy zmodyfikować taki obiekt i zapisać zmiany w bazie. Funkcja HttpResponseRedirect służy do przeniesienia użytkownika na inną stronę zamiast zwracania gotowej odpowiedzi ze stroną. Jako parametr podajemy nazwę URI zdefiniowanej w jednym z plików urls.py. Istnieje też możliwość podania dodatkowych parametrów, których potrzebuje docelowa funkcja. Wykorzystanie tego mechanizmu ma tę zaletę, że zawsze będziemy mogli odwołać się do niej przez nazwę (niewidoczną dla klienta). Jest to wtedy niezależne od zmiany nazwy aplikacji lub URL, który wskazuje na daną funkcję.

2. Routing w aplikacjach Django pozwala połączyć poszczególne adresy stron (URI) z odpowiednimi funkcjami generującymi te strony (views).

3 <https://docs.djangoproject.com/pl/1.10/intro/tutorial03/>.

Dodatkowo musimy powiązać stworzoną właśnie funkcję `activate` z odpowiednim URI. Dodajmy nowy wpis w pliku `account/urls.py`:

```
url(r'^activate/(?P<code>[a-zA-Z0-9]+)$', account.views.login,
    name='account_activate'),
```

Pozostaje jeszcze tylko wysłać maila i gotowe!

## CELERY

Zazwyczaj programiści pisząc aplikacje, mają tendencje do pisania kodu w miejscu, w którym jest on potrzebny. Nieco bardziej doświadczeni wyrzucą duże fragmenty kodu do osobnych funkcji. Jeszcze lepsi zrobią to tak, aby dany fragment był reużywalny i da się dobrze przetestować. Podobnie jest w naszej funkcji rejestrującej użytkownika. Kod wysyłający maila można wrzucić bezpośrednio do funkcji `register`. Co jednak, gdy samo wysłanie maila będzie trwało kilkanaście sekund ze względu na wolno działający serwer SMTP? Pomijając powyższe kwestie dobrych praktyk pisania kodu, takie podejście prowadzi do znacznego wydłużenia odpowiedzi użytkownikowi. Podobne sytuacje mogą się przydarzyć w różnych innych miejscach serwisu, np. przy przetwarzaniu otrzymanych w załączniku obrazków, generowaniu miniaturek lub podczas łączenia się z zewnętrznymi usługami. Zwykle chcemy, aby nasz serwis działał jak najszybciej. Jeśli użytkownik nie dostaje odpowiedzi od razu lub widzi, że coś długo się ładuje, to odświeża stronę aż do skutku, aby w końcu pójść do konkurencji. Według niektórych badań „długego” w tym przypadku może oznaczać już nawet kilka sekund.

Wysłanie wiadomości z linkiem aktywacyjnym może być naszym pierwszym punktem, w którym użytkownik będzie widział, że nasz serwis działa ociążale. W takim przypadku dobrą praktyką jest odesłanie użytkownikowi jakiekolwiek odpowiedzi. Informujemy go, że coś robimy i wkrótce będzie to gotowe. Na przykład zamiast pokazywać komunikat „Link aktywacyjny został wysłany – sprawdź pocztę” możemy zasugerować: „Wkrótce otrzymasz maila z linkiem aktywującym Twoje konto”. W tym czasie nasza aplikacja może zapisać informację o konieczności wykonania zadania w kolejce i szybko zwrócić tę odpowiedź. W międzyczasie inna maszyna będzie mogła pobrać takie zadanie i je zrealizować asynchronicznie.

Dokładnie taki scenariusz możemy zrealizować dzięki bibliotece Celery dla Pythona. Pozwala ona w bardzo transparentny sposób wywołać funkcje w tle, które mogą się wykonać w późniejszym czasie. Dzięki temu możemy przede wszystkim odciążyć nasz jeden serwer – przez wyrzucenie pojedynczych funkcji na inne maszyny. Po drugie, możemy znacznie zwiększyć bezpieczeństwo całej aplikacji. Włamanie się na maszynę obsługującą samą stronę nie da intruzowi dostępu do innych części serwisu, które działają za pośrednictwem Celery.

## Dodanie zadań

Aby wykorzystać zalety tego narzędzia, przede wszystkim będziemy potrzebowali samej biblioteki celery oraz django-celery:

```
pip install celery
pip install django-celery
```

W głównym pliku konfiguracyjnym Django, w `mywebsite/settings.py` dodajmy kilka linijek określających, jak Celery ma połączyć się z brokerem wiadomości. Do tego użyjemy wcześniej opisanego Redis'a:

### Listing 9. Dodatkowe ustawienia Django dla wsparcia Celery

```
BROKER_URL = 'redis://localhost:6379'
CELERY_RESULT_BACKEND = 'redis://localhost:6379'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = 'Europe/Paris'
```

Zgodnie z dokumentacją Celery ze strony <http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html> potrzebny nam jest jeszcze plik `mywebsite/celery.py`, który pozwala na skonfigurowanie całego środowiska Celery (można przekopiować z powyższego linka):

### Listing 10. Definicja aplikacji Celery w Django

```
from __future__ import absolute_import, unicode_literals
import os
from celery import Celery

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'proj.settings')

app = Celery('tasks')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()

@app.task(bind=True)
def debug_task(self):
    print('Request: %01r' % self.request)
```

To wystarczy, aby Celery mogło być uruchomione i wykorzystane razem z Django. Na koniec w naszej aplikacji `accounts` dodajmy plik `account/tasks.py`, który będzie zawierać wszystko to, co Celery ma uruchamiać w tle:

### Listing 11. Zadanie dla Celery odpowiedzialne za wysłanie emaile aktywującego konto

```
from account.models import Account
from mywebsite.celery import app

from django.core.mail import send_mail

@app.task
def send_registration_email(account_id):
    account = Account.objects.get(pk=account_id)
    send_mail('Activate your account',
              'Email content with link...', 
              'no-reply@mywebsite',
              [account.email],
              fail_silently=False)
```

Wygląda jak zwykła funkcja – czyż nie? Jedyną różnicą jest dekorator `@app.task`, który rejestruje tę funkcję w Celery. Należy też pamiętać, aby parametrami funkcji rejestrowanych w Celery były tylko stringi i liczby. Aby wywołać ją bezpośrednio z kodu naszej aplikacji i uruchomić kod tej funkcji lokalnie (czyli tradycyjną drogą, tak jak to zwykle robimy), wpisujemy:

```
from account.tasks import send_registration_email
...
send_registration_email(account.id)
```

Aby uruchomić powyższą funkcję za pośrednictwem Celery (czyli na dowolnym komputerze, na którym uruchomiony jest worker), wywołujemy następujące polecenie:

```
send_registration_email.delay(account.id)
```

Różnica polega jedynie na dopisaniu `.delay` do wywołania funkcji. Dzięki temu Celery zapisze w kolejce informację o konieczności

wywołania funkcji `send_registration_email` i przejdzie do dalszego wykonywania kodu.

Jak zapewne zauważycie, nie mamy tu możliwości zwrócenia żadnej wartości do miejsca, w którym wywołujemy funkcję. Jest to możliwe, jednak wymaga dodatkowego skonfigurowania Celery. Tracimy też wtedy zaletę, jaką jest asynchroniczne wywołanie naszego kodu, jednak dalej możemy znacznie odciążyć nasz główny serwer. Zadaniem, które byłoby dobrym przykładem, może być wygenerowanie miniaturki zdjęcia – operacja dość pracochłonna. Po przerzuceniu tego zadania na inną maszynę odciążamy nieco nasz serwer.

## Uruchomienie głównej aplikacji

Główną aplikację Django możemy uruchomić przez dołączony skrypt `manage.py`. Wcześniej należy jednak stworzyć migrację bazy danych (zwykle należy to robić po jakiekolwiek aktualizacji modelu danych) i zmigrować samą bazę:

```
./manage.py makemigrations  
./manage.py migrate
```

Finalnie możemy uruchomić wbudowany w Django serwer WWW:

```
./manage.py runserver
```

Nasza strona będzie dostępna pod adresem <http://localhost:8000/>.

## Uruchomienie workera

Całość będzie miała sens tylko wtedy, gdy zadania z kolejki Celery będą wykonywane. Na dowolnej innej maszynie musimy zatem zdeployować całą naszą aplikację. Jest to punkt, w którym rozwijanie aplikacji i jej produkcyjne uruchomienie mocno się od siebie różnią. Zwykle programiści tworzą i uruchamiają wszystko na jednym komputerze. W środowisku produkcyjnym administrator powinien mieć możliwość wykorzystania w jak najłatwiejszy sposób dostarczonych narzędzi.

Idealnym rozwiązaniem będzie wrzucenie całego naszego projektu do repozytorium Git (lub innego) i zautomatyzowanie całego deploymentu. Po pierwsze, możemy utworzyć kilka wersji pliku `mywebsite/settings.py` zawierających dane do połączenia z odpowiednim serwerem Redis oraz bazą danych. Django wykorzystuje zmienną środowiskową `DJANGO_SETTINGS_MODULE` do określenia, z jakiego pliku ma skorzystać. Drugą kwestią jest zawarcie w skrypcie przygotowującym środowisko całego procesu instalacji zależności:

### Listing 12. Uruchomienie workera Celery

```
sudo apt-get --yes install git python python-pip  
sudo pip install -U celery  
sudo pip install -U django  
sudo pip install -U django-celery  
git clone https://repository_url/mywebsite  
cd mywebsite  
export DJANGO_SETTINGS_MODULE="mywebsite.settings_production"  
celery -A mywebsite worker -l info
```

Ostatnia linijka powyższego kodu uruchamia worker Celery, który będzie wykonywał zapisane w kolejce zadania. Przed uruchomieniem należy sprawdzić, czy nie korzystamy z serwera Redis (zmienna `CELERY_RESULT_BACKEND` w pliku `mywebsite/settings.py`) ustawionego na localhost. Zamiast tego powinniśmy tak skonfigurować Redisa, aby nasłuchiwał na jednym z interfejsów sieciowych w naszej sieci lokalnej.

## AUTOMATYCZNY DEPLOYMENT

Przygotowanie aplikacji, która da się łatwo rozproszyć na wiele komputerów, to już duży sukces. Pozostaje jednak problem, jak i gdzie ją zainstalować. Jeśli zdecydujemy się na ręczną instalację na fizycznych serwerach, to prawdopodobnie za jakiś czas okaże się, że wszystkie osoby zaangażowane w projekt nie pamiętają, jak ją zainstalować. Dodatkowo, gdy dochodzimy do szczytu wydajności sprzętu, na którym to działa, pojawia się problem z czasem dołożenia kolejnego komputera. Poza samym deploymentem aplikacji musimy przecież zainstalować system, podłączyć i skonfigurować sieć, firewall oraz przypomnieć sobie, gdzie są poszczególne komponenty (baza danych, cache itd.).

Dobrym pomysłem może być wykorzystanie środowiska chmury, które pozwoli w kilka chwil uruchomić kolejne maszyny z workerem Celery lub zduplikować samą główną aplikację. Zintegrowanie się z takim środowiskiem pozwoli nam też zautomatyzować testy instalacji i skalowania serwisu, które zwykle nie należą do zbyt przyjemnych.

Jakiś czas temu opisywaliśmy na łamach „Programisty” skrypty CloudInit wspierane przez większość chmur, m.in przez Amazon EC2 oraz CoreCluster. Stwórzmy prosty skrypt, który pomoże przygotować nasz system:

### Listing 13. Przykładowy skrypt CloudConfig pozwalający na automatyczną instalację i uruchomienie workera na nowej instancji w chmurze, korzystając z systemu CloudInit

```
#cloud-config  
bootcmd:  
  - echo "ssh-rsa ..." >> /root/.ssh/authorized_keys  
  - apt-get --yes install git python python-pip  
  - pip install -U celery django django-celery  
  - git clone https://repository_url/mywebsite  
  - cd mywebsite  
  - export DJANGO_SETTINGS_MODULE="mywebsite.settings_production"  
  - celery -A mywebsite worker -l info &
```

Jeśli odpowiednio przygotowaliśmy plik ustawień dla produkcyjnej wersji (`DEBUG=False`, adres brokera wiadomości dla Celery oraz połaczenie z bazą danych), to bez problemu worker się uruchomi na tak postawionej wirtualnej maszynie. Żeby jeszcze bardziej uprościć cały proces, możemy pójść dalej i przygotować skrypt tworzący taką wirtualną maszynę w chmurze i wpinający ją do odpowiedniej sieci. Zainstalujmy bibliotekę pycore potrzebną do połączenia się z chmurą CoreCluster:

```
sudo pip install pycore
```

Następnie możemy zacząć pisać skrypt, który połączy się z API chmury i pozwoli na tworzenie zasobów:

### Listing 14. Stworzenie połączenia z API chmury CoreCluster

```
import pycore  
cloud = pycore.Cloud('http://mycloud_api:8000', 'my_login',  
'my_password')  
api = cloud.get_api()
```

Jeśli nie chcemy wpisywać loginu i hasła na sztywno do skryptu, możemy utworzyć obiekt API bezpośrednio, podając token do naszego konta:

### Listing 15. Stworzenie połączenia z API za pomocą tokena

```
import pycore  
api = pycore.Api('http://mycloud_api:8000', '94q0iwgy7slzfhuq34fwf....')
```

Następnie, za pomocą obiektu api, możemy sprawdzić nasze dostępne zasoby w chmurze i wykorzystać je do stworzenia nowej wirtualnej maszyny:

#### Listing 16. Przygotowanie zasobów chmury niezbędnych do utworzenia wirtualnej maszyny z naszą aplikacją Django

```
# Pobierzmy szablon sprzętu o nazwie Small
template = api.template_by_name('Small')[0]

# Sprawdzmy, czy taka sieć istnieje, jeśli nie, to stworzymy nową
if len(api.network_by_name('MyWebsite')) == 0:
    network = api.network_create(mask=26,
                                  name='MyWebsite',
                                  isolated=False,
                                  mode='routed')

# Zaalokuj wszystkie adresy wewnątrz tej sieci
network.allocate()
network = api.network_by_name('MyWebsite')[0]

# Zaktualizuj, że obraz istnieje. Jeśli nie, to trzeba
# analogicznie stworzyć nowy i wgrać jego zawartość z URL
image = api.image_by_name('Ubuntu Xenial Cloud')[0]
```

Wszystkie metody ...\_by\_name zwracają listę zasobów pasujących do nazwy. Ponieważ nazwy poszczególnych obiektów nie muszą być unikalne, bierzemy zawsze pierwszy obiekt z brzegu. Jeśli potrzebujemy mieć pewność, że pobierany przez API chmury obiekt jest tym właściwym (np. obraz dysku), możemy skorzystać z metod ...\_by\_id, na przykład api.template\_by\_id. Powyższe kilka linijek wyszukuje w chmurze wszystkie potrzebne nam zasoby – szablon sprzętu (dysk, procesor, pamięć), wirtualna sieć oraz obraz dysku. Z tymi informacjami jesteśmy w stanie stworzyć wirtualną maszynę.

Następnie możemy przejść do wgrania do naszego konta skryptu CloudConfig, który przygotowaliśmy przed chwilą:

#### Listing 17. Przekazanie do chmury definicji skryptu CloudConfig

```
script = open('cloudconfig.txt', 'r').read()
cloudconfig = api.coretalk.userdata_create(name='MyWebsite Worker',
                                            data=script)
```

i stworzyć nową wirtualną maszynę:

#### Listing 18. Zdefiniowanie i uruchomienie wirtualnej maszyny za pomocą biblioteki PyCore

```
# Tworzymy instancję z obrazu i szablonu sprzętu
instance = api.vm_create(name='MyWebsite Worker',
                          description='version xyz',
                          template=template,
                          base_image=image)

# Następnie podłączamy pierwszy wolny adres z naszej sieci
lease = network.lease_get_free()
lease.attach(instance)

# Rozszerzamy dysk do maksymalnego rozmiaru dostępnego w wybranym
# szablonie. Domyslnie obrazy dla chmur mają mały rozmiar - 2GB
instance.resize_image(template.hdd*1024*1024)

# Dolaczamy instrukcje CloudConfig
cloudconfig.attach(instance)

# I włączamy naszą maszynę
instance.start()
print("Instance %s was created. Address: %s" % (instance.id,
                                                lease.address))
```

### Linki

- ▶ Redis: <https://redis.io/>
- ▶ Biblioteka Celery: <http://www.celeryproject.org/>
- ▶ Django: <https://www.djangoproject.com/>
- ▶ CoreCluster: <https://cloudovery.org/>
- ▶ Dokumentacja biblioteki PyCore: <https://goo.gl/wlwhq>

Tak przygotowany skrypt może być dobrym narzędziem dla administratora – nie musi on przy każdym uruchomieniu szukać dokumentacji lub angażować programistów, którzy akurat coś zmienili. Każdorazowe uruchomienie nowego workera dla Celery działa tak samo, a wszelkie zmiany można w bardzo prosty sposób przetestować.

Mając takie narzędzie, pozostało tylko zaprzegnąć do pracy narzędzie typu Jenkins i przygotować testy instalacji i integracji, które będą uruchamiały się codziennie, bez angażowania kogokolwiek. Analogicznie możemy oskryptować tworzenie każdego komponentu naszej aplikacji, począwszy od głównego skryptu Django, aż po samą bazę danych. Modyfikując nieco zawartość pliku CloudConfig, możemy też zautomatyzować samą konfigurację Django, tak aby była automatycznie dopasowywana do zasobów w chmurze. Można to zrobić przez wyciąganie odpowiednich adresów IP poszczególnych komponentów i dopisywanie ich do pliku CloudConfig, który odpowiednio zapisze je w plikach konfiguracyjnych.

#### Listing 19. Przykład wylistowania wszystkich maszyn z danej sieci

```
# Pobierzmy liste adresów z naszej sieci,
# które są przypisane do instancji
attached_leases = [lease for lease in network.lease_list() if
lease.vm_id is not None]

# Pobierzmy odpowiadające im wirtualne maszyny
vms_in_network = [api.vm_by_id(lease.vm_id) for lease in attached_leases]

# Z powyższego wyniku wybierzmy tylko maszyny stworzone z nazwą
# MyWebsite Worker
worker_vms = [vm for vm in vms_in_network if vm.name ==
'MyWebsite Worker']
```

Korzystając z powyższego fragmentu, możemy wyciągnąć dowolne informacje na temat zasobów w naszym koncie, powiązanych ze stworzoną siecią. Dalej pozostało tylko wpisać je w odpowiednie pliki konfiguracyjne podczas deploymentu kolejnych instancji.

Powyższe technologie to zaledwie kropla w morzu tego, co dzisiaj możemy wybrać z dostępnych narzędzi do tworzenia rozproszych aplikacji i to tylko bazując „na własnym podwórku”. Istnieje jeszcze cała gama rozwiązań, które nie wymagają od programisty niczego poza poznaniem ich API. Jest to tzw. PaaS (*Platform as a Service*), czyli cała grupa usług dostępnych online, począwszy od baz danych, cache, aż po specyficzne usługi takie jak wysyłanie maili lub usługi bazujące na Google Maps. Są to usługi działające u zewnętrznych dostawców, których nie musimy nigdzie instalować. W zamian za to płacimy w zależności od ich wykorzystania. Pokazane tu technologie pozwolą nam stworzyć aplikację w Pythonie gotową na skalowanie jej na wiele serwerów, bez obawy o to, że w którymś momencie natknieniemy się problem nie do przeskoczenia, wymagający przepisania całej aplikacji.



### MACIEJ NABOŻNY

[maciej.nabozny@cloudovery.io](mailto:maciej.nabozny@cloudovery.io)

Programista i administrator systemów rozproszych. Od ponad siedmiu lat zajmuje się projektowaniem chmur obliczeniowych IaaS, począwszy od małych instalacji, aż po największe klastry obliczeniowe w Polsce. Od 2014 roku rozwija projekt otwartej chmury obliczeniowej CloudOver.org, który ma wyjść naprzeciw potrzebom małych i średnich instalacji, dla których OpenStack to zbyt duży. Od czasu do czasu organizuje również warsztaty z różnych tematów okochochmurowych. Autora można znaleźć przez grupę KrakCloud na meetup.com.

# Transmisja danych dźwiękiem w JavaScript od podstaw

## Część 2: Web Audio API

W pierwszej części artykułu (numer 8/2016) poznaliśmy od podstaw zagadnienie Dyskretnej Transformaty Fouriera z punktu widzenia programisty. Do realizacji prostej wymiany danych dźwiękiem w JavaScript potrzebujemy jednak czegoś więcej. Następnym klockiem w układance jest Web Audio API. Postaramy się wykorzystać z niego tylko te elementy, które będą przydatne do realizacji naszego celu. Pozwoli nam to zaimplementować własną klasę `AudioMonoIO`, która opakuje wspomniane API do postaci prostego interfejsu. Na końcu stworzymy aplikację, będącą wirtualnym pianinem, za pomocą której będziemy mogli przesyłać dźwięki pomiędzy dwoma urządzeniami.

**W**eb Audio API jest bardzo rozbudowanym systemem udostępnianym przez większość nowoczesnych przeglądarek. Daje ono możliwość przetwarzania sygnałów dźwiękowych oraz kontrolę tego procesu bezpośrednio z poziomu JavaScript. Otwiera nam to drogę do aplikacji umożliwiających np. obróbkę dźwięku, filtrowanie czy tworzenie muzyki ze zbioru sampli. W przypadku gier będą to wszelkie efekty dźwiękowe. Najprostszym przykładem może być potrzeba odwzorowania akustyki sceny (echo, pogłos) czy też emitowanie dźwięku kroków z proporcjonalną głośnością w lewym lub prawym głośniku w zależności od strony, z której nadciąga wróg. Web Audio API dla wielu z tych przypadków posiada już gotowe „klocki”.

Jak widzimy, oferowane możliwości są bardzo duże. Oczywiście nie będziemy korzystać ze wszystkich dostępnych elementów. Musimy się zatem zastanowić, co tak naprawdę będzie nam potrzebne do realizacji prostej wymiany danych. Zanim jednak tego dokonamy, przeprowadźmy małą retrospekcję tego, co udało nam się zrealizować do tej pory. Pozwoli nam to wstępnie określić wymagania odnośnie Web Audio API nawet bez jego znajomości.

W poprzedniej części artykułu poznaliśmy od podstaw prosty i intuicyjny algorytm Dyskretnej Transformaty Fouriera. Do jego przetestowania stworzyliśmy sztuczny sygnał składający się z kilku przebiegów sinusoidalnych. Jako że rzeczywisty kanał transmisyjny nigdy nie jest pozbawiony zakłóceń, zdecydowaliśmy się dodać łatwy do zasymulowania biały szum. Zbliżyliśmy się zatem w małym stopniu do realnego świata. Tak przygotowany sygnał przepuściliśmy przez wspomniany algorytm DTF. Ostatecznie udało nam się z powodzeniem wyciągnąć główne częstotliwości tworzące sygnał, nawet gdy był on dość zakłócony. Oprócz częstotliwości przebiegów sinusoidalnych udało nam się także wydobyć informację o ich przesunięciu fazowym. Warto zaznaczyć, że przy tych wszystkich działaniach nie została użыта żadna gotowa biblioteka DSP. Cel, jaki wyznaczyliśmy, to poznanie tematu od podstaw bez żadnego magicznego pudełka, którego nie bylibyśmy w stanie zaimplementować samodzielnie. Wyjątkiem jest tutaj oczywiście

sam proces próbkowania i emisji dźwięku. Tę część zwyczajnie musimy pozostawić sprzętowi.

Niestety ceną prostoty użytego algorytmu DTF jest jego bardzo wolne działanie w porównaniu z alternatywą, jaką jest FFT (ang. *Fast Fourier Transform*). Również generowanie w czasie rzeczywistym tylko jednej sinusoidy przy standardowym próbkowaniu 44.1 kHz wymaga wywołania 44100 razy funkcji `Math.sin()`. Warto o tym pamiętać, ponieważ może być to szczególnie istotne, gdy jednym z urządzeń, jakie użyjemy do testów, będzie np. smartfon.

Już na pierwszy rzut oka widać, że najważniejsze, czego potrzebujemy od Web Audio API, to uzyskanie dostępu do prawdziwych próbek audio z naszego mikrofonu. Jako że rolą drugiego urządzenia będzie nadawanie sygnału, musimy mieć także możliwość otwierania strumienia próbek na głośnikach. W obydwu przypadkach z powodzeniem wystarczy tylko jeden kanał dźwięku (mono).

Niestety, jak przekonałem się podczas testów praktycznych, z uwagi na wolniejsze urządzenia nie jest możliwe skorzystanie tylko z naszych własnych rozwiązań DSP. Na szczęście w Web Audio API możemy znaleźć gotowe elementy do przetwarzania wejścia, jak i generowania wyjścia.

Osobiście nie lubię takiej formy „oszukiwania” po wcześniejszym ustaleniu założeń projektu. Niestety jest to jedyne wyjście w przypadku np. smartfonów. Algorytmy wbudowane w API przeglądarki są dużo szybsze niż ich JavaScriptowe odpowiedniki oraz często są dodatkowo akcelerowane sprzętowo. Finalnie zatem postaramy się zaimplementować dwie drogi prowadzące do tego samego celu.

Wygląda na to, że mniej więcej wiemy już, czego oczekujemy od Web Audio API. Nie pozostało nam zatem nic innego, jak przesledzić szczegółowo jego elementy i połączyć wszystko w logiczną całość. Dobre byłoby także opakować całość komunikacji z Web Audio API do postaci prostego interfejsu zawierającego tylko to, co jest nam potrzebne. Mając do dyspozycji taką pojedynczą klasę, uprościmy znacznie kod przykładów, które będą pojawiać się w dalszej części artykułu. Naszą klasę nazwiemy `AudioMonoIO`.

To tyle tytułem wstępem. Nasz plan jest gotowy, zatem do dzieła!

## 1. WEB AUDIO API – WPROWADZENIE

Pierwszym krokiem do wykorzystania potencjału, jaki oferuje nam Web Audio API, jest stworzenie tak zwanego kontekstu audio. Jak tego dokonać? Wystarczy powołać do życia obiekt klasy `AudioContext` za pomocą operatora `new`. Instancja tak stworzonego obiektu stanowić będzie kontener dla węzłów audio, w których tak naprawdę odbywać się będzie faktyczne przetwarzanie strumienia dźwiękowego. Węzły te możemy łączyć ze sobą w dowolny sposób za pomocą metody `connect`. Wyjście jednego z nich może posłużyć jako wejście innego. Na przykład `GainNode` umożliwia zmianę poziomu głośności sygnału podanego na wejściu. Na wyjściu otrzymamy zatem podgłośniony lub ściszczy strumień, który może posłużyć za wejście innego węzła bądź węzłów.

Węzły, które posiadają jedynie wyjścia, nazywamy źródłowy mi (`SourceNode`). W naszym przypadku najważniejszym będzie `MediaStreamSourceNode`, który umożliwia dostęp do strumienia audio z naszego mikrofonu. Jego inicjalizacja wymaga nieco więcej zabiegów niż w przypadku innych węzłów, ale to opisujemy bardziej szczegółowo w dalszej części artykułu. Przykładem innego węzła źródłowego jest `OscillatorNode`, który generuje sygnał okresowy o zadanej częstotliwości i kształcie.

W naszym kontekście audio może istnieć wiele węzłów źródłowych. Strumienie te oraz ich dalsze drogi wcale nie muszą tworzyć jednej wspólnej połączonej całości. Każdy z nich może tworzyć osobny graf efektów, przez który przechodząc będzie pewien strumień audio.

Skoro dostępne są m.in. węzły źródłowe, to czy istnieje także węzeł wyjścia? Oczywiście istnieje i nazywa się `AudioDestinationNode`. Dostęp do jego instancji zrealizowany jest przez właściwość `destination` stworzonego wcześniej obiektu kontekstu. Węzeł ten jest inicjalizowany automatycznie wraz z kontekstem audio i reprezentuje „finalny cel” strumienia audio. W większości przypadków będzie on równoznaczny z głośnikami naszego komputera. Jeżeli zatem chcemy usłyszeć dźwięk wyprodukowany przez graf naszych nodów, należy ostatni jego element podłączyć do wspomnianego węzła destynacji.

Skoro instancja węzła destynacji dostępna jest w obiekcie kontekstu z automatu, to czy podobnie jest z innymi węzłami? Otóż nie – obiekty innych węzłów musimy utworzyć samodzielnie. Jest to całkiem logiczne, ponieważ w naszym grafie możemy zażyczyć sobie kilka nodów tego samego typu. Mając po jednej instancji każdego węzła, byłoby to po prostu niemożliwe. Tutaj z pomocą przychodzi nam sam kontekst audio. Udostępnia on metody twórcze dla każdego typu węzła. Innymi słowy operator `new` staje się zbędny.

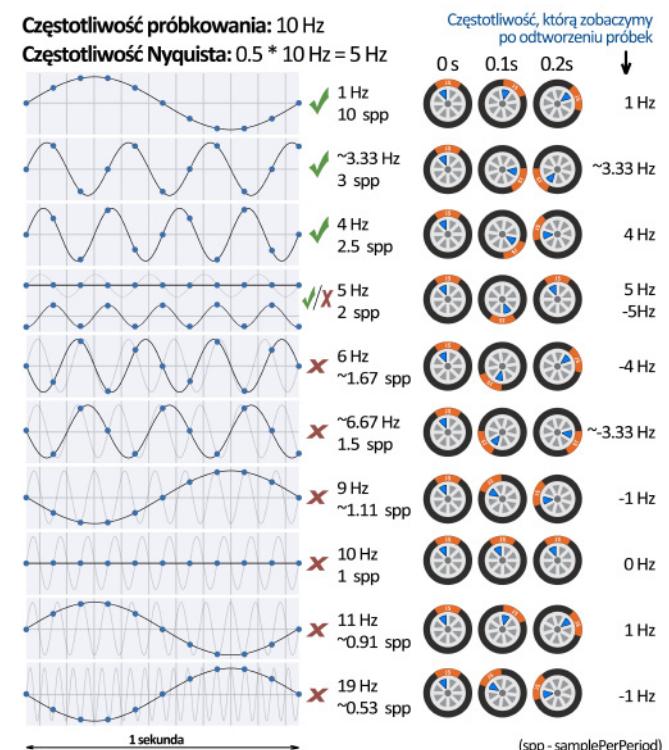
Chcąc utworzyć np. `GainNode`, musimy wywołać metodę `audioContext.createGain()`, oczywiście przy założeniu, że nasz kontekst audio znajduje się w zmiennej o nazwie `audioContext`. Uogólniając, konwencja tworzenia nodów jest następująca: chcąc powołać do życia obiekt typu `{Typ}Node`, musimy posłużyć się metodą `audioContext.create{Typ}()`.

Dlaczego zrezygnowano z podejścia wykorzystującego operator `new`? Zapewne między innymi dlatego, że każdy z węzłów działa tylko w obrębie swojego kontekstu audio. Próba wywołania metody `connect` na nodach należących do różnych kontekstów wyrzuci wyjątek. Użycie wzorca metody twórczej pozwala m.in. „pod maską” dokonać automatycznego powiązania stworzonego węzła z odpowiadającym mu kontekstem.

W Web Audio API dostępnych jest prawie 20 różnych typów nodów. W tym artykule skupimy się tylko na tych, które umożliwiają nam realizację transmisji danych. Zanim jednak przejdziemy do ich szczegółowego omawiania, zaczniemy od pojęć podstawowych.

## 2. KILKA SŁÓW O PRÓBKOWANIU

Po poprawnej inicjalizacji obiektu kontekstu audio możemy z niego odczytać bardzo ważną wartość, jaką jest częstotliwość próbkowania: `audioContext.sampleRate`. W większości przypadków będzie ona równa 44.1 kHz lub 48 kHz. Dlaczego akurat tyle? Dlatego aby w pełni pokryć zakres częstotliwości, jakie są w stanie usłyszeć ludzie. Przypomnijmy: przeciętny człowiek słyszy częstotliwości z zakresu od 16 Hz do 20 kHz. Skoro zatem największa częstotliwość słyszana przez człowieka to 20 kHz, dlaczego więc próbujemy z częstotliwością ponad dwukrotnie większą? Aby odpowiedzieć na to pytanie, prześledzmy przypadek z dużo mniejszą częstotliwością próbkowaniaową równą 10 Hz (Rysunek 1).



Rysunek 1. Częstotliwość próbkowania oraz częstotliwość Nyquista

Jak widzimy, przy częstotliwości próbkowania 10 Hz zapisanie jednego pełnego okresu sinusoidy o częstotliwości 1 Hz wymaga użycia 10 sampli. W poprzedniej części artykułu do tego celu używaliśmy określenia `samplePerPeriod`. Zwiększenie częstotliwości sinusoidy sprawia, że coraz mniej sampli przypada na jej jeden okres. Okazuje się, że gdy dochodzimy do częstotliwości sinusoidy równej połowie częstotliwości próbkowania (5 Hz), na jeden jej okres przypadają dokładnie dwa sample. W zależności od fazy fali może być to już za mało, by móc ją poprawnie zapisać. Po przekroczeniu 5 Hz w naszym materiale zaczną pojawiać się częstotliwości, których nie ma w rzeczywistości. Zniekształcenia tego typu noszą nazwę aliasingu. Wróćmy jeszcze do tego zagadnienia w dalszej części artykułu.

Z tego przykładu możemy wyciągnąć prosty wniosek. Maksymalna częstotliwość zawarta w sygnale nie może przekraczać połowy częstotliwości próbkowania. Wartość połowy częstotliwości próbkowania nazywana jest częstotliwością Nyquista. Warto to zapamiętać, gdyż ta nazwa będzie się przewijać w dalszej części tego artykułu.

Wróćmy do kontekstu audio. Dla jednej z popularnych częstotliwości próbkowania 44100 Hz połową będzie 22050 Hz. Oznacza to, że najwyższa częstotliwość, jaką będziemy mogli poprawnie od-

tworzyć z zapisanych próbek, będzie mniejsza niż 22050 Hz. Skoro przeciętny człowiek słyszy dźwięki o maksymalnej częstotliwości 20 kHz, po co nam zatem zakres od 20 kHz do 22.05 kHz? Wynika to z faktu, że sygnał przed próbkowaniem musi zostać poddany filtracji (filtr dolnoprzepustowy), aby usunąć wszystkie częstotliwości, których nie bylibyśmy w stanie poprawnie zapisać. Problem w tym, że nie istnieje filtr, który przepuszcza w pełni częstotliwości poniżej 20 kHz, usuwając przy tym całkowicie częstotliwości ponad 20 kHz. W praktyce zawsze musi istnieć pewien zakres pośredni. W tym przypadku ten zakres ma szerokość 2050 Hz. Czy jednak to my musimy martwić się filtrowaniem sygnału z mikrofonu przed jego próbkowaniem? Odpowiedź brzmi: nie, gdyż robi to za nas sprzęt. Od Web Audio API dostajemy jedynie informację, jaką jest obowiązująca wartość częstotliwości próbkowania. Niestety nie istnieje żadna metoda do zmiany tej wartości z poziomu skryptu.

Dla dociekliwych: wartość 44.1 kHz wybrano jakiś czas temu w wyniku porozumienia twórców sprzętu. Stała się ona standardem np. w formacie płyt CD z muzyką. Dodatkowo ma ciekawą właściwość, gdyż jest ona równa iloczynowi kwadratów czterech pierwszych liczb pierwszych:  $44100 = 2^7 * 3^2 * 5^2 * 7^2$ . Sprawia to, że dla wielu wygodnych dzielników (np. 2, 3, 4, 5, 6, 7, 9) dzieli się bez reszty. Warto jednak dodać, że obecnie odchodzi się od tej wartości na rzecz 48 kHz ( $48000 = 2^7 * 3 * 5^3$ ).

## 3. FALA AKUSTYCZNA

W tej sekcji opowiemu nieco, jak fala akustyczna podróżuje w powietrzu oraz jak ją generować i odbierać. Otóż rozchodzi się ona w formie zaburzeń jego gęstości i ciśnienia z prędkością rzędu 320–350 m/s (nieco ponad 1000 km/h). Przekłada się to na długość fali rzędu np. 6.7 m dla tonu 50 Hz, 33.5 cm dla tonu 1 kHz czy 2.2 cm dla tonu 15 kHz. Do wytwarzania takich zaburzeń nasze głośniki muszą być wyposażone w pewien ruchomy element. Ten element nazywamy membraną. Jej wychylenie jest zależne od naszych próbek audio. Wartość binarna każdej z nich trafia kolejno w stałych odstępach czasu na przetwornik cyfrowo-analogowy. Otrzymane napięcie, po wygładzeniu „schodków” filtrem, używane jest do sterowania wychyleniem membrany.

Gdy zatem w naszych próbkach zapisany jest np. czysty sinus o częstotliwości 8 kHz, spowoduje on naprzemienne wypychanie i cofanie membrany 8 tysięcy razy w ciągu sekundy. Skutkuje to naprzemiennym kompresowaniem i rozrzedzaniem powietrza przy membranie. W efekcie otrzymamy falę akustyczną.

W mikrofonie w skrócie zachodzi proces odwrotny. Tam także znajduje się membrana, lecz to nie napięcie, ale fala akustyczna.

na wprowadzi ją w drgania. Drgania te powodują proporcjonalne zmiany napięcia. Tak otrzymany sygnał przepuszczany jest przez filtr dolnoprzepustowy, którego celem w tym przypadku jest pozostawienie tylko dolnej części widma poniżej częstotliwości Nyquista. Ostatecznie sygnał przepuszczony jest przez przetwornik analogowo-cyfrowy, który umożliwia zapisanie wartości napięcia z membrany w postaci liczby binarnej. Jest ona wartością naszego pojedynczego sampła audio. Przy częstotliwości próbkowania 44.1 kHz otrzymujemy zatem nieco ponad 44 tysięcy liczb binarnych w każdej sekundzie. Najczęściej każda z nich jest 16-bitowym typem całkowitoliczbowym ze znakiem. Przekłada się to na wartości próbków z przedziału od -32768 do 32767.

## 4. PIERWSZA APLIKACJA W WEB AUDIO API

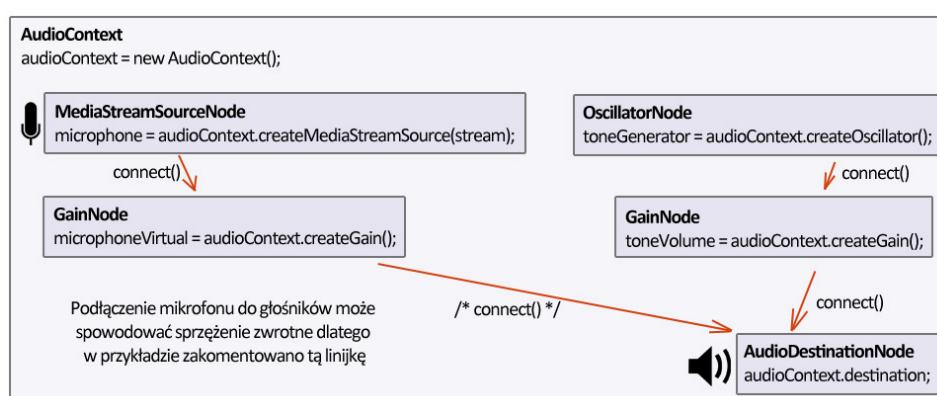
Zadaniem naszej pierwszej aplikacji będzie generowanie co jedną sekundę prostego tonu (czysta pojedyncza sinusoida) o losowej częstotliwości z zakresu od 1000 Hz do 3000 Hz oraz losowej głośności z zakresu od 0% do 1%. Ograniczenie maksymalnej głośności jest celowe. Zwyczajnie nie chcę mieć na sumieniu słuchu kogoś, kto otworzy ten przykład, mając słuchawki na uszach na pełnej głośności. Dodatkowo nasza aplikacja będzie inicjalizować wspomniany wcześniej węzeł mikrofonu. Wymaga on użycia strumienia audio zwracanego przez inne API przeglądarki.

W pierwszym przykładzie nie będziemy się jeszcze zajmować przetwarzaniem napływających sampli. Jedyne, co możemy tutaj zrobić, to podłączyć węzeł mikrofonu do węzła destynacji, co da nam możliwość słuchania np. swojego głosu na głośnikach. Z uwagi jednak na możliwość wystąpienia sprzężenia zwrotnego linijka łącząca te dwa węzły została zakomentowana. Zapraszam do testowania we własnym zakresie. Na Rysunku 2 przedstawiono węzły użyte w przykładzie w postaci grafu.

W Listingu 1 przedstawiono kod pierwszej aplikacji. Dla oszczędzenia miejsca pominięto w nim obsługę błędów oraz zabiegi potrzebne do pracy na starszych przeglądarkach. Brakujące elementy umieścimy w klasie AudioMonoIO.

**Listing 1. Generowanie prostego tonu oraz inicjalizacja węzła ze strumieniem z mikrofonu**

```
function init() {
    audioContext = new AudioContext();
    // input
    microphoneVirtual = audioContext.createGain();
```



Rysunek 2. Graf węzłów pierwszej aplikacji

```

connectMicrophoneTo(microphoneVirtual);
// output
toneGenerator = audioContext.createOscillator();
toneVolume = audioContext.createGain();
updateRandomTone();
toneGenerator.start();
toneGenerator.connect(toneVolume);
setInterval(updateRandomTone, 1000);

// przypięcie węzłów do głośników
/*
microphoneVirtual.connect( // uwaga na sprzężenie zwrotne
    audioContext.destination
);
*/
toneVolume.connect(audioContext.destination);
}

function connectMicrophoneTo(microphoneVirtual) {
    var constraints, audioConfig;

    audioConfig = {
        googEchoCancellation: false,
        googAutoGainControl: false,
        googNoiseSuppression: false,
        googHighpassFilter: false
    };
    constraints = {
        video: false,
        audio: {
            mandatory: audioConfig,
            optional: []
        }
    };
    navigator.mediaDevices.getUserMedia(constraints)
        .then(function (stream) {
            microphone = audioContext
                .createMediaStreamSource(stream);
            microphone.connect(microphoneVirtual);
        });
}

function updateRandomTone() {
    var frequency, gain, now;

    now = audioContext.currentTime;
    frequency = 1000 + Math.random() * 2000;
    toneGenerator.frequency.value = frequency;
    toneGenerator.frequency.setValueAtTime(frequency, now);

    gain = Math.random() * 0.01;
    toneVolume.gain.value = gain;
    toneVolume.gain.setValueAtTime(gain, now);
}
}

```

Pełny kod powyższego przykładu dostępny jest poniżej:

- » <https://audio-network.rypula.pl/audio-context-init>
- » <https://audio-network.rypula.pl/audio-context-init-src>

W dalszej części przyjrzyjmy się bardziej szczegółowo użytym węzłom. Wszystkie opisy zakładają, że instancja naszego kontekstu audio znajduje się w zmiennej o nazwie `audioContext`.

## 4.1 GainNode

Węzeł ten jest bardzo prosty. Można go stworzyć przy użyciu obiektu kontekstu audio poprzez wywołanie na nim metody `createGain`. Jego zadaniem jest zmiana poziomu głośności strumienia, który przez niego przechodzi. Sprowadza się to do pomnożenia wartości napływających sampli przez zadany współczynnik ( $>1$  głośniej,  $<1$  ciszej). Z dokumentacji możemy wyczytać, że poziomem tym sterujemy, przypisując odpowiednią wartość do zmiennej `value` we właściwości `gain`. Jest to prawda, lecz wiąże się z tym jeden problem. Otóż czas potrzebny na zmianę tej wartości jest zależny od implementacji Web Audio API. W niektórych przeglądarkach (np. Chrome) domyślnie włączone jest „wygładzenie zmian” tejże wartości. Ozna-

czy to, że głośność nie zostanie natychmiast zmieniona z poziomu A do poziomu B. Zamiast tego wartość ta będzie płynnie zmieniać się w czasie, tak by finalnie zatrzymać się na wartości B. W przypadku Chrome domyślny czas wygładzania wynosi 50 ms. W przeglądarce Firefox czas ten jest domyślnie zerowy.

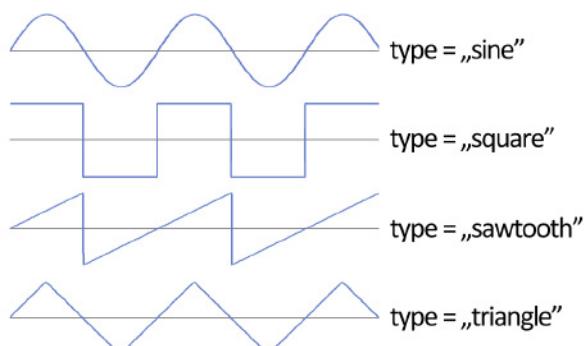
Podobne zachowanie występuje także w węźle `OscillatorNode`. Jak w przypadku zmian głośności nie jest to tak mocno zaauważalne, tak przy zmianach częstotliwości stanowi to już większy problem. My jednak naprawimy oba przypadki. Rozwiązaniem jest metoda `setValueAtTime`. Przyjmuje ona dwa parametry. Pierwszy to wspomniana wcześniej wartość współczynnika, drugi to czas wykonania tej zmiany. W naszym przypadku zmian chcemy wykonać natychmiast. W obiekcie naszego kontekstu audio dostępna jest specjalna właściwość zwracająca aktualny czas – `audioContext.currentTime`.

Podsumujmy zatem wszystko. By uniknąć problemów, należy najpierw ustawić nową wartość w „tradycyjny” sposób przy użyciu `gain.value`, a następnie w kolejnej linijce użyć metody `setValueAtTime`. Przykład z życia wzięty znajdziemy w Listingu 1 w metodzie `updateRandomTone`.

## 4.2 OscillatorNode

Węzeł ten umożliwia generowanie na swoim wyjściu sygnału określonego o zadanej częstotliwości. Tworzymy go za pomocą metody `createOscillator`. Aby węzeł rozpoczął generowanie przebiegu, musimy go jeszcze „uruchomić” poprzez wywołanie metody `start`. Z dokumentacji wyczytamy, że emitowaną częstotliwość możemy zmienić, przypisując nową wartość do zmiennej `value` we właściwości `frequency`. Tutaj także ustawienie nowej wartości skutkuje nieco odmiennym zachowaniem w zależności od implementacji modułu Web Audio API w przeglądarce. Do uzyskania natychmiastowej zmiany musimy zatem także posłużyć się dodatkowo metodą `setValueAtTime`. To rozwiązanie wyeliminuje niepożądane, płynne „przepływanie” dźwięku do zadanej częstotliwości. Efekt ten mógłby wpływać negatywnie na naszą transmisję danych.

Domyślnie wybranym typem kształtu generowanej fali jest „sine” (typ `sine`). Do dyspozycji mamy także trzy inne kształty: „prostokąt”, „piłę” oraz „trójkąt” (Rysunek 3). Aby zmienić domyślny typ, wystarczy do właściwości `type` przypisać string o wartości odpowiednio „square”, „sawtooth” lub „triangle”.



Rysunek 3. Kształty wbudowane w `OscillatorNode`

Istnieje także możliwość skonfigurowania własnego kształtu za pomocą metody `setPeriodicWave` oraz klasy `PeriodicWave`. Do celów transmisji danych w zasadzie moglibyśmy wybrać typ wbudowany „sine”. Wiąże się z tym jednak pewne ograniczenie. Otóż

domyślny sinus nie pozwala na zmianę fazy emitowanego przebiegu okresowego. By zatem mieć pełną kontrolę nad sygnałem, musimy skorzystać z możliwości konfiguracji własnego kształtu przebiegu. Oczywiście także wymodelujemy sinusa, jednak dodatkowo z możliwością ustawienia przesunięcia fazowego.

## 4.2.1 Sygnały okresowe – teoria

Konfiguracja obiektu PeriodicWave wymaga jednak trochę wstępnie teoretycznego o budowie sygnałów okresowych. Wykroczymy nieco poza zakres minimalnej wiedzy niezbędnej do transmisji danych. Jest to jednak o tyle istotne, że nie sposób o tym nie powiedzieć. Naszym celem jest stworzenie okresowego przebiegu o zadanej częstotliwości podstawowej. Innymi słowy, gdy narysujemy wynikowe próbki na wykresie w funkcji czasu, ich kształt powinien się cyklicznie powtarzać. Takim zachowaniem cechuje się oczywiście „czysta” sinusoida.

Zadajmy sobie teraz pytanie – czy możliwe jest manipulowanie wyglądem sygnału tak, by wciąż zachować okresowe powtarzanie się tego samego kształtu z częstotliwością podstawową? Otóż jest to możliwe. Wystarczy do naszej „bazowej” sinusoidy dodać drugą sinusoidę, której częstotliwość będzie dwukrotnie większa niż częstotliwość podstawowa. Dwukrotne zwiększenie częstotliwości sprawi, że w czasie trwania jednego cyklu naszej pierwszej sinusoidy „zmieszcza” się dokładnie dwa pełne cykle naszej drugiej sinusoidy. Idąc dalej, do tak zmienionego sygnału możemy dodać trzecią sinusoidę, której częstotliwość będzie 3x większa od częstotliwości podstawowej. Oznacza to, że w jednym okresie naszej pierwszej sinusoidy zmieszcza się dokładnie 3 pełne okresy trzeciej sinusoidy.

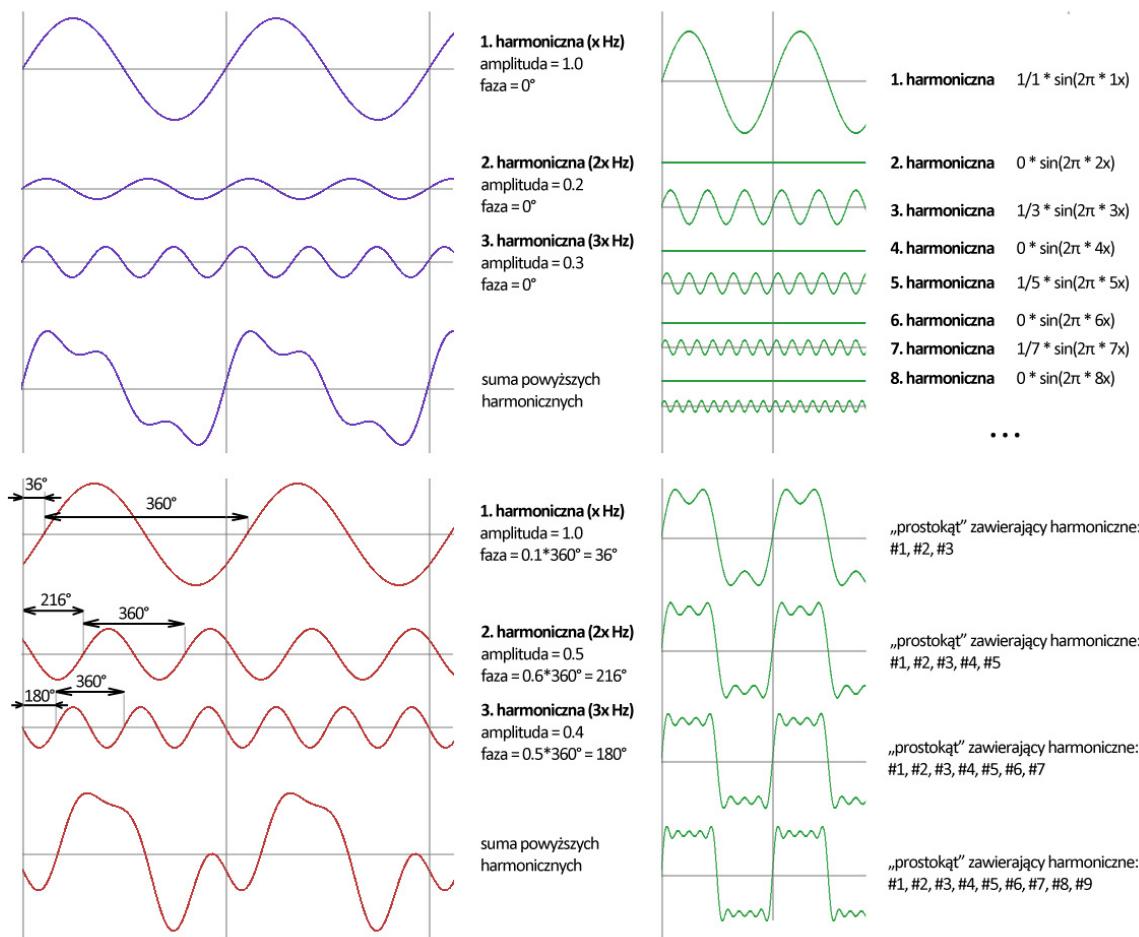
W analogiczny sposób możemy dokładać kolejne sinusoidy. Oczywiście limitem częstotliwości będzie częstotliwość Nyquista.

Zauważmy, że każdy nowy cykl sinusoidy o częstotliwości podstawowej będzie także początkiem cykłów wszystkich innych sinusoid składających się na nasz finalny sygnał. Kształt następnego cyklu będzie zatem taki sam. Sinusoidy spełniające te warunki называемy „składowymi harmonicznymi”, a pierwszą z nich „składową podstawową” lub „tonem podstawowym”. Jego częstotliwość jest równa naszej częstotliwości podstawowej. Sterując amplitudami harmonicznych oraz ich przesunięciem fazowym, możemy modelować kształt naszego przebiegu. Na Rysunku 4 przedstawiono kilka przykładów pokazujących, jak zmienia się przebieg w zależności od parametrów harmonicznych. Brak harmonicznej oznacza, że jej amplituda jest równa零.

Jak widzimy, za pomocą odpowiedniej liczby harmonicznych, ich amplitud oraz przesunięć fazowych możemy wygenerować w zasadzie dowolny kształt, włączając w to typy wbudowane w OscillatorNode, takie jak „prostokąt”, „piła” czy „trójkąt”.

## 4.2.2 Sygnały okresowe w praktyce, czyli klasa PeriodicWave

Do konfiguracji fali okresowej należy posłużyć się obiektem klasy PeriodicWave. Tworzymy go, wywołując metodę createPeriodicWave na obiekcie naszego kontekstu audio. Przyjmuje ona dwa parametry będące tablicami liczb zmiennoprzecinkowych. Tablice te powinny mieć takie same długości. Każda para liczb o takich samych indeksach w tablicach opisuje jedną składową harmoniczną naszego przebiegu. Do opisu harmonicznych



Rysunek 4. Modelowanie kształtu sygnału okresowego za pomocą harmonicznych

klasa PeriodicWave używa liczb zespolonych. Pierwszą tablicę traktujemy jako części rzeczywiste, natomiast drugą jako części urojone liczb zespolonych. Poprzez odpowiednie sterowanie wartościami w tablicach możemy wygenerować w sygnale wynikowym sinusoidy składowe o takich amplitudach i przesunięciach fazowych, jakie sobie życzymy.

Na tym etapie możemy zauważać podobieństwo do tego, o czym mówiliśmy przy okazji omawiania Dyskretnej Transformaty Fouriera w poprzedniej części tego artykułu. Wynikiem jego pracy były liczby zespolone mówiące o amplitudzie oraz przesunięciu fazowym badanych fal. Czy zatem OscillatorNode realizuje w pewnym sensie proces odwrotny? Oczywiście tak! Klasa PeriodicWave jest zdolna do transformacji opisu harmonicznych z dziedziny częstotliwości na dziedzinę czasu.

Bardzo ważny jest fakt, że para liczb znajdujących się w tablicach pod indeksem 0 nie opisuje pierwszej harmonicznej, lecz częstotliwość 0 Hz. Cokolwiek byśmy tam jednak nie wpisali, przeglądarka i tak wyzeruje tę wartość. Dla dociekiowych – moduł tej liczby zespolonej to tak zwany „DC offset”, czyli nic innego jak wartość średnia wszystkich próbek. Zmieniając tę wartość, moglibyśmy przesuwać nasz przebieg w górę lub w dół na wykresie.

Dlaczego zatem w implementacjach Web Audio API ta wartość jest zerowana? Takie przesunięcie przebiegu na wykresie nie ma po prostu większego sensu, gdyż nie dałoby żadnego zauważalnego efektu akustycznego. Membrana głośnika drgałaby bowiem tak samo z tą różnicą, że średnio byłaby nieco głębiej lub nieco płycej w głośniku. To jednak zredukowałoby zakres, w jakim znajdowałaby się nasz sygnał. Zabawa zaczyna się zatem od indeksu 1, który reprezentuje naszą częstotliwość podstawową (pierwsza harmoniczna). Każdy następny indeks będzie reprezentował kolejną harmoniczną.

Zauważmy, że klasa PeriodicWave nie posiada żadnej informacji o wartości częstotliwości podstawowej. Podajemy jedynie współczynniki szeregu harmonicznego. To OscillatorNode podczas swojej pracy generuje z podanych współczynników ostateczny strumień próbek. Częstotliwość jego pierwszej harmonicznej odpowiada ustalonej wcześniej wartości właściwości frequency.

W Listingu 2 pokazano, w jaki sposób wygenerować prosty kształt za pomocą klas OscillatorNode oraz PeriodicWave. Dla prostoty wybrano kształt „prostokąt”, gdyż do opisu jego harmonicznych nie potrzebujemy żadnych przesunięć fazowych. Wystellarz w zupełności odpowiednio dobrane wartości amplitud.

#### Listing 2. Generowanie przebiegu okresowego o kształcie „prostokąt”

```
oscillatorNode = audioContext.createOscillator();
baseFrequency = 1000;
amplitudeList = [
  0, // [0] 0 Hz (DC offset) ta wartość i tak jest zerowana
  1/1, // [1] harmoniczna #1 - {baseFreq} x 1 ton podstawowy
  0, // [2] harmoniczna #2 - {baseFreq} x 2
  1/3, // [3] harmoniczna #3 - {baseFreq} x 3
  0, // [4] harmoniczna #4 - {baseFreq} x 4
  1/5 // [5] harmoniczna #5 - {baseFreq} x 5
  // dla parzystych numerów amplituda wynosi zero,
  // dla nieparzystych amplituda jest równa 1/{nrHarm.}
];
// kształt prostokąt nie wymaga przesunięć
// harmonicznych w fazie - część rzeczywista
// to same zera
real = new Float32Array(amplitudeList.length);
// część urojona to dane z tablicy amplitud
imag = new Float32Array(amplitudeList);

periodicWave = audioContext.createPeriodicWave(real, imag);
oscillatorNode.setPeriodicWave(periodicWave);
oscillatorNode.frequency.value = baseFrequency;
```

```
oscillatorNode.frequency.setValueAtTime(
  baseFrequency, audioContext.currentTime
);
oscillatorNode.start();
```

Pełny kod oraz źródła znajdziemy poniżej:

- » <https://audio-network.rypula.pl/square-wave>
- » <https://audio-network.rypula.pl/square-wave-src>

Oczywiście nasz przebieg będzie tylko przybliżeniem idealnego przebiegu o kształcie „prostokąt”. Do jego budowy użyliśmy tylko 5 harmonicznych, dlatego nasz prostokątny kształt będzie w rzeczywistości pofałowany (Rysunek 4). Jego wygląd możemy poprawić poprzez dodanie kolejnych harmonicznych o odpowiednich amplitudach.

Łatwo zauważyc, że nasze wartości amplitud podaliśmy tylko w tablicy reprezentującej części urojone. O tym, dlaczego tak zrobiliśmy, powiemy nieco dalej.

Warte wspomnienia jest to, że wyjście z oscylatora jest normalizowane. Oznacza to, że wartość żadnej z próbek nie powinna przekroczyć zakresu od -1 do 1. Definiując tablice części rzeczywistej i urojonej, zwróciśmy zatem uwagę nie na faktyczne wartości długości wektorów harmonicznych (wartości bezwzględne liczb zespolonych), tylko na proporcje między tymi długościami.

Skoro zatem finalny strumień audio jest normalizowany, tak by wypełnić całość zakresu od -1 do 1, to jak go „przyciszczyć”? Odpowiedzią jest GainNode. Wystarczy wyjście z oscylatora podpiąć bezpośrednio na jego wejście i tam sterować finalną głośnością. Takie rozwiązanie zastosowano w przykładzie z Listingu 1 oraz Listingu 3.

#### 4.2.3 Klasa PeriodicWave – dodajemy przesunięcie fazowe

Na koniec sprawdźmy, w jaki sposób możemy sterować przesunięciem fazowym. W skrypcie z Listingu 2 wartości współczynników harmonicznych wpisaliśmy do tablicy reprezentującej części urojone oraz były one zawsze dodatnie. Stosując analogię do wektorów 2D, możemy powiedzieć, że wszystkie z nich były zawsze skierowane w górę na godzinę dwunastą. Dla obiektu klasy PeriodicWave jest to równoznaczne z brakiem przesunięcia fazy. W przypadku kształtu „prostokąt” było to wystarczające.

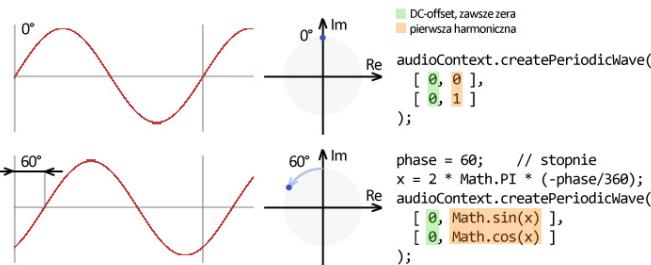
Zanim przejdziemy dalej, warto zaznaczyć, że będziemy tu rozważyć dwa przypadki przesunięcia fazowego. Pierwszy to „lokalne” przesunięcie fazy każdej ze składowych z osobna. W połączeniu z odpowiednią wartością amplitudy daje to możliwość modelowania kształtu przebiegu. Drugi to „globalne” przesunięcie w fazie całego naszego kształtu. Będziemy zatem szukać sposobu na taką zmianę przesunięć fazowych wszystkich harmonicznych, by w efekcie nasz gotowy kształt nie „rozjechał” się.

Warto teraz przypomnieć sobie diagram konstelacji z poprzedniej części artykułu. Punkt na diagramie reprezentował udział konkretnej częstotliwości w badanym sygnale. Odległość tego punktu od środka układu współrzędnych mówiła o amplitudzie powiązanej z nim fali, natomiast kąt pomiędzy osią Y a punktem o przesunięciu fazowym. Warto na tym etapie zaznaczyć jedną ważną różnicę. W przypadku diagramu konstelacji przyjęliśmy obrót punktu zgodnie z ruchem wskazówek zegara, aby zwiastować przesunięcie wykresu w prawą stronę. Jest to odwrotna konwencja do tej, jaka obowiązuje w klasie PeriodicWave. Warto o tym pamiętać, by nie pogubić się w obrotach.

Co zatem zrobić, aby przesunąć harmoniczną w fazie? Musimy zwyczajnie obrócić powiązany z nią wektor o pewien kąt, nie zmieniając jego długości. W którą stronę? Jak już ustaliliśmy – przeciwnie do ruchu wskazówek zegara. W rezultacie nasza sinusoida zostanie

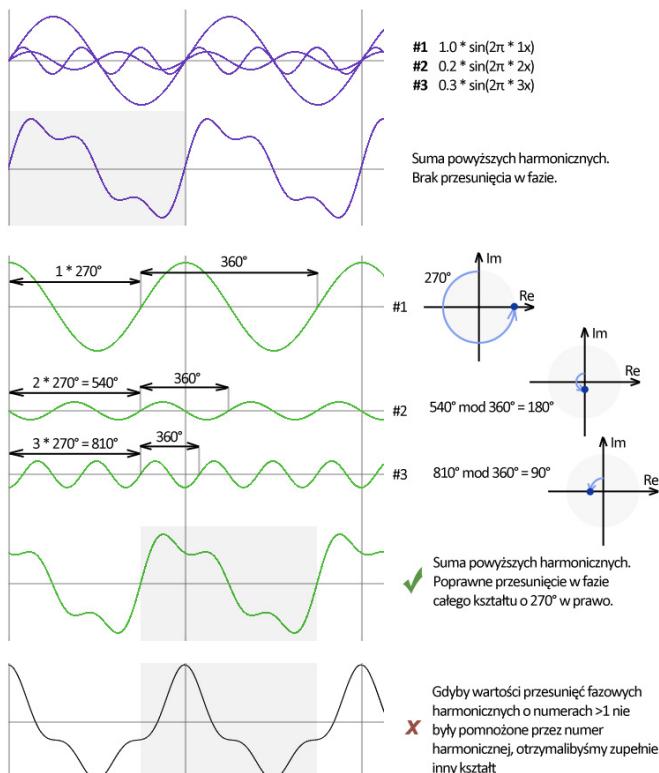
# PROGRAMOWANIE APLIKACJI WEBOWYCH

przesunięta w prawo. Na Rysunku 5 przedstawiono najprostszy przypadek, w którym nasz przebieg posiada tylko pierwszą harmoniczną. Oczywiście postępując analogicznie, możemy przesuwać w fazie harmoniczną o dowolnym numerze.



Rysunek 5. Przesuwanie składowej harmonicznej w fazie

Na tym etapie powinniśmy już wiedzieć, jak przy użyciu klasy PeriodicWave wymodelować w zasadzie dowolny kształt. Sprawdzmy więc teraz, w jaki sposób nasz kształt przesuwać w fazie jako całość. W przypadku pierwszej harmonicznej sprawa jest prosta – wartość kąta obrotu jest równa wymaganemu przesunięciu fazowemu. Na przykład obrót wektora pierwszej harmonicznej o 90 stopni da nam w rezultacie przesunięcie o 1/4 długości fali o częstotliwości podstawowej w prawo. Obrót o 180 stopni to przesunięcie o 1/2 długości fali. W przypadku kolejnych harmonicznych wydawać by się mogło, że należy także obracać o ten sam kąt. Okazuje się jednak, że to spowodowałoby „rozjechanie” się kształtu naszego przebiegu (Rysunek 6).



Rysunek 6. Przesunięcie w fazie całego kształtu zbudowanego z wielu harmonicznych

Jak widzimy, aby zachować kształt wykresu podczas przesunięcia, wektor każdej następnej harmonicznej o numerze większym od 1 musi „dogonić” obrót harmonicznej o numerze 1. Dzieje się tak dlatego, że długość fali każdej kolejnej harmonicznej stopniowo

zmniejsza się. Aby to skompensować, należy pomnożyć kąt obrotu przez numer harmonicznej. W efekcie każda następna harmoniczna dogoni przesunięcie pierwszej z nich. To sprawi, że wszystkie harmoniczne zaczną swój nowy cykl w tym samym miejscu, a nasz kształt w każdym kolejnym okresie częstotliwości podstawowej pozostanie taki sam. Oczywiście, gdy harmoniczne naszego kształtu miały już pewne „lokalne” przesunięcia w fazie decydujące o jego wyglądzie, należy je także dodać.

W Listingu 3 przedstawiono kod generujący również przebieg o kształcie „prostokąt”, jednak tym razem z możliwością zmiany zarówno „globalnego” przesunięcia fazowego, jak i poszczególnych harmonicznych z osobna. Dodatkowo użyty został węzeł GainNode umożliwiający zmianę głośności strumienia wytworzzonego przez OscillatorNode.

Listing 3. Generowanie przebiegu okresowego o kształcie „prostokąt” – tym razem z możliwością przesunięcia fazowego i zmiany głośności

```
volume = 0.5; // 50% głośności  
phase = 0.75; // przesunięcie całości w prawo o 270 stopni  
hAmplitude = [ // DC offset zostało pominięty  
  1, 0, 1/3, 0, 1/5  
];  
hPhase = [0, 0, 0, 0, 0]; // kształt prostokąt nie wymaga  
// przesunięcia faz harmonicznych  
real = new Float32Array(1 + hAmplitude.length);  
imag = new Float32Array(1 + hAmplitude.length);  
globalP = TWO_PI * (-phase); // 'globalne' przesunięcie fazy  
real[0] = 0; // DC-offset  
imag[0] = 0; // DC-offset  
for (i = 0; i < hAmplitude.length; i++) {  
  harmonicNr = 1 + i;  
  localP = TWO_PI * (-hPhase[i]); // 'lokalne' przesunięcie  
  real[harmonicNr] =  
    hAmplitude[i] * Math.sin(globalP * harmonicNr + localP);  
  imag[harmonicNr] =  
    hAmplitude[i] * Math.cos(globalP * harmonicNr + localP);  
}  
// ...  
oscillatorNode.connect(gainNode);  
gainNode.connect(audioContext.destination);  
// ...  
periodicWave = audioContext.createPeriodicWave(real, imag);  
oscillatorNode.setPeriodicWave(periodicWave);  
// ...  
gainNode.gain.value = volume;  
gainNode.gain.setValueAtTime(  
  volume,  
  audioContext.currentTime  
);
```

Pełny kod oraz źródła znajdziemy poniżej:

» <https://audio-network.rypula.pl/square-wave-phase>  
» <https://audio-network.rypula.pl/square-wave-phase-src>

Zastanówmy się teraz, czy jesteśmy w stanie sami zaimplementować funkcjonalność klas OscillatorNode oraz PeriodicWave. Dokładniej chodzi tu o możliwość generowania przebiegów okresowych składających się z wielu harmonicznych. Przypomnijmy: w poprzedniej części artykułu do generowania przebiegów okresowych używaliśmy funkcji generateSine. Zwracała ona kolejne wartości próbek sinusoidy o zadanej częstotliwości (podanej w samplePerPeriod), fazie oraz amplitudzie. Bardziej złożone przebiegi generowaliśmy poprzez sumowanie wartości próbek otrzymanych z fal składowych. Gdy zatem odpowiednio dobrzemy częstotliwości fal składowych, po zsumowaniu jesteśmy w stanie osiągnąć dokładnie taki sam efekt. Mamy zatem dwie drogi do osiągnięcia tego samego celu.

### 4.3 MediaStreamSourceNode

Po przeczytaniu opisu klasy `OscillatorNode` i `PeriodicWave` wiemy już, jak emitować dźwięk. Teraz skupimy się na tym, jak ten dźwięk wychwycić. Z Rysunku 2 możemy odczytać, że część grafu odpowiadająca za przechwytywanie danych z mikrofonu składa się z dwóch węzłów. Pierwszy to `MediaStreamSourceNode`, a drugi to `GainNode`. O tym, jaka jest rola drugiego, powiemy za chwilę. Pierwszy natomiast możemy stworzyć, wywołując metodę `createMediaStreamSource` z obiektu naszego kontekstu audio. To jednak nie koniec, gdyż metoda ta wymaga parametru typu `MediaStream`. By stworzyć obiekt o takim typie, musimy posłużyć się innym API przeglądarki, a dokładniej metodą `getUserMedia` z obiekcie `navigator.mediaDevices`. Jak widać, faktyczny strumień danych z mikrofonu nie pochodzi z Web Audio API. `MediaStreamSourceNode` jedynie go opakowuje, tak by można go było podłączać do innych węzłów.

Metoda `getUserMedia` przyjmuje parametr będący „listą wymogów” co do strumienia, jaki chcemy otrzymać. Lista ta to obiekt z dwoma polami – `audio` i `video`. W naszym przypadku interesuje nas tylko strumień `audio`. Najprostsza konfiguracja to po prostu przypisanie do pola `audio` wartości `true`, a do `video` wartości `false`. Pominięcie pola `video` jest równoznaczne z podaniem wartości `false`.

Okazuje się, że sama wartość `true` nie wystarczy. Przeglądarki domyślnie starają się „poprawić” dane napływające z mikrofonu poprzez zastosowanie różnego rodzaju filtrów. Chodzi tutaj o automatyczne dostosowywanie głośności czy też usunięcie zakłóceń, takich jak biały szum, echo itp.

Aby zmienić domyślną konfigurację, należy w polu `audio` umieścić obiekt z dwoma polami – `optional` i `mandatory`. W każdym z nim możemy umieścić kolejny obiekt z listą ustawień oraz flag `true` lub `false`. Tutaj niestety ciężko dokładnie powiedzieć, jaka jest lista wszystkich opcji i czy powinniśmy umieścić ją w polu `optional` czy `mandatory`. Na stan obecny można odnieść wrażenie, że to API ciągle się rozwija i finalny kształt nie został jeszcze ustalony. Przeszukując Internet, najczęściej jednak można natrafić na strukturę z Listingu 1. Oprócz obiektu konfiguracyjnego warto sprawdzić ustawienia mikrofonu w systemie. Czasem możemy tam znaleźć kilka opcji, które warto wyłączyć (np. *Automatic Gain Control* – AGC). Generalnie im bardziej „nietknięte” będą nasze sample, tym lepiej.

O tym, jak ważne jest wyłączenie wszystkich „poprawiaczy” dźwięku, przekonałem się osobiste, próbując przechwycić ton o stałej częstotliwości na jednym z laptopów. Pozostawienie domyślnej konfiguracji spowodowało, że czysta sinusoida nadawana przez inne urządzenie zostało całkowicie wytłumiona na urządzeniu odbierającym po kilku milisekundach. Może jest to niezły bajer w przypadku nagrywania głosu, jednak w przypadku transmisji danych, w której ta sinusoida byłaby nośną dla danych, jest to, dekadatnie mówiąc, niewskazane.

Ok, wiemy, jak wygląda konfiguracja, ale ciągle nie odpowiedzieliśmy na pytanie, dlaczego zdecydowaliśmy się na dwa węzły. Otóż metoda `getUserMedia` zwraca nam obiekt typu `Promise`. Oznacza to, że wynik działania tej metody wcale nie musi być natychmiastowy. Z punktu widzenia naszej aplikacji możemy jednak zwyczajnie chcieć kontynuować konfigurację dalszych węzłów i połączeń między nimi. Dlatego właśnie dodany został drugi węzeł `GainNode`, który w zasadzie pełni tu rolę pośrednika. Możemy go zatem podłączyć do dalszej części grafu i traktować tak, jakby był on czymś w rodzaju wirtualnego mikrofonu.

Dlaczego w Web Audio API zastosowano rozwiązanie z promise? Otóż dlatego, że dostęp do mikrofonu czy kamery wymaga jawnej zgody użytkownika. Po wywołaniu metody `getUserMedia` przeglądarka wyświetla odpowiedni komunikat z pytaniem. Jest to oczywiście podyktowane względami bezpieczeństwa, gdyż użytkownik może zwyczajnie nie chcieć udostępniać tego, co się dzieje w okolicy jego komputera. Fakt oczekiwania na akceptację lub rezygnację wspomnianego już komunikatu nie powinien jednak blokować wykonywania skryptu. Jak widać, użycie przez twórców Web Audio API obiektu typu `promise` jest tutaj jak najbardziej uzasadnione.

Jeżeli już mówimy o względach bezpieczeństwa, warto wspomnieć, że dostęp do strumieni zwracanych przez metodę `getUserMedia` nie jest już teraz tak łatwy jak to miało miejsce przed końcem 2015 roku. Wtedy właśnie w przeglądarce Chrome wprowadzono zmianę, która do poprawnego przechwytywania strumienia dźwięku lub/i obrazu wymaga dodatkowo użycia protokołu `https`. Nieszyfrowany protokół `http` zadziała tylko w przypadku korzystania z lokalnego serwera (127.0.0.1, localhost). Jest bardzo prawdopodobne, że niebawem wszystkie przeglądarki będą zachowywać się tak samo jak Chrome. W przeszłości nie było takich restrykcji. Nie zdziwi się więc, gdy natrafimy w Internecie na „martwe” przykłady użycia Web Audio API. Nie będą one działać w większości przypadków tylko dlatego, że użyto protokołu `http` zamiast `https`.

Wróćmy do naszego obiektu `promise` zwracanego przez metodę `getUserMedia`. Po jego pozytywnym rozwiązaniu otrzymujemy strumień `audio` (WebRTC `MediaStream`). To właśnie jego musimy podać jako parametr metody `createMediaStreamSource`. W skrypcie z Listingu 1 wszystkie opisane wyżej kroki umieszczone w metodzie `connectMicrophoneTo`. Przyjmuje ona jeden parametr będący węzłem `audio`. Właśnie do tego węzła dołączany jest asynchronicznie strumień `audio` z mikrofonu. W naszym przykładzie tym parametrem jest obiekt typu `GainNode` przypisany do zmiennej `microphoneVirtual`.

Na koniec informacja, która może oszczęścić czas i nerwy. Ni-gdy nie deklarujemy zmiennej przetrzymującej węzeł stworzony przez `createMediaStreamSource` jako lokalnej w funkcji będącej handlerem `then`. W zależności od implementacji silnika JS w przeglądarce może ona zostać usunięta przez *Garbage Collector* po kilku sekundach od inicjalizacji. Gdy tak się stanie, nasz obiekt typu `MediaStreamSourceNode` po prostu zniknie i pozostajemy z głuchą ciszą. Nie pomaga tutaj nawet fakt, że podłączamy go do zmiennej `microphoneVirtual` poprzez metodę `connect`. W teorii GC nie powinien go wtedy usuwać, jednak w praktyce okazuje się, że nie zawsze tak jest. Rozwiążaniem jest po prostu zadeklarowanie tej zmiennej w takim zasięgu, który nie zostanie usunięty po opuszczeniu funkcji umieszczonej w `then`.

## 5. WĘZŁY UMOŻLIWIJĄCE PRZETWARZANIE DANYCH Z MIKROFONU

W tej sekcji dowiemy się, w jaki sposób możemy wykorzystać strumień `audio` z naszego mikrofonu. W przykładzie z Listingu 1 przygotowaliśmy już nieco potrzebne klocki do pracy. Użyliśmy do tego celu węzła `GainNode` (zmienna `microphoneVirtual`) oraz asynchronicznie przypinanego do niego węzła `MediaStreamSourceNode` (zmienna `microphone`). Takie połączenie umożliwiło już na etapie inicjalizacji aplikacji podpięcie zmiennej `microphoneVirtual` do dalszej części naszego grafu. Nie było więc konieczne czekanie na prawdziwe dane z mikrofonu.

Co może pełnić rolę dalszej części grafu? Może być to np. węzeł `destination`. Do zastosowań transmisji danych podłączanie mikrofonu do głośników nie ma jednak większego sensu. Jedyne, co potrzebujemy, to przetwarzanie napływających próbek audio w taki sposób, by wyciągnąć z nich strumień symboli nadawanych przez inne urządzenie. Aby tego dokonać, mamy do wyboru dwie drogi.

Pierwszą jest użycie węzła `AnalyserNode`. Jest to rozwiązańe, które wcześniej nazwaliśmy pewną formą oszukiwania, gdyż wprowadza „magiczne pudełko”, które wykonuje za nas operacje DSP. Jak już jednak wspomnieliśmy, jest to jedyne rozwiązanie, jeśli chodzi o wolniejsze urządzenia mobilne, takie jak np. smartfony. Węzeł ten używa niezwykle wydajnego algorytmu FFT (ang. *Fast Fourier Transform*), więc jest w tym przypadku idealny.

Druga droga to praca na surowych samplach audio, które będziemy musieli sami przetworzyć. Oznacza to, że będziemy musieli zapragnąć do pracy algorytm DTF opisany w pierwszej części tego artykułu. Jego zasadniczą wadą jest bardzo powolne działanie, jednak jest on relatywnie prosty do zrozumienia. Do pracy wymaga na wejściu tablicy z wartościami kolejnych próbek. Nasz strumień z mikrofonu musi zatem zostać w jakiś sposób zamieniony na tablice JavaScriptowe. Z pomocą przychodzi `ScriptProcessorNode`. Jest to węzeł, który paczuje napływający strumień próbek do postaci tablic o różnych rozmiarach. Aby uzyskać dostęp do tych danych, wystarczy zarejestrować w węźle handler do eventu `onaudioprocess`. Zdarzenie to będzie cyklicznie wywoływanie w stałych odstępach czasu.

W naszej klasie `AudioMonoIO` zaimplementujemy możliwość skorzystania z obydwu dróg. Prześledźmy zatem szczegółowo obydwa węzły.

## 5.1 AnalyserNode

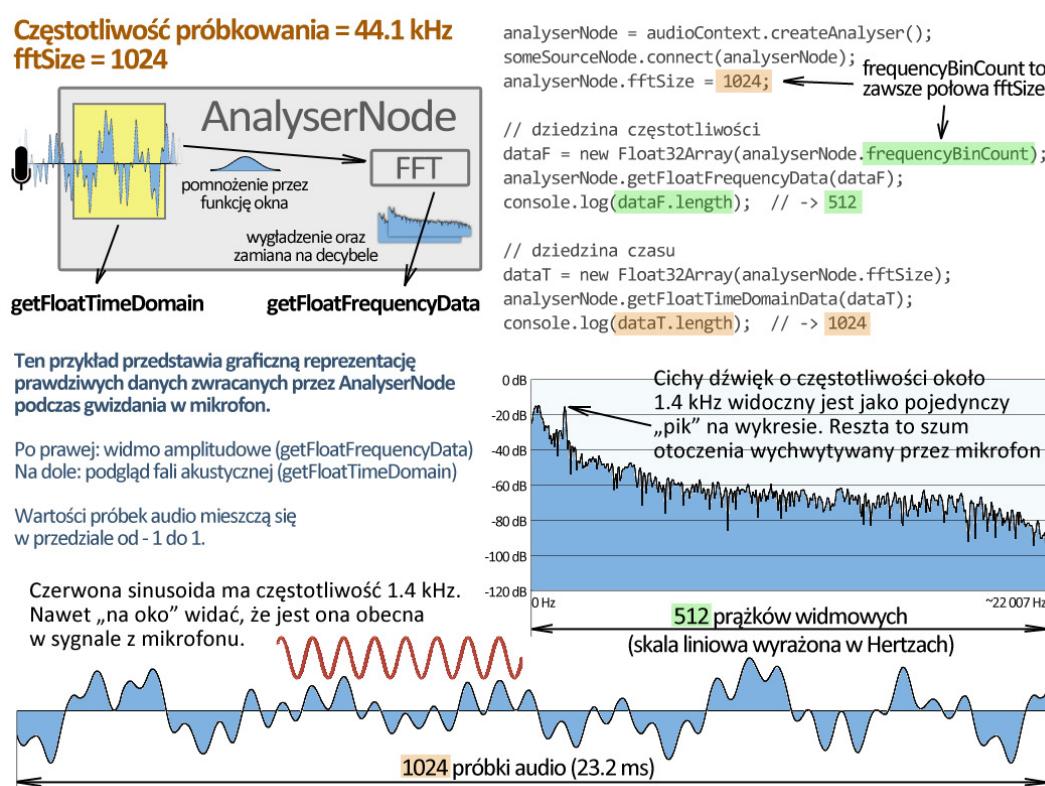
Węzeł ten tworzymy jak każdy inny węzeł za pomocą odpowiedniej metody kontekstu audio. W tym przypadku ta metoda to `create-`

`Analyser`. W skrócie `AnalyserNode` umożliwia podgląd w czasie rzeczywistym danych zarówno z dziedziny czasu, jak i z dziedziny częstotliwości. Dostęp do tych informacji możliwy jest za pomocą dwóch metod: `getFloatTimeDomainData` oraz `getFloatFrequencyData`. Obie z nich wymagają jednego parametru, który jest tablicą typu `Float32Array` o odpowiednim rozmiarze. Gdy wywołamy obydwie metody zaraz po sobie, otrzymane wyniki możemy traktować jako parę. W środku naszego węzła tablica próbek dziedziny czasu jest po prostu użyta jako wejście algorytmu obliczającego tablicę dziedziny częstotliwości.

Tak naprawdę istnieją jeszcze dwa odpowiedniki wspomnianych metod. Są nimi `getByteTimeDomainData` oraz `getByteFrequencyData`. Pod względem wydajnościowym nie ma jednak znaczenia, czy użyjemy wersji `Float` czy `Byte`. Skoro nie widać różnic, to... wybieramy wersję `Float`, gdyż oferuje większą precyzję.

Twórcy Web Audio API skorzystali z faktu, iż tablice podane jako parametr funkcji przekazywane są przez referencję. Dane wpisywane są więc bezpośrednio do przekazanych w parametrze tablic. By nie zgubić żadnych danych, tablice te muszą mieć ustalony z góry rozmiar. Zależy on ściśle od wartości parametru `fftSize`, o którym opowiem szczególnego za chwilę. W przypadku metody `getFloatTimeDomainData` rozmiar ten powinien wynosić dokładnie `fftSize`, natomiast dla `getFloatFrequencyData` musi być to połowa `fftSize`. Dla uproszczenia wartości połowy `fftSize` możemy pobrać bezpośrednio z instancji węzła za pomocą właściwości `frequencyBinCount`.

Czym jest więc `fftSize`? Jest to liczba sampli audio, które zostaną użyte do wykonania na nich algorytmu Szybkiej Transformaty Fouriera (FFT). Innymi słowy jest to szerokość okna, przez które podglądany jest napływający strumień próbek audio w dziedzinie czasu. Domyslnie szerokość ta jest równa 2048. Można ją jednak zmienić, przypisując nową wartość do właściwości `fftSize` naszego obiektu węzła. Rozmiar ten zawsze musi być liczbą będącą



Rysunek 7. Interpretacja wyników pracy `AnalyserNode`

potęgą dwójki z zakresu od 32 do 32768. Mamy zatem do dyspozycji 11 różnych wartości: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768. Na Rysunku 7 pokazano interpretację wyników pracy AnalyserNode dla fftSize równego 1024.

Używając elementu <canvas> oraz danych zwracanych przez AnalyserNode, możemy w łatwy sposób przedstawić graficznie falę akustyczną oraz jej widmo amplitudowe. Otrzymany wykres widma będzie jednak nieco inny niż ten, który analizowaliśmy w poprzedniej części artykułu. Różnica wynika z użytej na osi poziomej jednostki częstotliwości. W AnalyserNode jednostką jest Hertz, podczas gdy w naszej implementacji DTF użyliśmy dość nietypowo samplePerPeriod. Warto jednak zaznaczyć, że w obydwu przypadkach skala na osi poziomej jest liniowa. Użycie jednostki samplePerPeriod spowodowało zniekształcenie „górek” reprezentujących nasze dominujące częstotliwości. Wykres był zwyczajnie rozciągnięty z lewej strony, a ścisły z prawej. W przypadku AnalyserNode taki efekt nie występuje. Każda pojedyncza sinusoida o danej częstotliwości będzie zatem na wykresie „pikiem” o podobnej szerokości.

Kod oraz źródła przykładu używającego elementu <canvas> znajdziemy poniżej:

- » <https://audio-network.rypula.pl/analyser-node>
- » <https://audio-network.rypula.pl/analyser-node-src>

### 5.1.1 AnalyserNode pod lupą – algorytm FFT

W tej sekcji przyjrzymy się z grubsza działaniu FFT. Jego najbardziej znaną formą jest algorytm Cooley-Tukey pracujący na tablicach o rozmiarach będących potęgą dwójki. To właśnie dzięki niemu AnalyserNode jest w stanie zwracać dane z dziedziny częstotliwości niezwykle szybko. Nie będziemy jednak zagłębiać się w szczegóły jego implementacji, gdyż wykracza to poza zakres tego artykułu. FFT potraktujemy zatem jak czarne pudełko z wejściem i wyjściem, do którego wchodzą i wychodzą dane określonego typu. Warto tutaj jednak zaznaczyć ważną rzecz – AnalyserNode nie udostępnia żadnych metod do bezpośredniego korzystania z FFT. Wszystko dzieje się automatycznie podczas pracy węzła. O FFT mówimy tylko dlatego, by przybliżyć zasadę działania klasy AnalyserNode.

Zanim aktualny blok próbek audio z dziedziny czasu trafi na wejście FFT AnalyserNode stosuje na nim funkcję okna. Ten zbieg ma na celu wyeliminowanie efektu wycieku widma. Omawialiśmy to zagadnienie w poprzedniej części artykułu. Wewnątrz algorytmu FFT tak przygotowany blok wejściowy poddawany jest m.in. podziałowi na mniejsze bloki w myśl zasady „dziel i zwyciężaj”. Finalnie na wyjściu FFT otrzymujemy blok liczb zespolonych. W każdej z nich zawarta jest informacja o określonej częstotliwości znajdującej się w badanym sygnale. Powiązane z otrzymanymi liczbami zespolonymi częstotliwości zależą jednak ściśle od wybranego na początku rozmiaru fftSize. Jest to nieco odmienne działanie niż w przypadku algorytmu z pierwszej części tego artykułu. W jego przypadku możliwe było wybranie dowolnej częstotliwości, na którą chcemy się „nastroić”. Ważne jest jednak, że w przypadku obydwu algorytmów interpretacja otrzymanych liczb zespolonych jest taka sama.

Przypomnijmy: z tak otrzymanych liczb zespolonych możemy wyciągnąć informacje o przesunięciach fazowych oraz amplitudach fal z nimi związanych. Niestety w przypadku AnalyserNode w momencie wywołania metody `getFloatFrequencyData` lub `getByteFrequencyData` ta pełna informacja jest spłaszczona do postaci jedynie danych o amplitudzie. Innymi słowy, to, co otrzymujemy, to jedynie wartości bezwzględne liczb zespolonych

znajdujących się w „najświeższym” buforze wyników FFT. Niestety twórcy Web Audio API nie przewidzieli żadnej innej metody do otrzymania informacji o przesunięciu fazowym. Dlaczego? Niestety nie znalazłem nigdzie informacji na ten temat...

Wróćmy do rozmiarów tablic. Faktem jest, że w wyniku działania FFT otrzymamy na wyjściu tyle samo liczb zespolonych co wartości podanych na wejściu (fftSize). Dlaczego zatem AnalyserNode zwraca nam tylko połowę? Aby rozwiązać tę zagadkę, wystarczy spojrzeć, jak rozlokowane są częstotliwości powiązane z każdą liczbą zespoloną w tablicy otrzymanej na wyjściu FFT. Pod indeksem 0 znajdziemy częstotliwość 0 Hz. Część rzeczywista tej liczby zespolonej jest to tak zwany DC-offset, który jest niczym innym jak wartością średnią próbek z okna. Wspomnieliśmy o nim krótko przy okazji omawiania OscillatorNode i PeriodicWave. Jak zapewne podpowiada nam intuicja, idąc dalej po indeksach tablicy, każda następna liczba zespolona będzie powiązana z coraz to wyższą częstotliwością (indeksy 1, 2, 3, ...). Jest to prawda. Okazuje się jednak, że najwyższa użyteczna wartość częstotliwości przypadnie nie na samym końcu tablicy (indeks fftSize - 1), lecz w jej środku (indeks 0.5 \* fftSize). Rozmiar tablicy to liczba parzysta, więc dla ściśleści powinniśmy bardziej powiedzieć: w pierwszym możliwym indeksie drugiej połowy, patrząc od lewej. Nasz „środkowy” indeks będzie powiązany z częstotliwością Nyquista.

Faktem jest, że przekroczenie częstotliwości Nyquista powoduje efekt nazwany aliasingiem. Wtedy w zapisanym sygnale pojawią się częstotliwości, których nie ma w rzeczywistości. Najprostszą analogią jest przypadek filmowania kół samochodu, który rusza z miejsca, a następnie porusza się ruchem jednostajnie przyspieszonym. Po odtworzeniu filmu zobaczymy, że w miarę upływu czasu koła kręci się coraz szybciej. Gdy jednak częstotliwość obrotu kół przekroczy połowę częstotliwości, z jaką nasza kamera zapisuje klatki obrazu, koła samochodu zaczną obracać się w przeciwnym kierunku. Dodatkowo ich prędkość obrotu będzie stopniowo spaść (Rysunek 1). Gdy częstotliwość obrotu kół będzie równa częstotliwości zapisywania klatek obrazu, wtedy na filmie ujrzymy jadący samochód, którego koła nie obracają się wcale. Dalsze zwiększenie prędkości samochodu spowoduje cyklicznie pojawianie się na filmie tego samego efektu (ang. *wagon-wheel effect*). Okazuje się, że Dyskretna Transformata Fouriera ma analogiczne właściwości związane z częstotliwością obrotu wektora jednostkowego, który omawialiśmy w poprzedniej części artykułu.

Wróćmy do FFT. Ustaliliśmy, że każdy kolejny prążek powiązany jest z coraz to wyższą częstotliwością. Efekt aliasingu spowoduje jednak, że w drugiej połowie tak naprawdę zobaczymy widmo częstotliwości o wartościach ujemnych. Dla uproszczenia założymy, że rozpatrujemy wartości bezwzględne częstotliwości ujemnych powiązanych z indeksami. Przy takim założeniu każda następna liczba zespolona o indeksie większym od 0.5 \* fftSize będzie już powiązana z coraz to niższą częstotliwością. Wartości częstotliwości za połową będą zatem spadać tak samo szybko jak rosły przed połową. W pierwszej połowie częstotliwości rosną liniowo, natomiast w drugiej połowie liniowo maleją. Liczba zespolona pod indeksem 0.5 \* fftSize - x będzie zatem w pewnym sensie powiązana z taką samą częstotliwością co liczba zespolona pod indeksem 0.5 \* fftSize + x. Widzimy zatem, że mamy tu do czynienia z symetrią częstotliwości względem indeksu 0.5 \* fftSize. Przesuwając się cały czas w prawo, dotrzemy finalnie do końca tablicy. Ostatni indeks tablicy (fftSize - 1) będzie powiązany z taką samą częstotliwością jak indeks 1. Jak widzimy, tablica „skończyła się”, zanim dotarliśmy do elementu DC-Offset, który jest pod indeksem 0.

Pamiętajmy jednak, że do tej pory omówiliśmy symetrię częstotliwości powiązanych z elementami tablicy. O samych wartościach nic jeszcze nie powiedzieliśmy. O tym, czy one także wykazują pewną formę symetrii, opowiemu nieco dalej.

Wróćmy na chwilę do typu danych, jakie możemy podać na wejściu FFT. Ustaliliśmy, że AnalyserNode na wejściu otrzymuje wartości sampli audio z naszego okna. Jak na próbki audio przystało, są one liczbami rzeczywistymi. Jak się jednak okazuje, algorytm FFT na wejściu z powodzeniem przyjmuje także liczby zespolone.

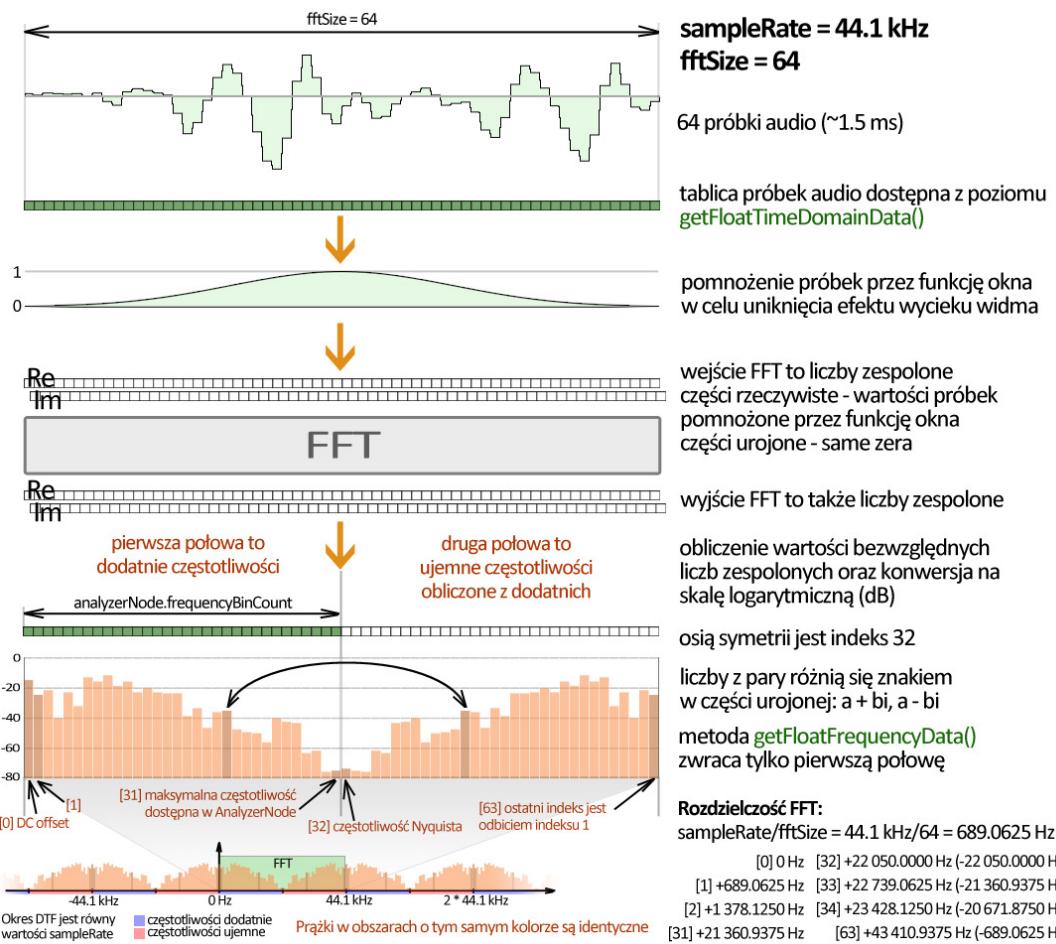
Chwileczkę... liczby zespolone na wejściu w dziedzinie czasu? Przecież wartość próbki audio to zapis binarny napięcia wytwarzanego przez wychylenie membrany mikrofonu! Okazuje się jednak, że w przetwarzaniu sygnałów liczby zespolone w dziedzinie czasu nie są niczym dziwnym. Sam byłem tym dość zdziwiony, dlatego że jest to trochę sprzeczne z intuicją. Jak bowiem wyobrazić sobie ujęte wychylenie membrany mikrofonu? Nie będziemy tutaj jednak zastanawiać się, jaki jest tego fizyczny sens. Zastanowimy się bardziej, jak ten „problem” obejść. Otóż rozwiązaniem jest stworzenie takiej liczby zespolonej, której część rzeczywista będzie równa wartości naszej próbki, a część urojona będzie równa zero. Problem rozwiązany! Dodatkowo właśnie tutaj zaczyna się finalna odpowiedź na pytanie, dlatego otrzymujemy tylko połowę fftSize. Okazuje się, że gdy części ujęte w wszystkich elementów wejściowych są równe zero, to po transformacie moduły otrzymanych liczb zespolonych będą symetrycznie odbite względem indeksu  $0.5 * fftSize$ . Dla dociekliwych – liczby zespolone z „parą” będą miały zwyczajnie przeciwny znak w części ujętej, np.  $a + bi$  oraz  $a - bi$ . Zmiana znaku nie zmienia długości

wektora, więc moduły są takie same. Zaznaczmy jednak raz jeszcze: jest tak tylko wtedy, gdy liczby zespolone z wejścia FFT mają w częściach ujętych same zera.

Skoro zatem moduły liczb zespolonych drugiej połowy są jedynie odbiciem modułów pierwszej połowy, oznacza to, że z powodzeniem można któryś z nich pominąć. Tak właśnie zrobiono w AnalyserNode. Dane zwarcane przez metodę `getFloatFrequencyData` są tak naprawdę pierwszą połową bloku zwróconego przez FFT. Do naszej dyspozycji jest zatem tablica o rozmiarze  $0.5 * fftSize$  od indeksu 0 (DC-Offset) do indeksu  $0.5 * fftSize - 1$  włącznie. Jak widzimy, dane o częstotliwości Nyquista (indeks  $0.5 * fftSize$ ) nie zawierają się w tym zbiorze, więc są dla nas niedostępne (Rysunek 8).

Skoro już wiemy, jaki jest sens danych zwartych przez metodę `getFloatFrequencyData`, zastanówmy się teraz, jaki jest skok częstotliwości między dwoma sąsiadującymi indeksami. Skoro indeks  $0.5 * fftSize$  to częstotliwość Nyquista (czyli połowa częstotliwości próbkowania) oraz potrzeba  $0.5 * fftSize$  elementów, by dotrzeć tam liniowo od 0 Hz, to możemy zapisać to wzorem:  $(sampleRate / 2) / (fftSize / 2)$ . Po uproszczeniu otrzymujemy  $sampleRate / fftSize$ . Podstawiając wartości, czyli np. próbkowanie 44.1 kHz oraz rozmiar  $fftSize$  równy np. 2048, otrzymamy wartość około 21.53 Hz. Ten wynik mówi nam o rozdzielczości częstotliwościowej FFT.

Czy to dużo? Oczywiście to zależy. Rozważmy więc najprostszy przykład z życia wzięty, czyli gwizdanie. W przybliżeniu możemy założyć, że częstotliwość gwizdania mieści się w przedziale od 1 kHz do 2 kHz. W tym zakresie odległość między kolejnymi często-



Rysunek 8. Symetria wyjścia FFT dla danych wejściowych będących liczbami rzeczywistymi

tliwościami nut wynoszą mniej więcej od 50 do 100 Hz. Widzimy zatem, że otrzymana rozdzielcość jest wystarczająco duża, by rozróżnić pojedyncze dźwięki osoby gwiżdżącej, np. popularne „do re mi fa sol la si do”. Są to kolejno zagrane białe klawisze w oktawie. Na Rysunku 9 do zapisu nazw dźwięków użyto metody znanej jako „Scientific Pitch Notation”. Różni się ona od zapisu używanego w Polsce, lecz moim zdaniem ten zapis jest bardziej logiczny i krótszy. Częstotliwości dźwięków wygenerowane, bazując na powszechnie stosowanym strojeniu równomiernie temperowanym o 12 dźwiękach na oktawę. Częstotliwość dźwięku A4 to 440 Hz.

### 5.1.2 Podgląd w czasie rzeczywistym

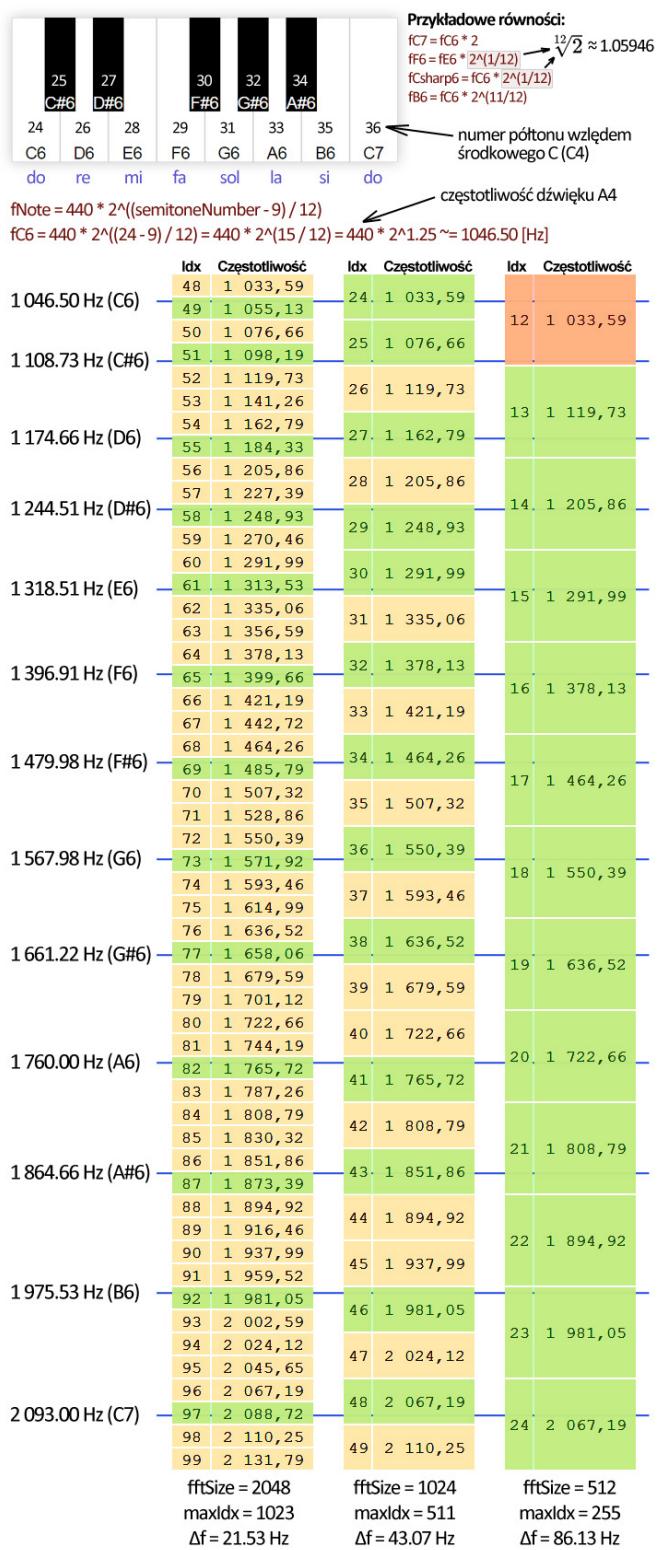
Na koniec zastanówmy, co tak naprawdę oznacza, że AnalyserNode umożliwia podgląd naszego sygnału w czasie rzeczywistym. Wiemy już, że algorytm FFT w nim użyty działa na bloku danych wejściowych o określonym rozmiarze. Zastanówmy się zatem, ile czasu taki blok trwa. Najmniejsza możliwa wartość fftSize to 32. Przy próbkowaniu 44.1 kHz taki pojedynczy blok trwa około 0.7 ms. Dla kontrastu największy możliwy blok ma długość 32768 sampli. Jego czas trwania jest już znacznie dłuższy, gdyż wynosi około 743 ms.

Jak to się przekłada na wychwytywanie sygnałów? W praktyce w dłuższych blokach krótko trwające sygnały będą wydawały się słabsze. Tym słabsze, im mniejszy procent okna będą wypełniać. Z kolei „czas reakcji” wykresu widma na pojawienie się sygnału będzie dłuższy.

Mając na myśli rozmiar bloku, chodzi nam tu o wielkość okna, przez które patrzymy na nasz sygnał w dziedzinie czasu. Do uzyskania wrażenia płynnego podglądu w czasie rzeczywistym okno to musi się w jakiś sposób przesuwać w miarę napływu nowych sampli audio. Powstanie zatem coś w rodzaju filmu, w którym każda klatka generowana jest na podstawie aktualnej ramki z wynikiem FFT. Aby zapewnić płynne przejście pomiędzy klatkami, w AnalyserNode dostępna jest opcja wygładzania dziedziny częstotliwości. Jest ona domyślnie włączona. W skrócie działa to tak, że elementy bloku danych zwracanego przez metodę getFloatFrequencyData wyliczane są jako średnia ważona elementów z bloku poprzedniego i aktualnego. Wygładzaniem tym możemy sterować, ustawiając wartość z przedziału od 0 do 1 do właściwości smoothingTimeConstant w instancji klasy AnalyserNode (0 oznacza brak wygładzania, 1 największe wygładzanie). Wartością domyślną jest 0.8.

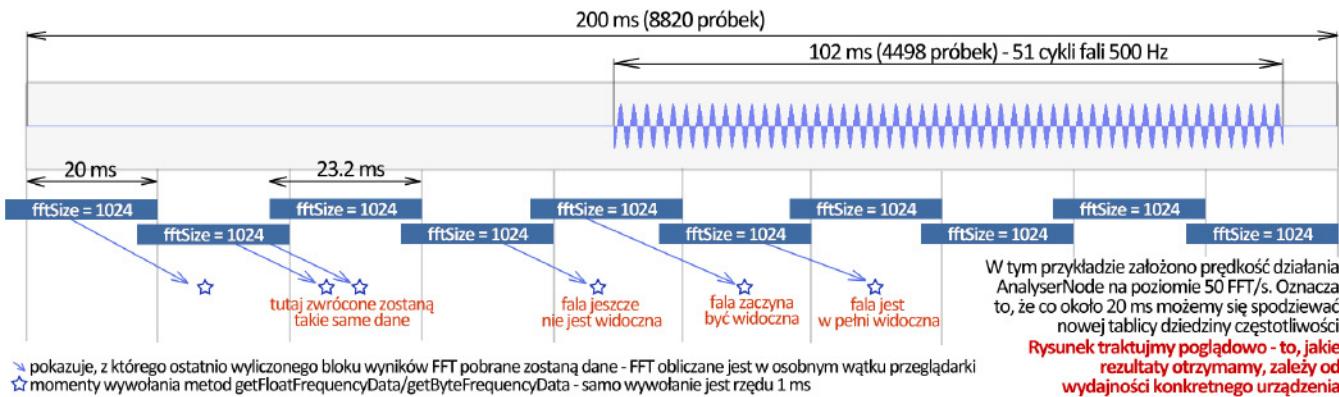
Wygładzanie niestety niesie za sobą pewne konsekwencje, jeśli chodzi o transmisję danych. Przypuśćmy, że nadajemy wiadomość Morsem na jednej stałej częstotliwości. Okresy ciszy rozdzielają kolejne symbole dłuższych i krótszych impulsów. W szczególnym przypadku stosując wygładzanie, spowodujemy zlanie się wszystkich symboli w jeden. Nasze przerwy nie byłyby zwyczajnie widoczne jako wyraźne spadki amplitudy prążka widmowego powiązanego z naszą częstotliwością nośną. Na potrzeby transmisji danych generalnie lepiej jest całkowicie wyłączyć wygładzanie. Możemy tego dokonać, przypisując 0 do właściwości smoothingTimeConstant.

Na koniec warto zastanówć się, jaki jest limit odświeżania podglądu w „czasie rzeczywistym”. Podsumujmy – by nie „zgubić” żadnej części sygnału i uzyskać zadowalającą szybkość odświeżania, następujące po sobie okna powinny zachodzić „na zakładkę”, a przesunięcia powinny być możliwie minimalne. Czy zatem możliwe jest tak częste odświeżanie danych zwracanych przez AnalyserNode, by kolejno występujące po sobie okna oddalone były od siebie o wartość jednej próbki? Otóż nie. W Web Audio API zdefiniow-



Rysunek 9. Rozdzielcość FFT a zdolność wykrywania dźwięków podczas gwizdzienia

wano pojęcie o nazwie „render quantum”. Jest to z góry określona liczba sampli, którą jednorazowo, w sposób atomowy, przetwarza kontekst audio. Dokumentacja Web Audio API definiuje tę wartość jako równą 128. To ograniczenie skutkuje tym, że nie jest możliwe odświeżenie np. danych zwracanych przez getFloatFrequencyData częściej niż co 128 sampli (około 2.9 ms). Po prostu wywołania metod zawierające się w tym samym „kwancie renderowania” będą zawsze zwracały te same dane. Stanie się tak również wtedy, gdy będziemy starać się odczytywać dane szybciej niż nasz sprzęt



Rysunek 10. W jaki sposób AnalyserNode dostarcza podglądu w czasie rzeczywistym

jest w stanie je przetwarzać. Tutaj warto jednak zaznaczyć, że algorytm FFT działa w osobnym wątku przeglądarki i nie powoduje „prycinania” strony (Rysunek 10).

### 5.1.3 Wydajność FFT w porównaniu z metodą intuicyjną DTF

Na końcu przeanalizujmy wydajność użytego w AnalyserNode algorytmu FFT. Jego złożoność wynosi  $O(N * \log_2(N))$ , podczas gdy złożoność metody opisanej w poprzedniej części artykułu to  $O(N^2)$ . Taki zapis może na początku zbyt wiele nie mówić, lecz gdy podstawimy do wzorów liczby szybko, zobaczymy, jak duża jest ta różnica. Dla fftSize o wartości 4096 będzie to odpowiednio 49152 ( $4096 * \log_2(4096)$ ) oraz 16777216 ( $4096 * 4096$ ). Algorytm FFT będzie zatem szybszy około 341x. Idąc dalej, dla fftSize o wartości 32768 będzie to już odpowiednio 491520 ( $32768 * \log_2(32768)$ ) oraz 1073741824 ( $32768 * 32768$ ). Dla tego przypadku FFT będzie już szybszy około 2185x. Im większa wartość N, tym większe stają się różnice wydajności na korzyść FFT. Założymy, że mamy maszynę zdolną wykonać milion rozważanych operacji w ciągu sekundy. W pierwszym przykładzie ( $N = 4096$ ) obliczenia będą trwać około 50 milisekund dla FFT oraz prawie 17 sekund dla metody standardej. W drugim przykładzie ( $N = 32768$ ) będzie to już około 500 ms dla FFT oraz prawie 18 minut dla metody standardej. Widoczne wyraźnie, że jedną opcją dla generowania widma sygnału w czasie rzeczywistym jest FFT. Nie bez powodu ten algorytm często nazywa się jednym z najważniejszych algorytmów w historii. To właśnie jego szybkość sprawiła, że dzisiaj możemy się cieszyć technologiami takimi jak Wi-Fi, LTE, DVB-T. Nawet leciwy już format MP3 używa FFT. Oczywiście praktycznych zastosowań tej metody jest o wiele więcej.

### 5.1.4 AnalyserNode – podsumowanie

W tej sekcji przeanalizowaliśmy dość szczegółowo AnalyserNode. Po krótkiej analizie wydajnościowej można stwierdzić, że bardzo trudno jest zrezygnować z rozwiązań szybkich na rzecz tylko rozwiązań prostych i intuicyjnych zaimplementowanych przez nas samodzielnie. Nie chodzi już tutaj tylko o wolniejsze urządzenia mobilne. Przy tworzeniu aplikacji transmitującej dane dość często potrzebne będzie użycie widma sygnału zwyczajnie po to, by zweryfikować okolice częstotliwości, na którą staramy się nastroić. Dlatego też w naszej klasie AudioMonoIO udostępnimy także potencjał, jaki oferuje FFT. Pamiętajmy jednak, że gdy tylko interesuje nasz pojedynczy prążek, wciąż możemy skorzystać z „naszego” intuicyjnego algorytmu.

Ważna uwaga co do samego algorytmu FFT. Otóż nie zwraca on wyników będących pewnego rodzaju przybliżeniem Dyskretnej Transformaty Fouriera. Jest to algorytm dający dokładnie te

same wyniki, tylko w nieporównywalnie krótszym czasie. Niestety AnalyserNode „spłaszcza” je do postaci jedynie wartości bezwzględnych liczb zespolonych. Oznacza to, że informacja o przesunięciu fazowym jest dla nas niedostępna. Ciągle jednak jesteśmy w stanie narysować np. wykres widma amplitudowego, co w większości przypadków jest wystarczające.

## 5.2 ScriptProcessorNode

Węzeł ten umożliwia dostęp do pojedynczych próbek przechodzących przez nasz graf stworzony w kontekście audio. Jego działanie i konfiguracja jest bardzo prosta. Wystarczy stworzyć nowy obiekt za pomocą metody `createScriptProcessor` znajdującej się w kontekście audio. Przyjmuje ona 3 parametry. Pierwszy to rozmiar paczki z sampłami, jaką będziemy cyklicznie przetwarzając i/lub generując podczas pracy naszej aplikacji. Wartość tego parametru musi być liczbą będącą potęgą dwójki z przedziału od 256 do 16384. Mamy zatem do dyspozycji 7 możliwości – 256, 512, 1024, 2048, 4096, 8192 oraz 16384. Mały rozmiar bufora umożliwia uzyskanie mniejszych opóźnień, jednak może powodować powstawanie niepożądanych trzasków. Duży rozmiar bufora powinien wyeliminować te problemy jednak wtedy musimy liczyć się z większymi opóźnieniami. Istnieje także ósma możliwość, czyli podanie zera. Wtedy Web Audio API podczas tworzenia węzła sam dobrze najbardziej odpowiedni rozmiar bufora, biorąc pod uwagę wydajność środowiska, na jakim pracuje.

Drugi i trzeci parametr mówi o liczbie kanałów kolejno dla wejścia i wyjścia. Do naszych zastosowań wystarczy dźwięk „mono”, więc ta wartość powinna wynosić jeden lub zero. Dlaczego zero? Ponieważ gdy będziemy potrzebować węzła pełniącego rolę tylko generatora próbek, interesować nas będzie jedynie wyjście. Wtedy liczba kanałów wejścia może być równa zero. Czy zatem w sytuacji odwrotnej, gdy interesują nas tylko napływające próbki, możemy postąpić analogicznie (1 kanał wejścia, 0 kanałów wyjścia)? Wasadzie tak, ale nie do końca. Otóż w przeglądarce Chrome od dawna kilkunastu miesięcy występuje pewien błąd. Gdy do węzła `ScriptProcessorNode` podłączymy wejście, ale nie podłączymy wyjścia, wtedy nasz węzeł nie będzie działać prawidłowo. Nie pomaga nawet fakt podania zera jako liczby kanałów wyjścia. Po prostu zdarzenie z paczką do przetworzenia nie będzie uruchamiane. Rozwiązaniem jest podłączenie wyjścia naszego script procesora np. do węzła `destination`. Tablica próbek wyjścia jest domyślnie wypełniona zerami, więc naszego węzła nie będzie „słuchać”. Chcąc zastosować sztuczkę z węzłem `destination`, liczbę kanałów wyjścia musimy ustawić na 1 (mono).

W zasadzie jednak ten problem nas nie dotyczy, gdyż wcale nie musimy tworzyć osobnych węzłów `ScriptProcessorNode` dla sampli napływających i sampli generowanych. Możemy zwykle utworzyć jeden wspólny obiekt `ScriptProcessorNode`. W zdarzeniu, które jest przez niego odpalone, mamy dostęp zarówno do tablicy wejścia, jak i wyjścia.

Jak podpiąć się do tego zdarzenia? Wystarczy do właściwości `onaudioprocess` przypisać zwykłą funkcję. Będzie do niej przekazywany jeden parametr `audioProcessingEvent`, z którego możemy wydobyć wszystko, czego potrzebujemy (Listing 4).

#### Listing 4. `ScriptProcessorNode` – tablice z samplami

```
spNode = audioContext.createScriptProcessor(4096, 1, 1);
spNode.onaudioprocess = function (audioProcessingEvent) {
    var monoIn, monoOut;

    // potrzebujemy tylko kanału o indeksie zero (mono)
    monoIn = audioProcessingEvent.inputBuffer
        .getChannelData(0),
    monoOut = audioProcessingEvent.outputBuffer
        .getChannelData(0);

    sampleInHandler(monoIn);
    sampleOutHandler(monoOut, monoIn);
}

function sampleInHandler(monoIn) {
    console.log(monoIn.length); // -> 4096
}

function sampleOutHandler(monoOut, monoIn) {
    console.log(monoOut.length, monoIn.length);
    // -> 4096 4096
}
```

W przykładzie z Listingu 4 przetwarzanie i generowanie sampli odleutowaliśmy do osobnych funkcji, używając przy tym jednego węzła. Jego rozmiar bufora ustawiliśmy na 4096. Oznacza to, że każda z przekazywanych tablic będzie także tego rozmiaru. Funkcja `sampleInHandler` otrzyma zatem jeden parametr będący tablicą 4096 próbek wejścia np. z mikrofonu. Jest to w tym przypadku fragment o długości około 93 ms, zakładając standardowe próbkowanie (1000 \* 4096 / 44100). Z tej tablicy będziemy jedynie czytać. W przypadku funkcji `sampleOutHandler` będą to dwa parametry – `monoOut` oraz `monoIn`. Zmienna `monoOut` to także fragment o długości około 93 ms, jednak tym razem strumienia audio, który zostanie odegrany np. na naszych głośnikach. Do tej tablicy będziemy więc tylko pisać. Gdy jednak zależy nam, by nasz węzeł „generował” ciszę, wystarczy nie pisać wcale. Jak już wspomnieliśmy, domyślnie nasza tablica `monoOut` jest wypełniona zerami. Drugi parametr `monoIn` możemy traktować jako opcjonalny. Możemy dzięki niemu generować dane wyjściowe na podstawie danych wejściowych.

Możemy teraz zapytać – po co w ogóle używać `ScriptProcessorNode` do przetwarzania napływających sampli? Przecież `AnalyserNode` ma metodę `getFloatTimeDomainData`. Ona także zwraca sampl z wejścia węzła. Generalnie tak, jednak w przypadku `AnalyserNode` nie bardzo mamy kontrolę nad tym, jaki konkretnie blok dostaniemy. Oczywiście możemy próbować „celować” w odpowiednie miejsca w czasie np. timerem. Nigdy jednak nie będziemy mieć pewności, czy występujące

po sobie bloki nie będą zachodzić na siebie lub czy nie będzie między nimi małej przerwy. Ze `ScriptProcessorNode` jest inaczej. O ile nasz procesor będzie nadawał z przetwarzaniem, będą to zawsze bloki występujące bezpośrednio po sobie co do sampla. Zachowana jest zatem ciągłość napływających próbek i każda z nich wystąpi tylko raz.

`ScriptProcessorNode` ma jednak jedną dużą wadę. Otóż działa on w głównym wątku JavaScriptowym. Wątek ten odpowiadający jest także za interakcję z użytkownikiem. Możemy się o tym łatwo przekonać, uruchamiając `while (true)` w skrypcie bezpośrednio dołączonym do HTMLa. Nasza zakładka po prostu przestanie odpowiadać. Gdy zatem przetwarzanie naszych sampli będzie trwało zbyt długo, możemy spowodować nie tylko traski w audio, ale też „przywieszenie” naszej strony. Rozwiążaniem byłoby tu przeniesienie tych operacji do osobnego wątku. Okazuje się, że Web Audio API dokładnie do tego zmierza. W dokumentacji `ScriptProcessorNode` od dawna oznaczony jest jako *deprecated*. Zostanie on zastąpiony czymś w rodzaju audio Web Workerów. Na razie jednak nowe rozwiązania nie są dostępne. Warto jednak o tym pamiętać, by w pewnym momencie dokonać migracji.

Kod oraz źródła przykładu wykorzystującego `ScriptProcessorNode` znajdziemy poniżej:

- » <https://audio-network.rypula.pl/script-processor-node>
- » <https://audio-network.rypula.pl/script-processor-node-src>

## 6. WEB AUDIO API – PODSUMOWANIE

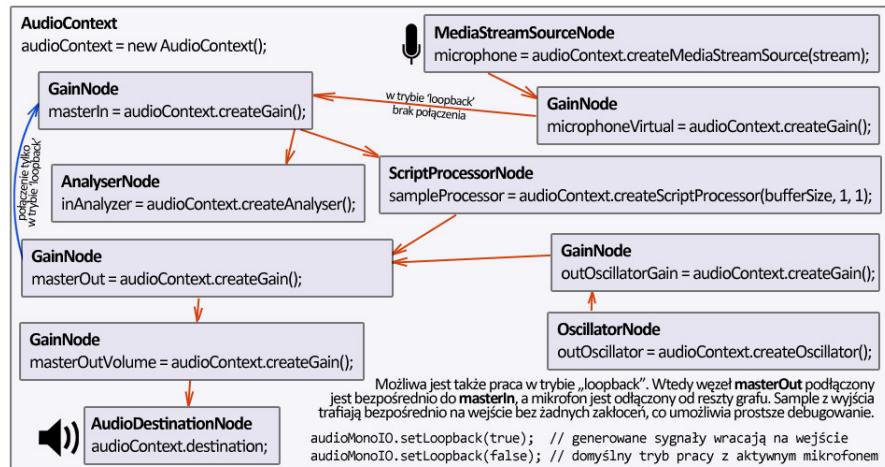
Węzły, które omówiliśmy, powinny w zupełności wystarczyć do zrealizowania prostej wymiany danych. Warto tutaj zaznaczyć, że Web Audio API jest ciągle rozwijane. Dobrym tego przykładem jest `ScriptProcessorNode`, który niebawem zostanie zastąpiony rozwiązaniem pracującym na osobnym wątku.

Opisane węzły to jedynie część potencjału drzewiastego w Web Audio API. Gdy chcemy dowiedzieć się czegoś więcej, warto zajrzeć na poniższe strony:

- » [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Audio\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API)
- » <https://webaudio.github.io/web-audio-api/>

## 7. IMPLEMENTUJEMY WŁASNĄ KLASĘ AUDIOMONOIO

Na tym etapie powinniśmy na tyle dużo wiedzieć o Web Audio API, by móc wyciągnąć z niego tylko to, co nam potrzeba. Do transmisji



Rysunek 11. Graf węzłów audio w klasie `AudioMonoIO`

danych będziemy potrzebowali tylko jednego kanału (mono). Jak wcześniej ustaliliśmy, oprócz ręcznego generowania próbek chcemy także wykorzystać potencjał rozwiązań wbudowanych, takich jak AnalyserNode czy OscillatorNode. Musimy zatem użyć w zasadzie wszystkich opisanych wcześniej węzłów. Na Rysunku 11 przedstawiono graf węzłów, jaki chcemy zaimplementować w klasie AudioMonoIO.

Po dodaniu obsługi błędów i zabiegów potrzebnych do pracy na starszych przeglądarkach w wyniku otrzymamy kompletną klasę AudioMonoIO. Jej kod dostępny jest na stronie projektu Audio-Network. Jest to jeden plik JavaScript bez żadnych dodatkowych zależności. Możemy go użyć w swoim projekcie, gdy nie chcemy wgrywać się w szczegóły komunikacji z Web Audio API.

» <https://audio-network.rypula.pl/audio-mono-io-class>

## 8. AUDIOMONOIO W PRAKTYCE

Na pierwszy ogień przetestujemy bardzo podstawowe użycie naszej klasy. Zaczniemy więc od utworzenia jej obiektu: var audioMonoIO = new AudioMonoIO(fftSize, bufferSize, smoothingTimeConstant). Pierwszy parametr konstruktora mówi sam za siebie. Jest to rozmiar FFT. Kolejny parametr o nazwie bufferSize jest to rozmiar bufora ScriptProcessorNode. Jak pamiętamy, im niższa wartość, tym mniejsze opóźnienia. Ostatni parametr mówi o wygładzaniu wyniku FFT. Na potrzeby transmisji danych najlepiej ustawić go na zero. Warto zaznaczyć, że parametry konstruktora nie są wymagane. Gdy ich nie podamy, zostaną użyte wartości domyślne.

Co zatem będzie robić nasza testowa aplikacja? Otóż na głośnikach postaramy się wygenerować dźwięk „syreny”, składający się z dwóch tonów podstawowych zmieniających się co jedną sekundę. Nasza syrena będzie dodatkowo zmiksowana z białym szumem, by zaprezentować obie metody generowania dźwięku, jakie mamy do dyspozycji w AudioMonoIO. Dźwięk syreny wygenerujemy za pomocą metody setPeriodicWave. W naszym przykładzie użyjemy tylko dwóch pierwszych jej parametrów: częstotliwości oraz głośności. Kolejne trzy to „globalne” przesunięcie fazowe, tablica amplitud harmonicznych oraz tablica faz harmonicznych. Skorzystamy z nich w jednym z późniejszych przykładów. Pod spodem ta metoda używa OscillatorNode, więc nie powinna obciążać naszego procesora tak jak generowanie fali samodzielnie próbka po próbce.

Wracając do białego szumu – wytworzmy go za pomocą metody setSampleInHandler. Przyjmuje ona funkcję będącą handlery do napływających paczek próbek. Paczka ta będzie dostępna jako parametr monoIn dostępny w funkcji naszego handlера. Parametr ten to oczywiście tablica sampli danego wycinka strumienia wyjściowego. By uzyskać pożądany efekt, wystarczy więc do każdego z tych sampli przypisać losową wartość z przedziału od -1 do 1. W praktyce nie chcemy, by nasz szum przyćmił syrenę, więc ograniczymy zakres do wartości z przedziału od -0.05 do 0.05.

Strumień sampli z mikrofonu przepuścimy natomiast przez „szukacz” największej wartości bezwzględnej amplitudy próbek z dziedziny czasu. Da to nam w rezultacie informację o ogólnej głośności sygnału z wejścia. W aplikacji tę informację wyświetlimy w formie paska, który będzie wypełniał się proporcjonalnie do głośności. Jako że sample dziedziny czasu możemy odczytać na dwa sposoby, umieścimy dwa paski. Będziemy mogli dzięki temu sprawdzić, czy zachowują się tak samo. Do tego celu użyjemy więc metody getTimeDomainData (tak naprawdę metoda getFloat-

TimeDomainData z AnalyserNode) oraz handlera ustawionego przez setSampleInHandler (pod spodem ScriptProcessorNode). W teorii powinniśmy uzyskać zbliżone wyniki, ponieważ ich okna powinny być relatywnie blisko siebie.

Nie byłoby zabawy, gdybyśmy nie wykorzystali także potencjału FFT. Odczyt wyników pracy transformaty możliwy jest poprzez metodę getFrequencyData. Przypomnijmy: zwracane dane to amplitudy częstotliwości składowych sygnału. Gdyby więc przeszukać całą tablicę w poszukiwaniu indeksu tablicy o najwyższej amplitudzie, moglibyśmy odgadnąć dominującą częstotliwość, jaką „słyszy” nasz mikrofon. Brzmi nieźle, pytanie tylko, jak zamienić numer indeksu na częstotliwość? Otóż rozdzielcość FFT jest równa częstotliwości próbkowania podzielonej przez rozmiar FFT. Wystarczy więc pomnożyć otrzymaną rozdzielcość przez znaleziony indeks, a otrzymana wartość będzie tym, czego szukamy.

Wszystkie operacje inicializacji tablicy Float32Array o odpowiednim rozmiarze ukryto w AudioMonoIO, by maksymalnie uprościć komunikację. Zarówno metoda getFrequencyData, jak i getTimeDomainData zwróci nam zatem tablicę w tradycyjny sposób jako wynik jej działania. Zobaczmy więc, jak wygląda fragment kodu naszego przykładu (Listing 5).

Listing 5. Fragment przykładu wykorzystującego AudioMonoIO

```
/* ... */
function nextAnimationFrame() {
    timeDomain = audioMonoIO.getTimeDomainData();
    freqDomain = audioMonoIO.getFrequencyData();
    frequencyPeak = getIndexOfMax(freqDomain) * fftResolution;
    timeDomainMaxAbsValueAnalyser = normalizeToUnit(
        timeDomain[getIndexOfMaxAbs(timeDomain)]
    );
    domGaugeAnalyser.style.width = (timeDomainMaxAbsValueAnalyser
        * 100) + '%';
    domPeakFrequency.innerHTML = frequencyPeak.toFixed(2) + ' Hz';
    requestAnimationFrame(nextAnimationFrame);
}

function init() {
    txFrequency = 2000;
    /* ... */
    audioMonoIO = new AudioMonoIO(8192, 8192); // fftSize, bufferSize
    fftResolution =
        audioMonoIO.getSampleRate() / audioMonoIO.getFFTSIZE();

    audioMonoIO.setSampleInHandler(function (monoIn) {
        timeDomainMaxAbsValueRaw = normalizeToUnit(
            monoIn[getIndexOfMaxAbs(monoIn)]
        );
        domGaugeRaw.style.width = (timeDomainMaxAbsValueRaw * 100)
            + '%';
    });
    audioMonoIO.setSampleOutHandler(function (monoOut, monoIn) {
        for (var i = 0; i < monoOut.length; i++) {
            monoOut[i] = 0.05 * (Math.random() * 2 - 1); // biały szum
            // możliwe jest także użycie sampli z mikrofonu:
            // monoOut[i] += 0.1 * monoIn[i];
        }
    });
    audioMonoIO.setPeriodicWave(txFrequency, 0.5); // głośność 50%
    setInterval(function () {
        txFrequency = (txFrequency === 2000 ? 2500 : 2000);
        audioMonoIO.setPeriodicWave(txFrequency, 0.5); // głośność 50%
    }, 1000);
    /* ... */
}
```

Pełna wersja jest niewiele dłuższa. Poniżej możemy znaleźć linki do wersji live oraz źródła:

» <https://audio-network.rypula.pl/audio-mono-io-basic>  
» <https://audio-network.rypula.pl/audio-mono-io-basic-src>

Po uruchomieniu przykładu na pierwszy rzut oka widać, że czas reakcji pierwszego paska ogólnej głośności oraz wartości najbliższej częstotliwości jest w zasadzie natychmiastowy. Możemy to przetestować, zwyczajnie pukając w mikrofon. Obydwa bazują na `AnalyserNode`. Tego jednak nie można powiedzieć o drugim pasku głośności używającym metody `setSampleInHandler` bieżącej na `ScriptProcessorNode`.

Różnica odświeżania wynika to z faktu, iż `AnalyserNode` z założenia powinien dostarczać dane jak najszybciej, dając wrażenie pracy w czasie rzeczywistym. Można to osiągnąć, umieszczając wywołania metod `getTimeDomainData` oraz `getFrequencyData` w konstrukcji używającej `requestAnimationFrame`. Jest to funkcja informująca przeglądarkę, że chcemy narysować kolejną klatkę ekranu z poziomu JavaScriptu. Przeglądarka kolejkuje nasze żądanie i wywołuje podany handler tuż przed jej wewnętrznym cyklem przerysowania ekranu. Dla małych wartości `fftSize` węzeł `AnalyserNode` działa bardzo szybko. W praktyce oznacza to, że nasze dane będą się odświeżać w zasadzie z prędkością odświeżania ekranu (60 fps, czyli co około 16 ms).

Z paczką sampli dostępną w handlerze metody `setSampleInHandler` jest inaczej. Tam prędkość odświeżania zależy od rozmiaru bufora. W naszym przypadku przy inicjalizacji obiektu `Audio-MonoIO` wstawiliśmy ten rozmiar na 8192. Oznacza to, że w jednej sekundzie mamy tylko około 5 „klatek” ( $44100 / 8192 = 5.4$  fps). Przekłada się to na odświeżanie co około 185.8 ms.

Taki czas odświeżania to prawie 1/5 sekundy, więc można to także usłyszeć podczas startu naszej aplikacji. Syrena uruchamia się od razu, podczas gdy biały szum pojawia się z lekkim opóźnieniem. Web Audio API na starcie zwyczajnie nie ma danych i daje nam trochę czasu na ich wypełnienie (do 185.8 ms w tym przypadku). Jeżeli nie zdążymy zmieścić się w tym czasie, nasz dźwięk będzie poszarpany. Dlatego też mały rozmiar bufora daje mniejsze opóźnienie, lecz może powodować problemy objawiające się np. trzaskami.

Na zakończenie przyjrzyjmy się bliżej informacji o najbliższej częstotliwości. Nasza syrena składa się z dwóch naprzemianie emitowanych tonów o częstotliwości 2000 Hz oraz 2500 Hz. Gdy zatem nasz mikrofon wyłapie dźwięk głośników, będziemy w stanie zweryfikować, czy te wartości się pokrywają. Przy założeniu, że nasza maszyna pracuje na standardowym próbkowaniu 44100 Hz, a `fftSize` jest równe 8192, daje nam to rozdzielcość widma około 5.3833 Hz ( $44100 / 8192$ ). Dla takiej rozdzielcości najbliższy prążek dla 2000 Hz będzie miał indeks 372. Oznacza to, że gdy transmitowany jest ton 2000 Hz, powinniśmy go odebrać za pomocą FFT jako 2002.59 Hz ( $372 * 5.3833$  Hz). Analogicznie dla tonu 2500 Hz będzie to indeks 464, który związany jest z częstotliwością 2497.85 Hz ( $464 * 5.3833$  Hz). Testy pokazują, że jesteśmy w stanie otrzymać dokładnie takie rezultaty. Pamiętajmy, że rozdzielcość możemy zwiększyć, używając większych rozmiarów `fftSize`.

Jak widać, ten przykład stanowi podwaliny pod transmisję danych. Gdy naszą syrenę uruchomimy na innej maszynie, podczas gdy druga maszyna będzie „słuchać”, będziemy w stanie łatwo odtworzyć, który ton jest aktualnie transmitowany.

## 8.1 Zabawa harmonicznymi i przesunięciem fazowym

Metoda `setPeriodicWave` zdolna jest także do generowania dźwięku na podstawie amplitud oraz przesunięć fazowych składowych harmonicznych. Ich zmiana przy zachowaniu tej samej czę-

stotliwości będzie wpływać na barwę dźwięku. To właśnie dzięki nim jesteśmy w stanie odróżnić różne instrumenty muzyczne czy też różne osoby emitujące ten sam dźwięk. Wiemy z sekcji o `OscillatorNode`, że harmoniczne o numerach większych niż 1 są to wielokrotności tonu podstawowego. Na wykresie widma amplitudowego pojawią się zatem w formie serii „pików”.

Możemy się pokusić o uproszczenie działania naszego ucha i ośrodka słuchu w mózgu do postaci zagadnień, które już poznaliśmy. Nasz aparat słuchowy dokonuje zatem w pewnym sensie Transformaty Fouriera. Potem nasz mózg „wybiera” pierwszy pik ze spektrum (patrząc od lewej) i odczytuje jego częstotliwość. To powoduje wrażenie dźwięku o danej wysokości. Każdy następny pik zmienia już tylko jego barwę. Dzięki temu jesteśmy w stanie powiedzieć, czy ktoś nam np. wspomniany dźwięk zagrał na skrzypcach czy zwyczajnie zagwizdał. Oczywiście to bardzo duże uproszczenie, ale daje ono pewien pogląd na to, jak działa nasz słuch.

Metoda `setPeriodicWave` umożliwia także sterowanie „globalnym” przesunięciem fazowym całego kształtu. W dziedzinie czasu będzie to oznaczać przesuwanie naszego kształtu w prawo lub w lewo bez zmiany jego wyglądu. Czy to także da się „usłyszeć”? Nie-stety, nie. Odgrywane po sobie takie same przebiegi okresowe będą brzmiały dokładnie tak samo, nawet gdy będą przesunięte w fazie.

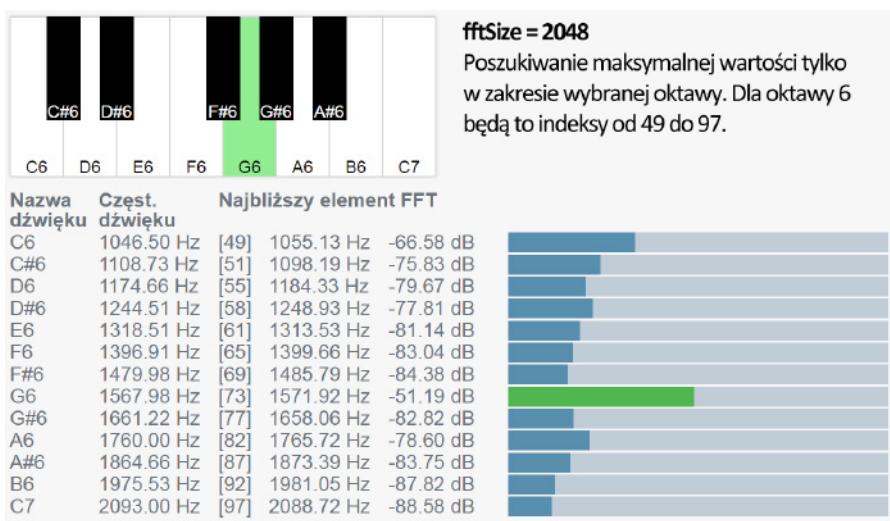
Na koniec wróćmy do przykładu. Działającą aplikację oraz jej kod możemy znaleźć pod linkami podanymi poniżej. Umożliwia ona oprócz generowania dźwięków także ich wizualizację. O samym procesie rysowania powiemy nieco więcej w następnej części tego artykułu. W przykładzie oprócz zwykłego sinusu dostępne są także przybliżenia przebiegów takich jak „piła”, „prostokąt” czy „trójkąt”. Wygenerowano je w całości za pomocą tablic amplitud harmonicznych. W Internecie można też znaleźć amplitudy harmoniczne dla instrumentów takich jak pianino czy wiolonczela. Brzmią one ciągle „komputerowo”, lecz są już nieco bardziej przyjemne niż czysta sinusoida.

- » <https://audio-network.rypula.pl/harmonics-and-phase>
- » <https://audio-network.rypula.pl/harmonics-and-phase-src>

## 8.2 Wykrywanie dźwięków pianina

Mając możliwość generowania dźwięku oraz wykrywania dominującej częstotliwości, możemy pokusić się o stworzenie nieco bardziej użytkowej aplikacji. Co powiemy na proste pianino z możliwością detekcji emitowanego dźwięku na drugim urządzeniu? Podwaliny pod tego typu aplikację przedstawiliśmy przy okazji przykładu z „syreną”. Do realizacji tego pomysłu musimy także dobrać odpowiednią częstotliwość do każdego klawisza. Potrzebne wzory znajdziemy na Rysunku 9. Mogą być one jednak nieco enigmatyczne, dlatego warto rozwinąć je o dodatkowy komentarz.

W skrócie: dźwięki na pianinie pogrupowane są w oktawy. Można je łatwo zlokalizować, gdyż tworzą charakterystyczny powtarzający się wzór złożony z klawiszy białych i czarnych. Wybierzmy zatem dla przykładu dowolny klawisz i znajdźmy jego odpowiednik w oktawie wyżej. Okazuje się, że jeśli częstotliwość pierwszego dźwięku oznaczymy jako  $x$  Hz, to częstotliwość klawisza z oktawy wyżej będzie równa  $2x$  Hz. Analogicznie dźwięk z oktawy niższej będzie miał częstotliwość  $0.5x$  Hz. Jak zatem dobrać częstotliwość pozostałych klawiszy? Otóż w oktawie mieści się 12 półtonów (klawisze białe i czarne). Przyjętym standardem jest strojenie równomiernie temperowane. W praktyce oznacza to, że częstotliwość klawisza  $n + 1$  możemy wygenerować, mnożąc częstotliwość klawisza  $n$  przez stałą. Ta stała to pierwiastek dwunastego stopnia z dwóch,



Rysunek 12. Nadawanie i odbieranie dźwięków muzycznych

czyli w przybliżeniu 1.059463. Skąd się wzięła taka dziwna liczba? Otóż stąd, że po wspomnianych 12 półtonach nasza częstotliwość musi być 2x większa niż częstotliwość, z której wyszliśmy. Gdy pomnożymy x dwanaście razy przez naszą stałą, otrzymamy finalnie 2x. Trafimy więc poprawnie na dźwięk o oktawie wyższy. Takie strojenie sprawia, że możemy zagrać melodię, zaczynając z dowolnego klawisza. Dopóki przesuniemy wszystkie dalsze dźwięki melodii o tyle samo półtonów co pierwszy, będzie ona ciągle „tą samą melodią”. Po prostu dla naszego mózgu znaczenie mają stosunki częstotliwości kolejnych dźwięków, a nie ich faktyczne wartości. Więcej informacji na ten temat możemy znaleźć na Wikipedii:

- » [https://en.wikipedia.org/wiki/Scientific\\_pitch\\_notation](https://en.wikipedia.org/wiki/Scientific_pitch_notation)
- » [https://en.wikipedia.org/wiki/Equal\\_temperament](https://en.wikipedia.org/wiki/Equal_temperament)

Przypomnijmy: do znalezienia w sygnale częstotliwości dominującej wystarczy w tablicy wyników FFT znaleźć element o maksymalnej wartości. Z numeru indeksu możemy łatwo obliczyć szukaną częstotliwość. Przeszukiwanie całej tablicy wyników FFT ma jednak wadę. Możemy zwyczajnie wychwycić częstotliwości, które co prawda będą „najgłośniejsze”, lecz niekoniecznie będą związane z nadawanymi przez drugie urządzenie dźwiękami. Ten problem możemy jednak łatwo rozwiązać, ograniczając zakres częstotliwości do tylko jednej oktawy. Wtedy przeszukiwanie tablicy wyników FFT ograniczy się tylko do wąskiego zakresu indeksów (Rysunek 12). W oktawie znajduje się 12 półtonów. Da to nam w rezultacie możliwość transmitowania 12 unikalnych symboli pomiędzy dwoma urządzeniami.

Działający przykład i kod możemy znaleźć pod poniższymi linkami. Pamiętajmy, by zawsze wybrać ten sam numer oktawy na obydwa urządzenia.

- » <https://audio-network.rypula.pl/piano>
- » <https://audio-network.rypula.pl/piano-src>

Zaprezentowana aplikacja oferuje oczywiście bardzo podstawową formę komunikacji. O tym, jak przesyłać prawdziwe bajty danych, opowiemy w następnej części tego artykułu.

## 8.3 Dokumentacja

Niektóre z przedstawionych w artykule przykładów używają klas, które z łatwością możemy użyć w innych projektach. Te klasy to np. `AudioMonoIO`, `FFTResult`, `MusicCalculator`. Dwie ostatnie zostały użyte w przykładzie z pianinem. Link do ich opisów znajdziemy poniżej:

- » <https://audio-network.rypula.pl/example-docs>

## 9 PODSUMOWANIE

Mam nadzieję, że chociaż jedna osoba czytająca ten artykuł zgodzi się ze mną, że Web Audio API to bardzo ciekawy moduł z ogromnym potencjałem. Jego możliwości oczywiście nie kończą się tylko na opisanych węzłach. Omawiając każdy z nich, często wykraczaliśmy poza wiedzę niezbędną do realizacji prostej transmisji danych. Myślę jednak, że zawsze warto poznać temat od podszewki. Wtedy jest nam dużo łatwiej wyciągnąć tylko to, co jest nam potrzebne. Upraszczamy nam to życie później. Efektem takiego uproszczenia jest nasza własna klasa `AudioMonoIO`. Ukrywa ona wszystkie szczegóły komunikacji z Web Audio API, wystawiając jedynie prosty interfejs.

Tematyka przetwarzania sygnałów jest o tyle specyficzna, że często szukając informacji, możemy natrafić na opisy bazujące na technice analogowej. Przez dziesięciolecia była to jedyna droga, by przesyłać coś np. za pomocą fal radiowych. Dopiero rozwój elektroniki umożliwił osiągnięcie podobnych efektów w sposób cyfrowy. Przekłada się to także na przetwarzanie dźwięku. Jednym z celów tej serii jest ujednolicenie tych wszystkich informacji i przekazanie ich w zrozumiałej dla programistów formie. Programiści zawsze widzą tablice próbek, a nie zmieniające się w czasie napięcie. Filtry to dla nas algorytmy, a nie układy zbudowane z kondensatorów, oporników czy cewek.

Naszym celem jest wykorzystanie Web Audio API do realizacji wymiany danych binarnych. W następnej części artykułu zabierzemy się zatem za brakujące elementy układanki. Omówimy m.in. techniki modulacji cyfrowej, opakujemy intuicyjny algorytm Dyskretnej Transformaty Fouriera z części pierwszej artykułu do postaci klasy oraz stworzymy kilka prostych aplikacji przesyłających dane za pośrednictwem dźwięku.



**ROBERT RYPUŁA**  
robert.rypula@gmail.com

Pasjonat komputerów i programowania. Od wielu lat związany z aplikacjami webowymi. Obecnie zatrudniony w PGS Software na stanowisku Frontend Developer. Wcześniej doświadczenie zdobywał w firmie Okinet. Poza technologiami webowymi twórca m.in. aplikacji HgtReader umożliwiającej rendering topografii całej Ziemi (OpenGL/Qt). Fan pisania własnych rozwiązań od zera wszędzie tam, gdzie jest to możliwe.

ZAPRASZAMY NA

# NOKIA OPEN DAY

## 2017



**3.06**  
Wrocław

Więcej szczegółów wkrótce na stronie: [nokiawroclaw.pl](http://nokiawroclaw.pl)

Enter  
**NOKIA**

# Zawód: Data Scientist, czyli jak zostać jednorożcem

Kariera w zawodzie programisty ma niewątpliwie wiele zalet, ale korzyściami, które mają najbardziej długofalowy charakter i przyczyniają się do wysokiego współczynnika zadowolenia z wykonywanej pracy przez programistów, są różnorodność wykonywanych zadań i mnogość otwartych perspektyw.

Jeden z wymiarów tej różnorodności to dziedzina, z którą stykamy się podczas tworzenia aplikacji i którą w dostatecznej mierze musimy poznąć, żeby zrozumieć specyficzne dla niej problemy. Niezależnie od tego, czy są to finanse, czy telekomunikacja, czy też rozwiązania dla instytucji publicznych, programista pokazuje w pełni swoją siłę, łącząc wiedzę informatyczną z inną dziedziną, w której odpowiednio zaaplikowane rozwiązania IT wnoszą znaczącą wartość dla ich odbiorców.

Perspektywy rozwoju i możliwości poruszania się po rynku pracy są dla programistów również bardzo szerokie, szczególnie w ostatnich (i zapewne wielu kolejnych) latach. Jeśli jesteśmy znuzeni jedną technologią, bez trudu znajdziemy wiele ofert związanych z pracą z dziesięcioma innymi. Czasami ta wielość możliwości, jak i coraz szybciej pojawiające się i przemijające trendy i mody na języki, czy paradygmaty programowania, bywa również uciążliwa. Od współczesnego programisty wymagana jest niezwykła wręcz zdolność adaptacji i dokształcania się przez całe życie. Technologia, która jeszcze dwa lata temu była nowinką, w tym roku może równie dobrze stać się popularnym rozwiązaniem używanym w większości zespołów developerskich, jak też i niewypałem, z którego wszyscy się wycofują.

W ostatnich latach okazało się, że ciekawych ścieżek kariery, którymi poruszać się mogą programiści, jest jeszcze więcej. Na horyzoncie pojawiła się nowa, wyjątkowo poszukiwana rola w zespołach IT – programistów, którzy potrafią analizować i znaleźć wartość w danych. Stało się tak niejako z konieczności – wszyscy gracze rynkowi zaczęli zbierać dane, i to dane w dużych ilościach. Dane z kliknięć, wejść na stronę, otwarć maili itd. wydawały się bardzo interesujące, dopóki nie okazało się, że tą „interesującą” wiedzę nie tak łatwo z nich wydobyć. Potrzeba do tego umiejętności z przynajmniej kilku dziedzin.

Oczywiście analiza danych nie jest niczym nowym, statystycy, analitycy czy specjaliści od hurtowni danych to zawody znane od dawna. Nowością było tu połączenie w jednej osobie kompetencji kojarzonych bardziej z nauką, a więc matematyczno-statystycznych, z inżynierijnymi kompetencjami w obszarze programowania. To wszystko, okraszone dobrymi umiejętnościami „miękkimi”, a więc komunikacyjnymi i prezentacyjnymi, a także znajomością danej dziedziny problemu, dało obraz idealnego kandydata na stanowisko *Data Scientist*, jednorożca z blyskiem w oku.

## CZYM TAK NAPRAWDĘ ZAJMUJE SIĘ DATA SCIENTIST

Najogólniejszym celem dla takiej osoby jest wydobycie interesującej dla danego biznesu wiedzy z dostępnych danych. Dla sklepu

internetowego może to być wiedza o klientach, ich segmentacja i wydobycie wzorców zachowań dla każdej z wyróżnionych grup. Może to być system rekomendacyjny, który zwiększa sprzedaż, proponując klientom dokładnie to, czego potrzebują, zanim sami jeszcze sięgną po odpowiedni produkt. Dla domu mediowego może to być optymalny sposób prezentacji treści reklamowych, dopasowanych do użytkowników i ich preferencji.

Najbardziej wartościowa wiedza pojawia się w takich zastosowaniach w danych zawierających miliardy rekordów – próbka musi być wystarczająco liczna, aby zadziałała statystyka. Stąd też głośne od kilku lat hasło *big data*, które ściśle powiązane jest z rolą *Data Scientist* i które stanowiło praprzyczynę konieczności połączenia wiedzy statystycznej z inżynierską w jednej roli. Analiza tak dużych danych zaczęła bowiem wymagać specjalistycznej infrastruktury i umiejętności posługiwania się nowymi paradygmatami programowania, aby uzyskać jakiekolwiek interesujące wnioski.

## JAK ZOSTAĆ DATA SCIENTIST

Od razu nasuwa się pytanie – skąd, biorą się „*Data Scientists*”? Czy są to programiści, którzy nauczyli się statystyki, czy odwrotnie – analitycy, którzy podszkolili się z programowania? Gdzie zdobyli potrzebne kompetencje? Okazuje się, że background osób pełniących te role w zespołach jest bardzo zróżnicowany i każdy z wymienionych scenariuszy jest możliwy. Kluczowe okazują się tu takie cechy jak docieklełość i umiejętność rozwiązywania problemu, co w powiązaniu z ugruntowanymi kompetencjami z zakresu programowania lub analizy danych pozwala myśleć o wejściu w tę nową rolę.

W zależności od kompetencji posiadanych na początku drogi ścieżka do roli *Data Scientist* może być bardzo różna. Często jest to długi okres „stażu”, podczas którego osoba na stanowisku juniorskim zdobywa brakujące umiejętności, na przykład przechodzi kolejne etapy wtajemniczenia jako początkujący programista. Wspierać się można mnogością materiałów dostępnych online: kursów, tutoriali czy podręczników. Szczególnie wartościowe są kursy uniwersyteckie, typu Coursera czy edX.

Dużym minusem zdobywania wiedzy samodzielnie, korzystając z tego rodzaju zasobów, jest jednak brak bezpośredniego kontaktu z nauczycielem i innymi uczącymi się osobami, borykającymi się z podobnymi problemami, które pojawiają się na etapie zdobywania nowej, trudnej do przyswojenia wiedzy. W pamięci wielu osób, które studiowały, czy nawet uczyły się na poziomie szkolnym, pozostają długie popołudnia spędzone z innymi towarzyszami niedoli na wspólnym rozwiązywaniu zadań, pomaganiu sobie i wzajemnym wyjaśnianiu trudniejszych kwestii. Nieprzypadkowo w kulturze zachodniej tak kluczową rolę odgrywają kampusy



uniwersyteckie – miejsca, gdzie studenci mieszkają wspólnie przez okres nauki, żeby być dla siebie wsparciem i motywacją. W przypadku materiałów internetowych jesteśmy często pozostawieni sami sobie i wymaga to ogromnej determinacji, żeby z sukcesem i o czasie ukończyć kurs online, wykonywany przecież często wieczorami po pracy czy w weekendy.

Obecna oferta uniwersytecka jest wciąż dość ograniczona, jeśli chodzi o zakres *Data Science*. Wynika to z typowego dla uczelni problemu wolnej adaptacji do szybko zmieniającego się otoczenia. Jest to z jednej strony zaleta i mechanizm obronny nauki – przed przemijającymi trendami – ale z drugiej strony uniemożliwia szybkie wdrażanie wartościowej wiedzy na wczesnym etapie rozwoju danego obszaru. Jednym z wyjątków są prowadzone od 2016 roku studia podyplomowe Data Science i Big Data na Politechnice Warszawskiej (<http://ds.ii.pw.edu.pl>). Studia te łączą ugruntowaną wiedzę z zakresu statystyki, analizy danych i uczenia maszynowego z najnowszymi trendami przetwarzania dużych danych, rozwiązań technologicznych i infrastrukturalnych, wiążących się z *big data*.

Alternatywą do zdobywania wiedzy na studiach, co trwa zwykle przynajmniej rok w trybie weekendowym, są intensywne kursy zawodowe, określane jako bootcampy programistyczne. Bootcampy wywodzą się ze Stanów Zjednoczonych, gdzie koncepcja ta wyrosła z połączenia tradycji kampusów uniwersyteckich z potrzebą szybkiego i intensywnego nauczenia się praktycznych umiejętności w obszarze IT. Jest to więc taki format nauki, który łączy zalety klasycznej edukacji uniwersyteckiej (stały kontakt z innymi studentami, „zanzurzenie się” w środowisku, którego chcemy być częścią, dostęp do nauczycieli, z którymi można dyskutować i prosić o wyjaśnienia problemów) z kursami zawodowymi (intensywna nauka praktycznych umiejętności, a więc tak dobranych, by były jak najbardziej dopasowane do bieżących potrzeb rynku i przekazywanie przez osoby mające na co dzień styczność z rzeczywistą pracą na podobnym stanowisku).

Bootcampy programistyczne trwają zwykle od 6 do kilkunastu tygodni i pozwalają w krótkim czasie zdobyć umiejętności pozwalające na rozpoczęcie kariery w wybranej roli w zespole IT. W zależności od konkretnej ścieżki i indywidualnych możliwości bootcamp może po pierwsze umożliwić w ogóle zdobycie pierwszej pracy w zawodzie w branży IT albo też znaczco skrócić czas stażu na stanowisku juniorskim. Intensywna nauka poprzez rozwiązywanie praktycznych problemów na bootcampie jest najczęściej odpowiednikiem kilkumiesięcznego stażu w przedsiębiorstwie. Nauka polega bowiem głównie na rozwiązywaniu problemów, oczywiście odpowiednio wyizolowanych, które pojawiają się w rzeczywistej pracy programisty czy specjalisty *Data Science* i które rozpatrywane są w środowisku jak najbardziej zbliżonym do takiej właśnie realnej pracy.

W kontekście *Data Science* najbardziej znane bootcampy zagraniczne pozwalające na zdobycie kompetencji z tego zakresu to Metis czy Galvanize. W Polsce pierwsza ścieżka zawodowa z tego zakresu dostępna jest w Kodołamaczu (<http://kodołamacz.pl>), gdzie podczas 7 tygodni nauki realizowany jest odpowiednik rocznego kursu studiów podyplomowych, z silnym naciskiem na praktyczne umiejętności potrzebne na takim stanowisku w największych polskich firmach i z wykorzystaniem najpopularniejszych obecnie technologii. Ukończenie tego rodzaju kursu daje silną podstawę zarówno w obszarze statystyki, analizy danych, jak i uczenia maszynowego (w tym *deep learning*), a także – od strony bardziej inżynierskiej – w obszarze programowania zorientowanego na przetwarzanie danych i technologie *big data*, np. Apache Spark.

Niezależnie od wybranej ścieżki cel jest wyjątkowo ekscytujący: możliwość pracy w tym obszarze IT, który najbardziej obecnie zmienia otaczający nas świat. Internet rzeczy, autonomiczne samochody, systemy komunikujące się z nami w języku naturalnym – to jest przyszłość, która dzieje się już teraz, a narzędzia, które pozwalają na zmierzenie się z tymi problemami, są na wyciągnięcie ręki.



### ŁUKASZ KOBYLIŃSKI

Chief Science Officer w Sages, programista, Data Scientist i trener. Od wielu lat zajmuje się analizą danych i uczeniem maszynowym, początkowo w odniesieniu do obrazów, a obecnie w zastosowaniu do przetwarzania języka naturalnego. Członek Rady Programowej Studiów Big Data i Data Science na Politechnice Warszawskiej oraz opiekun merytoryczny bootcampu Kodołamacz.pl.

# Jak sprzedać refaktoryzację?

## Przypadek Nordea Bank AB

Zaskakująco refaktoryzacja nie jest często wyzwaniem technicznym. Zespoły zazwyczaj trafnie diagnozują nieefektywny design kodu. Z reguły również mają pomysły, co należy z tym zrobić, jakich narzędzi użyć, jakich przekształceń refaktoryzacyjnych. Gdyby mieli do dyspozycji wystarczającą ilość czasu oraz wystarczający budżet, to prawdopodobnie doprowadiliby sprawy do końca.

**M**yśląc „refaktoryzacja”, być może przychodzą Ci na myśl przekształcenia typu *Extract Class* czy *Introduce Polymorphism*. To są istotne przekształcenia codziennej pracy z kodem. W tym artykule mamy na myśli coś bardziej złożonego – strategiczną refaktoryzację kodu. To rozróżnienie zostało wprowadzone przez konsultantów BNS IT w metodzie „Naturalnego porządku refaktoryzacji” – opracowanym przez nich procesie pracy z kodem odziedziczonym. Więcej na ten temat w artykule *Natural Course of Refactoring – a Refactoring Workflow* ([www.infoq.com/articles/natural-course-refactoring](http://www.infoq.com/articles/natural-course-refactoring)).

Refaktoryzacja jest jednak w pierwszej kolejności przedsięwzięciem organizacyjnym, na który trzeba zdobyć zgodę oraz fundusze i którym trzeba zarządzać w dłuższym okresie czasu. Posłuchaj, w jaki sposób podeślismy do tego tematu w projekcie systemu e-bankowości w Nordea Bank AB S.A. Oddział w Polsce.

### KONTEKST

W wyniku decyzji biznesowych Nordea IT Polska (obecnie Nordea Bank AB S.A. Oddział w Polsce) przestała rozwijać system e-bankowości na rynek polski, utrzymując jedynie ten produkt w krajach bałtyckich. Po jakimś czasie zaczęły pojawiać się następujące trudności:

- » Klienci biznesowi długo oczekiwali na zlecone funkcjonalności – przeciętnie czas ten wynosił kilka miesięcy od momentu zgłoszenia,
- » Niektóre zgłoszenia były niemożliwe do zrealizowania ze względu na ograniczenia technologii, w których stworzony był system e-bankowości,
- » Klienci czasem zamawiali pojedyncze funkcjonalności twozone w nowszych technologiach u lokalnych dostawców. Konieczność utrzymywania tych funkcjonalności oraz ich integracji z systemami bankowymi spoczywała na Nordea IT Polska.

Na początku stycznia 2015 roku nawiązaliśmy współpracę (Michał Bartylewski/BNS IT i Nordea IT Polska), której cel został sformułowany następująco: skrócić czas dostarczania nowych funkcjonalności do 30 dni. Aby nadać tej nowej inicjatywie w organizacji bardziej nacalny wymiar, nadaliśmy jej nazwę – Action-30.

### Jaka jest rzeczywistość?

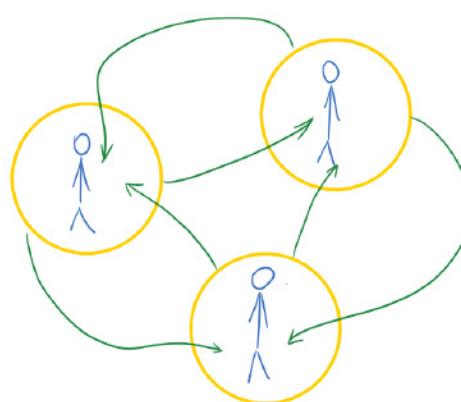
Gdy zaczynasz pracować z organizacją w kontekście jakiegoś tematu, zazwyczaj napotykasz się z wieloma punktami widzenia.

Niezwykle ważne jest, aby już na samym początku uświadomić sobie, że są tylko różne narracje na temat tej samej rzeczywistości i żadna z nich nie jest bardziej prawdziwa niż inne. Rozmawiasz z różnymi osobami z organizacji, które niejednokrotnie prezentują wykluczające się informacje, a jednak każda z tych informacji jest spójna i wydaje się być uzasadniona.

Jednak w gąszczu tych informacji da się zauważyc powtarzany schemat – rozmówcy zazwyczaj mówią o kimś lub o czymś innym: o innych departamentach, o współpracownikach, o klientach. Rzadko kiedy mówią o sobie, na przykład:

- » Zespoły się opóźniają,
- » PO powinien wiedzieć, czego chce,
- » Wymagania nie są wystarczająco precyzyjne;
- » Klienci często zmieniają zdanie;
- » Security zawsze blokuje wdrożenia;
- » itp..

Jest sposób wyrażania się, który nazywamy „komunikatem Ty”. Jeśli zastanawiasz się teraz „no i co z tego, że tak rozmawiają?”, to przypatrz się wizualizacji tego typu komunikacji na Rysunku 1.



Rysunek 1. Komunikat Ty

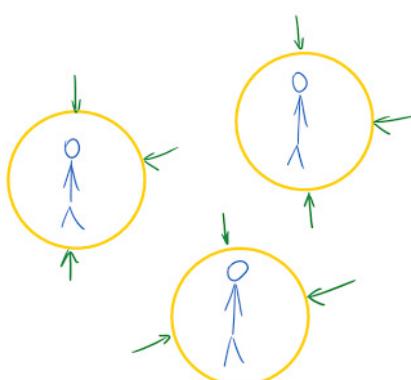
Gdy jedyne, co posiadasz, to informacje od rozmówców przekazane w formie „komunikatu Ty”, to nie diagnozują one potrzeb danej organizacji. Jeśli każdy wypowiada się wyłącznie na temat kogoś/ czegoś innego, to jedyne, czego się dowiesz, to opinie rozmówców o kimś/o czymś. Co więcej, ta forma komunikacji sugeruje, że przyczyną istniejących problemów są „ci inni”. W takiej sytuacji możliwymi rozwiązaniami są „walcz z nimi” lub „uciekaj od nich”.

W trakcie diagnozowania sytuacji organizacji korzystniej jest tak prowadzić rozmowę, aby uzyskiwać informację w postaci „ko-

munikatów Ja". Są to wypowiedzi w postaci: *Chcę..., Nie chcę..., Brakuje mi..., Potrzebuję..., Podoba mi się, gdy...* Na przykład:

- » Zamiast *Zespoły się opóźniają*, możesz usłyszeć:
  - » *Chcę zamykać projekty w terminie,*
  - » *Nie chcę pracować pod presją czasu,*
  - » *Chcę być możliwie szybko informowany o pojawiających się problemach.*
- » Zamiast *PO powinien wiedzieć, czego chce*, możesz usłyszeć:
  - » *Nie w pełni rozumiem, po co robimy ten produkt,*
  - » *Potrzebuję więcej kryteriów akceptacyjnych do zleconych zadań,*
  - » *Chcę mieć realny wpływ na rozwój tego produktu.*

Ponownie przypatrz się wizualizacji komunikacji z użyciem formy „Ja” (Rysunek 2).

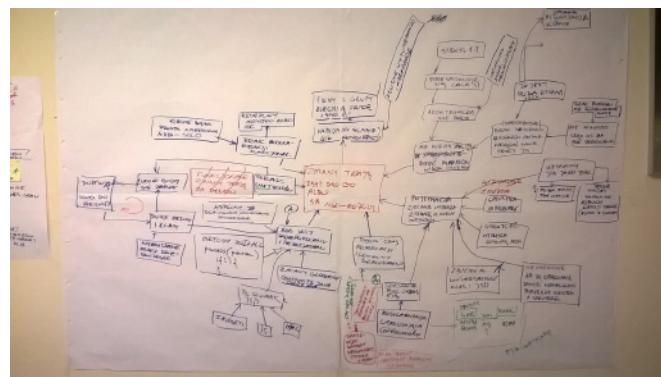


Rysunek 2. Komunikat Ja

Zauważ, że tym razem już sama forma wypowiedzi sugeruje możliwe rozwiązania dla organizacji. Co więcej, eksponowanie indywidualnych potrzeb osób, zespołów czy jednostek organizacyjnych jest otwarciem na współpracę w celu zaspokojenia tych potrzeb.

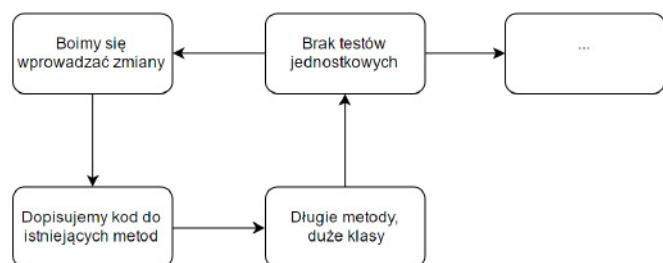
### Źródła problemu

Pierwszym krokiem w poszukiwaniu przyczyn problemów rozwoju systemu e-bankowości było przygotowanie ogólnego diagramu RCA (ang. Root Cause Analysis)<sup>1</sup>.



Rysunek 3. Diagram RCA

Wyjściowym objawem, od którego zaczęliśmy analizę, było: „Zmiany trwają długo albo są niemożliwe”. To, czego poszukiwaliśmy, to przyczyny, do których zbiega się znaczna liczba objawów oraz zapętleń. Przykład takiego zapętlenia pokazano na Rysunku 4.

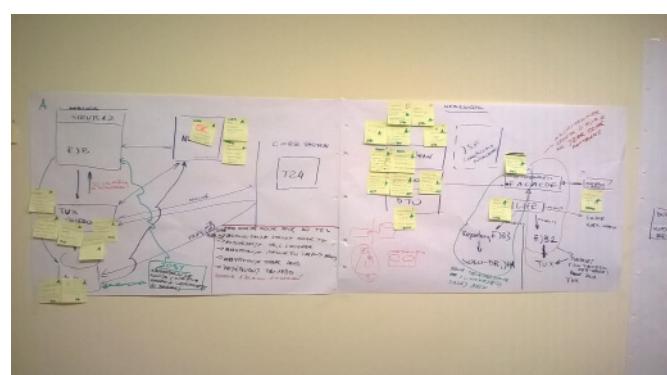


Rysunek 4. Zapętlenie na diagramie RCA

Posiłkując się opracowanym diagramem, wskazaliśmy dwie źródłowe przyczyny analizowanego problemu, że „Zmiany trwają długo albo są niemożliwe”:

- » **Używanie szkieletu Struts 1.2** – system, nad którym pracowaliśmy, był rozwijany od prawie dziesięciu lat. Pierwotnie użyto w nim Struts 1.2 – szkieletu do tworzenia aplikacji webowych; w roku 2015 to rozwiązanie było już archaiczne i znacznie ograniczało możliwość tworzenia nowoczesnych aplikacji internetowych.
- » **Nadmierna uniwersalność kodu** – system był tworzony przez wiele osób, wdrażany w kilku krajach, w których zależnie od regulatora usług bankowych niektóre funkcjonalności nieznacznie różniły się od siebie; owe zróżnicowanie zostało zaimplementowane poprzez umieszczenie instrukcji warunkowych po stronie widoku, serwisów aplikacyjnych oraz w niektórych usługach domenowych. W konsekwencji kod charakteryzował się względnie dużą złożonością cyklotomatyczną<sup>2</sup>.

### Analiza architektury



Rysunek 5. Analiza architektury

Kolejnym krokiem była sesja diagnozowania problemów związanych z istniejącą architekturą. Za kluczowe, mające bezpośredni i największy związek z czasem dostarczania funkcjonalności, uznaliśmy:

- » **Konieczność przestrzegania architektury Struts 1.2** – architektura ta nie była stworzona według paradygmatu *convention over configuration*, co wydłuża dodawanie oraz modyfikowanie funkcjonalności.
- » **Przetwarzanie biznesowe na stronach JSP (najczęściej przypadki)** – ten sposób pisania jest wyjątkowo nieprzyjazny szybkiemu wprowadzaniu zmian.
- » **Duża ilość nieużywanego kodu** – w związku z tym, że system przestał być wdrażany w Polsce, kod z tym związany był

1. Ponieważ nie jest celem artykułu szczegółowo omawianie użytych technik, zainteresowanych odsyłamy na początek do [https://en.wikipedia.org/wiki/Root\\_cause\\_analysis](https://en.wikipedia.org/wiki/Root_cause_analysis).

2. Ang. Cyclomatic Complexity – [https://en.wikipedia.org/wiki/Cyclomatic\\_complexity](https://en.wikipedia.org/wiki/Cyclomatic_complexity).

nieużywany i nierozerwijany, a jednocześnie obecny w głównej linii kodu (ang. *code baseline*).

- » **Silne zależności od innych systemów** – ten aspekt wydłużał czas implementacji przede wszystkim wtedy, gdy nowe wymaganie powodowało zmiany w głównym systemie transakcyjnym (ang. *core system*).

## Analiza procesu dostarczania oprogramowania



Rysunek 6. Mapowanie procesu dostarczania oprogramowania

Na etapie analizy procesu dostarczania oprogramowania skupiliśmy się na poszczególnych czynnościach pomiędzy zgłoszeniem zapotrzebowania biznesowego a wdrożeniem oprogramowania. Szukaliśmy w pierwszej kolejności wąskich gardeł procesu, a także takich rodzajów zgłoszeń biznesowych, które najczęściej były odrzucane z powodu ograniczeń technologicznych. Największy wpływ na przepustowość tego procesu miały:

- » **Wielozadaniowość** – programiści byli zaangażowani w więcej niż jeden projekt, w skrajnych przypadkach samo przełączenie się pomiędzy zadaniami z różnych projektów, włączając w to przełączenie między środowiskami programistycznymi oraz wejście w tryb efektywnego realizowania zadań, zajmowało od 30 do 60 minut.
- » **Nieefektywne spotkania** – ilość projektów jednocześnie realizowanych przekładała się na ilość spotkań.
- » **Specjalizacja programistów** – utrudniało to współpracę oraz powodowało duże lokalne obciążenia poszczególnych programistów.

## Metoda pracy

Od samego początku zdawaliśmy sobie sprawę z zestawu ograniczeń, w których musieliśmy się poruszać:

- » *Action-30* nie może zaburzać toczących się projektów.
- » Początkowo w przedsięwzięciu może wziąć pięciu ochotników.
- » Każdy z programistów może poświęcić na pracę nad *Action-30* jeden dzień w tygodniu.
- » Możliwie szybko trzeba pokazać biznesową korzyść z inwestycji w refaktoryzację.
- » Metoda pracy powinna być ciekawa i angażująca, gdyż praca nad *Action-30* jest zajęciem dodatkowym.

Ostatecznie zdecydowaliśmy, że będziemy pracować w a'la Scrum. „A'la”, ponieważ nie był to właściwie Scrum, lecz swobodna na jego temat wariacja. Nazwanie naszej metody „a'la Scrumem” miało bardziej charakter PR-owy i motywacyjny dla zespołu niż korzystanie ze szkieletu Scrum. Założenia metody pracy były następujące:

- » Cały zespół pracuje nad *Action-30* wyłącznie w czwartki.

- » A'la sprint trwa jeden miesiąc, czyli 4 czwartki.
- » Po zakończeniu a'la sprintu odbywa się retrospekcja, przegląd z możliwie dużą liczbą potencjalnych interesariuszy oraz planowanie kolejnego a'la sprintu.
- » Nad zadaniami pracujemy dwójkami.
- » W porozumieniu z Kierownikiem Działu podjęliśmy wspólne zobowiązanie konsekwentnego unikania nieproduktywnych spotkań.
- » Aby ułatwić zespołowi osiągnięcie sukcesu, w trakcie planowania szczegółowo dekomponowaliśmy zadania na składowe nie dłuższe niż 4 godziny.
- » Każdy a'la sprint ma konkretny i mierzalny cel do osiągnięcia.

## A'la sprint 1

Na pierwszy a'la sprint zaplanowaliśmy zestaw zadań organizacyjnych oraz tylko jedno zadanie techniczne. Polegało ono na wyodrębnieniu z systemu e-bankowości bezpiecznego webserwisu udostępniającego dane dla jednego z widoków.

Dlaczego tylko jedno zadanie techniczne na 20 osobodni? Przede wszystkim dlatego, aby mieć niemal zagwarantowany sukces pierwszego sprintu. Inicjatywy takie jak *Action-30* są dość niestabilne w organizacji, niewielka awaria w bieżących projektach może skutecznie uniemożliwić zespołowi zaplanowane prace nad nową inicjatywą. Dla przyszłości całego przedsięwzięcia korzystniej jest mieć jeden malutki sukces, niż wielkie plany, co do których entuzjazm ulatnia się po dwóch tygodniach.

## Retrospekcja

Podsumowanie zespołu po pierwszym a'la sprintie było następujące:

- » *Cel został osiągnięty.*
- » *Bardzo dobrze pracuje się w dwie osoby nad zadaniem.*
- » *Spotkania się ustrukturyzowały i zagościły, z perspektywy czasu trwania jest ich mniej.*
- » *Udało się zrobić więcej niż planowaliśmy.*
- » *Wprowadźmy jeszcze więcej zespołowego planowania.*

## Obserwacje

Pierwszą sprawą, która momentalnie zwróciła naszą uwagę, było to, że *Action-30* zaczyna żyć własnym życiem. Pozostali pracownicy rozmawiają o tej inicjatywie w kuchni, przy kawie. Interesują się, pytają, czy można się przyłączyć.

Ku zaskoczeniu zespołu udało się wykonać więcej pracy niż to zostało zaplanowane. Stało się tak dlatego, że poszczególni programiści poświęcali na pracę nad *Action-30* więcej czasu niż planowane czwartki. Sam ten fakt jest bardzo interesujący. Oczywiście żaden z bieżących projektów nie ucierpiał. Wysnuwamy na tej podstawie wniosek, że **jeśli coś jest fajne i ciekawe, to czas się znajdzie**. Co więcej, ten „dodatkowy” czas nie wymaga formalnego harmonogramowania i procedowania. Ludzie z własnej inicjatywy inicjują lokalne optymalizacje, aby pracować nad tym, co rzeczywiście się im podoba.

Pierwszy a'la sprint unaocznił również ogromną wartość managera w przedsięwzięciach tego typu – inicjujących zmianę. Zespół świetnie sobie poradzi z zadaniami technicznymi, lecz w przypadku zadań organizacyjnych, a zwłaszcza takich, które zmieniają sposób współpracy pomiędzy jednostkami organizacyjnymi i przełamują silosy, potrzebna jest pomoc kogoś, kto zna formalne i nieformalne ścieżki decyzyjne w organizacji. Dzięki pomocy managera zespół

ograniczył ilość spotkań oraz zainicjował inne kanały komunikacji ze współpracującymi wydziałami.

W kontekście zespołu zaczynały być również widoczne powstające zasady współpracy (ang. *team norms*):

- » Zawsze pracujemy w dwie osoby nad zadaniem.
- » Świętujemy sukcesy
- » Unikamy nieproduktywnych spotkań.
- » ...

a także wytyczne refaktoryzacji (ang. *refactoring guidelines*) w formie opisu typowych zadań oraz problemów i ich rozwiązań. Wytyczne te porządkowały wiedzę zespołu. Przeznaczone były również dla przyszłych członków zespołu.

## A'la sprint 2

Począwszy od drugiego a'la sprintu, planowaliśmy regularne pokazywanie interesariuszom korzyści biznesowych z refaktoryzacji. Choć refaktoryzacja jest zmianą struktury kodu bez zmiany jego zewnętrznych funkcjonalności, to bez wskazywania na efekty biznesowe jest ona praktycznie nie do „sprzedania”. Wykorzystaliśmy dwa podejścia:

- » eskalowanie organizacyjnych korzyści z refaktoryzacji – wskazujemy mierzalne korzyści z refaktoryzacji dla organizacji, np. skrócenie czasu developmentu.
- » architectural slices of customer-centric features – pracę refaktoryzacyjne zawsze łączymy (w rozsądnej proporcji) ulepszeniem/dodaniem wartościowych funkcjonalności.

## Raport – podstawowe narzędzie komunikacji z organizacją

W trakcie początkowych analiz zwróciliśmy uwagę na dużą ilość nieużywanego kodu. Tak jak wspomnieliśmy wcześniej, system przestał być rozwijany na rynek polski, jednak kod związanego z specyfiką polskiej bankowości był wciąż obecny w głównej linii kodu. Dodatkowo kod nie był opakowany („zenkapsulowany”), lecz umieszczony wewnętrz algorytmów. Oznacza to, że zarządzanie zmiennością oraz rozbudowa systemu odbywała się nie poprzez dodawanie nowych implementacji, lecz poprzez dokładanie instrukcji warunkowych wewnętrz metod (Rysunek 7).

Aby dotrzeć z przekazem do decydentów, postanowiliśmy wyrazić korzyść z usunięcia tego kodu w języku pieniężny i harmono-

gramu. W tym celu przygotowaliśmy próbkę kodu, w którym zmierzyliśmy ilość wierszy, oraz złożoność cyklomatyczną przed oraz po usunięciu kodu związanego z polskim rynkiem. Choć mierzenie ilości wierszy kodu daje niewiele informacji programistycznie, to dla osoby nietechnicznej jest to synonim rozmiaru oprogramowania.

O wiele ciekawsza jest złożoność cyklomatyczna kodu. Daje ona информацию o liczbie ścieżek decyzyjnych w kodzie, a zatem jest również miarą ilości testów koniecznych do pokrycia funkcjonalności. Okazało się, że po usunięciu „polskiego” kodu złożoność tej próbki zmalała o 15%, co przełożyło się na redukcję liczby koniecznych testów o 29.

W tym momencie pozwoliliśmy sobie na trochę karkolomną szacunkową ekstrapolację – aproksymując otrzymane wyniki po wszystkich podobnych próbkach w kodzie i przeliczając liczbę zredukowanych testów na dni robocze, otrzymaliśmy pewne przybliżenie na temat tego, ile „mendejdów” możemy zaoszczędzić dzięki proponowanej refaktoryzacji. To jest już informacja, która z pewnością zostanie usłyszana...

W dużej organizacji podstawowym narzędziem komunikacji o dużym znaczeniu są raporty. Raz opublikowany raport jest przekazywany pomiędzy skrzynkami mailowymi niczym plotka w kuchni, a informacja w nim zawarta dociera do szerokiego grona odbiorców. Przygotowaliśmy więc raport z opisanymi szacunkami i przesłaliśmy do najbliższego przełożonego. Tak jak się spodziewaliśmy, informacja o korzyściach z refaktoryzacji została usłyszana oraz zrozumiana.

## Pokażmy wartość biznesową

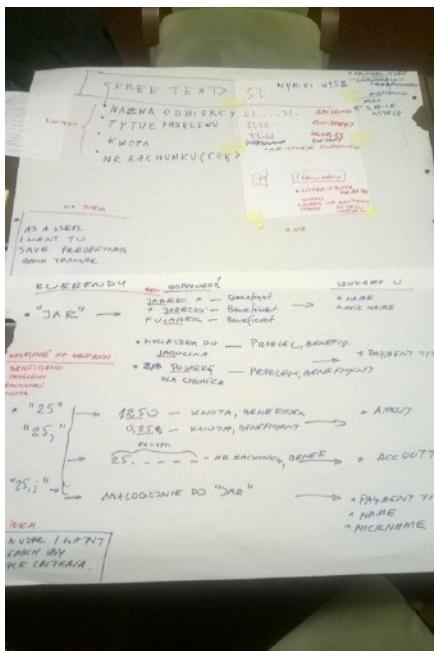
Drugą metodą działania było łączenie prac refaktoryzacyjnych z ulepszaniem/dodawaniem wartościowych funkcjonalności. Choć refaktoryzacja jest z definicji upiększeniem kodu bez zmiany jego funkcjonalności, to jednak wydaje nam się konieczne ze względów organizacyjnych i finansowych łączyć ją z dodawaniem wartości do produktu, preferując zazwyczaj tę drugą aktywność.

Postanowiliśmy, że biznesową wizytówką refaktoryzacji będzie zrealizowanie jednej z tych funkcjonalności, które do tej pory były oczekiwane przez biznes, lecz niemożliwe do wykonania ze względów technologicznych.

Tak jak wspominaliśmy, jedną z głównym przyczyn problemów z dostarczaniem oprogramowania było używanie szkieletu Struts 1.2 – zdecydowaliśmy się wypróbować AngularJS.

	Status	r66825	r66826	Δ	r66825	r66826	Δ
					LOC		
business/facade/MobileActivationFacade.java	Zmieniony	926	936	-10	61	59	2
business/facade/MobileActivationFacadePL.java	Usunięty	168	0	168	7	0	7
ions/sms/MobileActivationAction.java	Zmieniony	1137	1090	47	99	92	7
ions/sms/MobileActivationActionPL.java	Usunięty	120	0	120	8	0	8
pp/actions/events/MobileActivationWapAction.java	Zmieniony	59	50	9	5	3	2
one_entry.jsp	Zmieniony	290	186	104	9	6	3
ss-config.xml	Zmieniony	11752	11699	53	ndt.	ndt.	ndt.
config_pl.xml	Zmieniony	1569	1561	8	ndt.	ndt.	ndt.
:/weblogic.xml	Zmieniony	3	4	-1	ndt.	ndt.	ndt.
ogic.xml	Zmieniony	39	38	1	ndt.	ndt.	ndt.
			ΣΔ	499		ΣΔ	29
			ΣΔ [%]	3%		ΣΔ [%]	15%

Rysunek 7. Konsekwencje usunięcia nadmiarowego kodu



Rysunek 8. Biznesowa wizytówka refaktoryzacji

Ponieważ zespół nie miał wcześniejszych doświadczeń z nową technologią, skupiliśmy się w pierwszej kolejności na dokładnym wstępnie specyfikowaniu ekranów użytkownika, a następnie zaprosiliśmy specjalistę od JavaScrpit na kilkugodzinny warsztat. Jego zadaniem było przygotowanie w AngularJS widoków oraz przekazanie podstaw jego konstrukcji i działania tak, aby zespół był w stanie wdrożyć go w swoim systemie.

W ten sposób programiści mogli się skupić na refaktoryzacji backendu, ostylowaniu nowych ekranów oraz zintegrowaniu ich z systemem e-bankowości. Zmniejszyliśmy drastycznie próg wejścia w nową technologię i umożliwiłyśmy łatwiejsze osiągnięcie sukcesu. Na tamtym etapie priorytetem było wypromowanie konieczności refaktoryzacji, a nie migracja do nowej technologii.

*Biznes chce wiecji*

Od tamtej pory planowaliśmy kolejne a'la sprints tak, aby refakto-ryzując, dostarczać jednocześnie biznesową wizytówkę tej refakto-ryzacji – najczęściej w postaci najbardziej pożądanych przez biznes funkcjonalności.

Współpracując z nami manager dbał, aby na review a'la sprintów było obecnych możliwie dużo potencjalnych interesariuszy. Widocz-



MICHAŁ BABTYZEI

m.bartuzel@hncit.nl

Trener i konsultant w firmie BNS IT. Bada i rozwija metody psychologii programowania, pomagające programistom lepiej wykonywać ich pracę. Na co dzień autor zajmuje się zwiększaniem efektywności programistów poprzez szkolenia, warsztaty oraz coachingi i treningi.



ŁUKASZ KORCZYŃSKI

Doświadczony programista JEE, od 6 lat związany z Grupą Nordea. Fascynat technologii, Software Craftmanship i zwinności.

nym efektem takiego działania był fakt, że kierownicy projektu prosili o dostarczenie kolejnych „ładnych” funkcjonalności”. Jednocześnie do inicjatywy Action-30 przyłączyło się kilku programistów z innych zespołów, a zatem nasz marketing wewnętrzny odniósł skutek.

Najbardziej zaskakującą informacją była ta, że „podobno top management zainteresował się Action-30”. Po weryfikacji okazało się, że była to, precyzyjnie rzecz ujmując, plotka. Jednak dla naszej inicjatywy jak najbardziej korzystna. Zrozumieliśmy wtedy, że *Action-30* przebiło się ze swoim komunikatem do świadomości organizacji i zaczęło żyć własnym życiem.

## **Jak to wszystko się skończyło?**

W wyniku ruchów na poziomie korporacji zapadła decyzja o stopniowym wygaszaniu rozwoju systemu e-bankowości w Polsce. Zatem nie udało nam się doprowadzić *Action-30* do końca. Proaktywność zespołu została jednak zauważona, gdyż otrzymali oni zadanie przejęcia ważnych projektów w „nowoczesnych” technologią od zagranicznych partnerów.

To, co przede wszystkim podkreślał zespół, to fakt, że się da. Że można zainicjować i wypromować poważną zmianę w dużej organizacji tworzącej wrażliwe oprogramowanie. Programiści doświadczyli, że mają wpływ na decyzje organizacji i że mogą przebić się ze swoimi technicznymi potrzebami. Czy można chcieć wiecej?

- We proved that SCRUM works inside Nordea
  - We pointed out weaknesses in Netbank codes
  - We tried and proved that new things are possible
  - We proved that we can do nice looking components
  - We are able to sell our inspiration and results to managers
  - We learnt a looooooot
  - We are capable of making those changes!

*Rysunek 9 Podsumowanie Action-30 przygotowane przez zespół*

Tekst został przygotowany za zgodą Nordea Bank AB S.A.  
Oddział w Polsce.

P O L E C A M Y :

ŁÓDŹ / 10-12.07.2017

**TECHNIKI PRACY Z KODEM**  
**SOFTWARE CRAFTSMANSHIP**

1. Software Craftsmanship
2. Formułowanie algorytmu
3. Komentarze
4. Nazewnictwo
5. Upraszczanie metod
6. Komponowanie metod
7. Naturalny Porządek Refaktoryzacji™

WARSZAWA / 12-14.07.2017

**TECHNICAL LEADERSHIP™**  
**ROLA LIDERA TECHNICZNEGO**

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

N A J B L I Ż S Z E   S Z K O L E N I A   W   2 0 1 7   R O K U :

Nowoczesne architektury aplikacji	Łódź	08-09.06.2017	1800,00 PLN
Getting Things Programmed	Łódź	22-23.06.2017	1800,00 PLN
Architektura aplikacji biznesowych	Warszawa	28-30.06.2017	2100,00 PLN
Wzorce projektowe i refaktoryzacja do wzorców	Warszawa	05-07.07.2017	2100,00 PLN
Techniki pracy z kodem	Łódź	10-12.07.2017	2100,00 PLN
Technical Leadership™	Warszawa	12-14.07.2017	2100,00 PLN

CENY NETTO

# BSidesSF 2017 – Pinlock

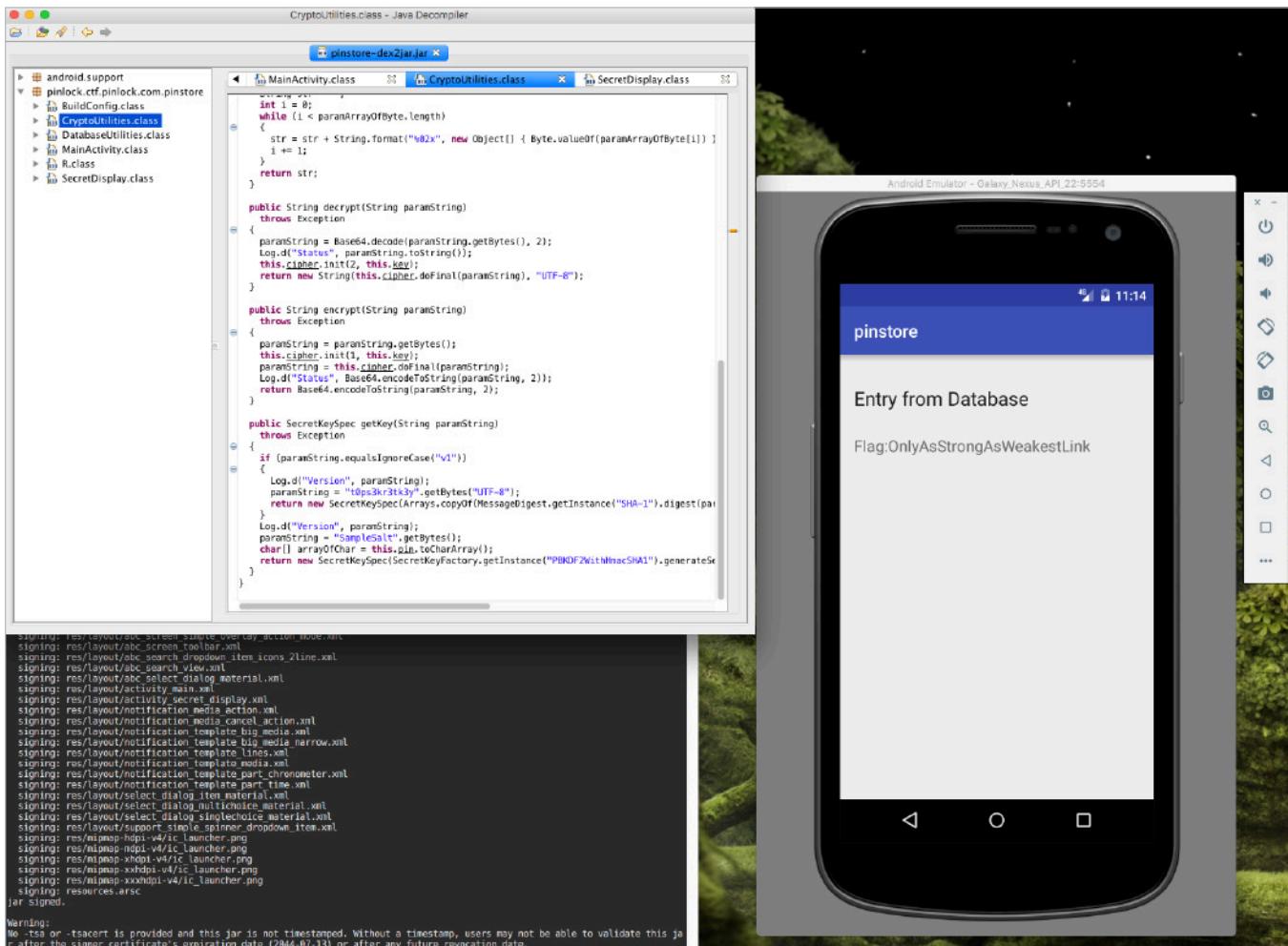
Drugi weekend lutego spowodował, że zacząłem wątpić w regułę „jeden CTF na dwa tygodnie”, bo między 10 a 14 lutego odbywały się aż trzy. W tym CTF towarzyszący konferencji BSides San Francisco 2017.

Zadanie było sporo, o różnym poziomie trudności, we wszystkich chyba najpopularniejszych kategoriach: *crypto*, *forensics*, *misc*, *pwn*, *reverse engineering*, *web*. Co ważne, zadania zazwyczaj faktycznie odpowiadały kategoriom, do których były przyporządkowane (kto chociaż raz przeżył to poirytowanie, kiedy zadanie teoretyczne z kategorii *crypto* okazuje się zgadywanką,

wie, o czym mówię). Kilka zadań było typu *on-site* (możliwych do rozwiązania jedynie, jeśli fizycznie uczestniczyliśmy w konferencji).

Parę dni po CTFie organizatorzy udostępnili na GitHubie wszystkie zadania, wraz z kodem źródłowym, a także samą platformę (<https://github.com/BSidesSF/ctf-2017-release/>). Dzięki temu można ściągnąć interesujące nas zadania (np. omawiane w arty-

CTF	BSides San Francisco 2017 <a href="https://bsidessf.com/">https://bsidessf.com/</a>
Waga CTftime.org	?? ( <a href="https://ctftime.org/event/414">https://ctftime.org/event/414</a> )
Liczba drużyn (z niezerową liczbą punktów)	531
System punktacji zadań	Od 1 (proste) do 666 (trudne) punktów
Liczba zadań	46
Podium	1. dcua (Ukraina) – 6773 pkt. 2. OpenToAll (Świat) – 5178 pkt. 3. scryptos (Japonia) – 5093 pkt.
Zadanie	Pinlock (Mobile RE 150)



kule) bądź też pełne repozytorium i wygenerować obraz Dockera, dzięki czemu na własnym komputerze uruchomimy serwer ze wszystkimi zadaniami, będziemy mogli wrzucać zdobyte flagi itd.

## PINLOCK

W zadaniu otrzymujemy aplikację na platformę Android (plik *pinstore.apk*) oraz krótką instrukcję:

*To pierwsza aplikacja mobilna tych autorów. Próbujej bezpiecznje przechowywać poufne dane bezpośrednio w aplikacji. Czy gdzieś tam może znajdować się flaga?*

Zacznijmy od zainstalowania aplikacji w emulatorze. Proponuję skorzystać z najpopularniejszego rozwiązania, czyli avd z Android Studio (<https://developer.android.com/studio/>). Szczegółowe instrukcje instalacji samego Android Studio, jak i konfiguracji emulatorka avd znajdują się na tej samej stronie. Zakładając, że mamy już skonfigurowane avd, wszystkie potrzebne kroki możemy wykonać z linii komend. Potrzebne narzędzia w Linuksie domyślnie znajdują się w katalogu `~/Android/`, a w Mac OS w `~/Library/Android`. Aby ułatwić sobie pracę, dodajmy odpowiednie podkatalogi do \$PATH:

```
$ export PATH= ~/Library/Android/sdk/tools:$PATH
```

Analogicznie dodajemy następujące podkatalogi `platform-tools` i `build-tools`.

```
$ emulator -list-avds
Galaxy_Nexus_API_22
Nexus_5X_API_25_x86
$ emulator -avd Galaxy_Nexus_API_22 &
$ adb devices
List of devices attached
emulator-5554 device
$ adb install pinstore.apk
[100%] /data/local/tmp/pinstore.apk
pkg: /data/local/tmp/pinstore.apk
Success
```

Uruchamiamy aplikację, pojawia się prośba o podanie PIN. Bez wielkich nadziei spróbowałem 1337 i oczywiście pułko – *Incorrect PIN, try again*. Czas zabrać się za analizę. Potrzebujemy do tego dwóch narzędzi: apktool (<https://ibotpeaches.github.io/Apktool/>), którym zdekodujemy zawartość aplikacji, oraz dex2jar (<https://github.com/pksam/pkb1988/dex2jar>), za pomocą którego przekonwertujemy bajtkod maszyny Dalvik na bajtkod Javy.

```
$ apktool decode --no-src pinstore.apk
...
$ d2j-dex2jar -f pinstore.apk
dex2jar pinstore.apk -> ./pinstore-dex2jar.jar
```

Zobaczmy, co ciekawego znajdziemy w katalogu `pinstore` (czyli w wyniku pracy apktool). Najpierw `assets/README`:

```
v1.0:
- Pin database with hashed pins

v1.1:
- Added AES support for secret

v1.2:
- Derive key from pin
[To-do: switch to the new database]
```

Następnie, jeszcze bardziej interesujący plik, `assets/pinlock.db`, który okazuje się bazą SQLite:

```
$ sqlite3 pinlock.db
SQLite version 3.14.0 2016-07-26 15:17:14
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  pinDB    secretsDBv1    secretsDBv2
sqlite> select * from pinDB;
1|d8531a519b3d4dfbebe0259f90b466a23efc57b
sqlite> select * from secretsDBv1;
1|hcsvUnln5jMdw3GeI4o/
tx85vaEf1PFAKQ3kPsRW2o5rR0a1JE54d0BLkzXPtqB
sqlite> select * from secretsDBv2;
1|Bi528nD1NBcX9BcCC+ZqGQo10z01+GOWSmvxRj7jg1g=
sqlite> select * from android_metadata;
en_US
sqlite>
```

Wygląda na to, że README mówi prawdę i zamiast PINu w tabeli zapisana jest funkcja skrótu (*hash*). Spróbujmy odtworzyć PIN.

## ROZGRZEWA: ZNAJDOWANIE PINU

Zakładając, że PIN zwyczajowo ma jedynie 4 cyfry, znalezienie go metodą *brute-force* nie stanowi żadnego problemu. Wystarczy dowiedzieć się, jakiej funkcji użyć (w tym przypadku jest to SHA-1) i wygenerować *hash* dla wszystkich możliwych PINów (10 tysięcy kombinacji). Ponieważ krótkie hasła składające się z samych cyfr mają wiele zastosowań, jest spora szansa, że ktoś tę pracę wcześniej już za nas wykonał. Spróbujmy szczęścia w bazie <https://crackstation.net> – bingo! Nasz PIN to 7498. Sprawdźmy w aplikacji uruchomionej w emulatorze. Faktycznie, PIN działa, jednak na kolejnym ekranie dostajemy coś, co wygląda jak template – *Here is what the data will look like*.

Czas wrzucić wyprodukowany przez dex2jar plik `pinstore-dex2jar.jar` do dekomplilatora Javy, JD-GUI (<http://jd.benow.ca/>). Szybki rekonesans i już wiemy, że ostatni wpis w README mówi prawdę – aplikacja ma jedynie część kodu pozwalającą na korzystanie z „nowej” bazy, czyli `secretsDBv2` (metoda `CryptoUtilities#getKey()`), ale nadal korzysta jedynie z wersji pierwnej: metoda `DatabaseUtilities#fetchSecret()` ma na stałe wpisany string `SELECT entry FROM secretsDBv1;` a w `SecretDisplay#onCreate()` wywołanie `#decrypt()` odbywa się na obiekcie klasy `CryptoUtilites` stworzonym z na sztywno wpisanym v1. W tym momencie jest już jasne, że string, który pokazała aplikacja po wpisaniu PINu, jest wynikiem dekodowania zawartości tabeli `secretsDBv1`. Naszym celem jest zdekodowanie zawartości `secretsDBv2` (jak widać w metodzie `encrypt()`, dodatkowo należy użyć algorytmu `PBKDF2WithHmacSHA1` zamiast `SHA-1`). Przy okazji możemy potwierdzić, że wprowadzany PIN (który już znamy) faktycznie traktowany jest funkcją `SHA-1` (metoda `getHash` klasy `CryptoUtilities` wywoływana w `MainActivity`).

Mamy dwie opcje rozwiązania: zaimplementować krótki program, który rozszyfruje string, lub zmodyfikować aplikację, by korzystała z nowej bazy. W trakcie CTFa zastosowałem metodę pierwszą, jednak dla celów dydaktycznych warto pokazać obie.

## PODEJŚCIE 1: SKRYPT DESZYFRUJĄCY

Co wiemy? Hasło zakodowane jest w formacie Base64 (na co wskazuje metoda `decrypt()`), a interesujący nas fragment metody `getKey()` wygląda tak:

```
paramString = "SampleSalt".getBytes();
char[] arrayOfChar = this.pin.toCharArray();
return new SecretKeySpec(SecretKeyFactory.getInstance("PBKD
F2WithHmacSHA1").generateSecret(new PBEKeySpec(arrayOfChar,
paramString, 1000, 128)).getEncoded(), "AES");
```

Znamy salt (SampleSalt), znamy pin (7498), ostatnia linia pokazuje, jakich algorytmów (i ich parametrów) musimy użyć. Dokumentacja Javy podpowiada, że argumenty PBEKeySpec to kolejno: password, salt, iterationCount, keyLength.

Rozwiązuając zadanie w trakcie turnieju, korzystałem z Javy, która niestety nie jest moją mocną stroną, a okazało się, że w kilku miejscach to, co działa na Androidzie, nie bardzo chce działać w „zwykłej” Javie. Problemem była konwersja pomiędzy typami: String, byteString i char[ ]. W końcu się udało, jednak postanowiłem „w wolnej chwili” zrobić to lepiej i zaimplementować potrzebny kod w Pythonie. Dokumentacja odpowiednich modułów plus grzebanie na StackOverflow zaowocowały poniższym skryptem:

```
from hashlib import pbkdf2_hmac
from Crypto.Cipher import AES

salt = 'SampleSalt'
pin = '7498'
e = 'Bi528nD1NBcX9BcCC+ZqGQo10z01+GOWSmvxRj7jg1g='.
decode('base64')

k = pbkdf2_hmac('SHA1', pin, salt, 1000, 16)
print AES.new(k, AES.MODE_ECB).decrypt(e)
```

Jak widać, kod jest bardzo podobny do oryginału. Podpowiedź co do trybu pracy dla AES (MODE\_ECB) w pierwotnym kodzie znajduje się w metodzie konstruktora klasy `CryptoUtilities`. Odpowiednikiem `keyLength` w implementacji w Pythonie jest `dklen`, który podawany jest w bajtach, a nie bitach (stąd 16 zamiast 128). Uručhamiemy skrypt i w nagrodę otrzymujemy flagę:

```
$ python solver.py
Flag:OnlyAsStrongAsWeakestLink
```

## PODEJŚCIE 2: MODYFIKACJA APLIKACJI

Zapominamy o poprzednim podrozdziale – spróbujmy zmodyfikować aplikację. Potrzebujemy kolejnego narzędzia – smali (<https://github.com/JesusFreke/smali>), za pomocą którego możemy assemblewać i deassemblewać bajtkod Dalvika:

```
$ bksmali pinstore/classes.dex
```

W wyniku uruchomienia bksmali otrzymujemy katalog `out/`, a w nim, w `pinlock/ctf/pinlock/com/pinstore` zdeasembolowany kod w plikach z rozszerzeniem `*.smali`. W pliku `Secret-Display.smali` odnajdujemy linię:

```
const-string v7, "v1"
```

i zamieniamy `v1` na `v2`.

Natomiast w pliku `DatabaseUtilities.smali` zmieniamy nazwę bazy z `secretsDBv1` na `secretsDBv2`, w linii:

```
const-string v1, "SELECT entry FROM secretsDBv1"
```

Tworzymy nowy plik `classes.dex` ze zmodyfikowanym kodem:

```
$ smali out -o classes.dex
```

A następnie kopujemy ten plik do katalogu `pinstore`, gdzie znajduje się reszta aplikacji, po czym możemy stworzyć nową wersję:

```
$ apktool b -o newpinstore.apk pinstore
```

Jeśli spróbujmy ją zainstalować za pomocą adb (analogicznie jak na początku artykułu instalowaliśmy oryginalną wersję), operacja się nie powiedzie, bo aplikacja jest niepodpisana. Podpiszmy ją więc (keytool jest częścią instalacji SDK Javy, a apksigner i zipalign – Android Studio). W przypadku wątpliwości odsyłam do dokumentacji: <https://developer.android.com/studio/publish/app-signing.html>, można też oczywiście skorzystać z GUI Android Studio. Stworzenie kluczy i podpisanie aplikacji w linii komend odbywa się tak:

```
$ keytool -genkeypair -v -alias ctf -keyalg RSA -keysize 2048
-validity 999 -keystore ctf-key.jks
$ zipalign -v -p 4 newpinstore.apk newpinstore-aligned.apk
$ apksigner sign --ks ctf-key.jks --out newpinstore-ready.apk
newpinstore-aligned.apk
```

Kolejna próba instalacji (tym razem używamy oczywiście nowego pliku `newpinstore-ready.apk`)... kolejne niepowodzenie. Powodem jest to, że próbujemy zaktualizować aplikację, a nie podnieśliśmy numeru wersji. Proponuję odinstalować istniejącą aplikację i zainstalować nową wersję. Po uruchomieniu aplikacji i wpisaniu PINu na kolejnym ekranie pojawia się flaga:

Flag:OnlyAsStrongAsWeakestLink

## PODSUMOWANIE

Opisane zadanie nie było szczególnie trudne, mimo to praca nad rozwiązaniem dała mi mnóstwo satysfakcji. Temat analizy, a tym bardziej modyfikacji aplikacji na Androide był dla mnie zupełnie nowy, bo zadania wykorzystujące tę platformę pojawiają się na CTFach stosunkowo rzadko.

Jarosław Górný



### Dragon Sector

Rozwiązanie zadania `Pinlock` zostało nadesłane przez Jarosława Górnego, grającego gościnnie z Dragon Sector – jedną z polskich drużyn CTFowych: <http://www.dragonsector.pl/>.

*Włamiemy się legalnie do Twojego systemu,  
zanim nielegalnie zrobią to inni*

BEZPIECZEŃSTWO SYSTEMÓW IT  
**SECURITUM**



# BEZPIECZEŃSTWA

aplikacje webowe | certyfikowani  
aplikacje mobilne | pentesterzy  
sieci | OSCE | OSCP  
socjotechnika | CISSP | CEH

[ PRZESZŁO 150 TESTÓW BEZPIECZEŃSTWA  
REALIZOWANYCH ROCZNIE ]

[SECURITUM.PL/OFERTA/](http://SECURITUM.PL/OFERTA/)

# angielski dla programistów. Lekcja 2

Przedstawiam drugą lekcję minikursu angielskiego dla programistów. Odpowiedzi do ćwiczeń zamieściłem w Internecie, aby nikogo nie kusiło do nich zaglądać podczas nauki :)

## *Server-side scripting*

**Web browsers** communicate with **web servers** using the HyperText Transport Protocol (HTTP). When you **click a link** on a **web page**, **submit a form**, or run a search, an **HTTP request** is sent from your browser to the target server. The request includes a URL **identifying** the affected **resource**, a **method** that defines the required **action** (for example to **get**, **delete**, or **post** the resource), and may include additional information **encoded** in URL parameters (the field-value pairs **sent via a query string**), as **POST data** (data sent by the HTTP POST method), or in associated **cookies**.

Most major websites use some kind of server-side technology to dynamically display different data as required. Displaying all of these using completely different **static pages** would be completely **inefficient**, so instead such sites display **static templates** (built using HTML, CSS, and JavaScript), and then dynamically **update** the data displayed inside those templates when needed.

A dynamic website is one where some of the **response content** is generated dynamically only when needed. On a dynamic website HTML pages are normally created by inserting data from a **database** into **placeholders** in HTML templates (this is a much more efficient way of storing large amounts of content than using **static websites**). A **dynamic site** can return different data for a URL based on information provided by the user or stored preferences, and can perform other operations as part of returning a response (e.g. sending notifications).

Most of the code to support a dynamic website must **run on the server**. Creating this code is known as "**server-side programming**" (or sometimes "back-end scripting").

Server-side code can be written in any number of programming languages — examples of popular **server-side web languages** include PHP, Python, Ruby and C#. Server-side code has **full access** to the server **operating system** and the developer can choose what programming language (and specific version) they wish to use.

## Ćwiczenia

1. Dopusz słowa lub wyrażenia z lewej kolumny do słów lub wyrażeń z prawej. Potrafisz je wszystkie przetłumaczyć na język polski?

- |              |          |
|--------------|----------|
| 1. full      | request  |
| 2. dynamic   | content  |
| 3. click     | template |
| 4. HTTP      | system   |
| 5. query     | browser  |
| 6. response  | a form   |
| 7. operating | access   |
| 8. static    | string   |
| 9. web       | website  |
| 10. submit   | a link   |

2. Odpowiedz na poniższe pytania. Możesz cytować tekst. Powtarzaj, aż będziesz w stanie udzielić odpowiedzi bez patrzenia na tekst.

1. How do web browsers communicate with the web server?
2. What happens when you click a link on a web page?
3. What does an HTTP request include?
4. What is a dynamic website?
5. How are pages on a dynamic website created?
6. What is server-side scripting?

3. Dokończ zdania zaczynające się od wyrażenia *On a dynamic website* wg podanego wzoru.

Przykład: *On a dynamic website – web pages.* -> *On a dynamic website web pages are generated dynamically.*

1. content dynamically
2. content when needed
3. responses generated
4. data into placeholders
5. data returned
6. data into database
7. data based on preferences
8. different data for a URL

4. Jeszcze raz wykonaj ćwiczenie 3., ale ustnie i starając się nie zaglądać do tekstu ani odpowiedzi. Powtarzaj ćwiczenie dotąd, aż nie będziesz mieć trudności z tworzeniem tych zdań.

## Odpowiedzi

Odpowiedzi do ćwiczeń znajdują się na stronie: <https://goo.gl/pqIQ5u>. Źródła tekstu CC-BY-SA 2.5: <https://goo.gl/qAXK3G>, <https://goo.gl/1Cdax>

## Słownik

- web browser – przeglądarka internetowa
- web server – serwer sieciowy
- click a link – kliknąć odnośnik
- web page – strona internetowa
- submit a form – zatwierdzić formularz
- HTTP request – żądanie HTTP
- identify – identyfikować, określić
- resource – zasób
- method – metoda
- action – czynność, akcja
- get – pobrać
- delete – usunąć
- post – wysłać
- encoded in – zakodowany w
- send via – wysyłać za pośrednictwem
- query string – łańcuch zapytania
- POST data – dane POST
- cookies – ciasteczka
- inefficient – niefektywny
- static template – statyczny szablon
- update – aktualizować
- response content – treść odpowiedzi
- database – baza danych
- placeholder – element zastępczy
- static website – statyczna witryna internetowa
- dynamic site – dynamiczna witryna internetowa
- run on the server – działać na serwerze
- server-side programming – programowanie serwerowe
- server-side web language – serwerowy język programowania
- full access – pełny dostęp
- operating system – system operacyjny



**ŁUKASZ PIWKO**

[piwko.lukas@gmail.com](mailto:piwko.lukas@gmail.com)

Tłumacz angielskiej i francuskiej literatury programistycznej z ok. 70 książkami na koncie, nauczyciel, wykładowca i maniak technologii programistycznych.

# ELITARNI

30 marca 2017



**STADION  
LEGIA WARSZAWA**



**INTERVIEW  
IN THE SKY**



[www.targi.praca.pl](http://www.targi.praca.pl)

Partnerzy



Patroni medialni



DZIENNIK  
GAZETA PRAWNA

dla Studenta.pl

# WordPress i Joomla! Zabezpieczanie i ratowanie stron WWW



**S**tare chińskie przysłowie, a raczej przeświadczenie, brzmi: „Obyś żył w ciekawych czasach”. Co te słowa mogą mieć wspólnego z najnowszą książką autorstwa Pawła Frankowskiego? Wbrew pozorom bardzo dużo. Czasy, w jakich przyszło nam żyć i funkcjonować, są bardzo niespokojne. Nieświadomi zagrożeń współczesnego Internetu klienci szukają osób mogących zrobić im stronę jak najniższym kosztem, jednak nie zastanawiają się nad kwestią jej bezpieczeństwa, ba – nawet jak o niej słyszą, to ją ignorują. Autor w swojej najnowszej pozycji porusza właśnie tę tematykę. Uwrażliwia czytelnika w kwestii aktualizacji posiadanych serwisów, zwraca uwagę na zagrożenia płynące z braku opieki nad stroną, podaje, jakie wymierne straty mogą generować te zaniedbania, a także – co najcenniejsze w tej pozycji – sugeruje różne sposoby obrony przed atakami.

Autor na około 240 stronach przekazuje, wbrew pozorom, ogromną ilość wiedzy. Nie ogranicza się w żaden sposób tylko do obu wymienionych w tytule systemów, sugerując także, w miarę możliwości, uniwersalne rozwiązania. Konstrukcja książki jest bardzo przemyślana, praktycznie można zapoznać się z rozdziałami w dowolnej kolejności, a osoby nie mające styczności z którymkolwiek z omawianych systemów mogą po prostu pominąć rozdział w danej chwili im niepotrzebny (choć osobiście polecam – choćby pobicie – jednak zaznajomić się z nim, by móc okaże się źródłem ciekawych inspiracji). Taka konstrukcja wymusiła niestety lekkie powtórki, jednak nie są one w żaden sposób uciążliwe, a umożliwiają zapoznanie się z treścią w miarę potrzeb. Styl przekazania całej tej złożonej treści jest bardzo przystępny. Czytając słowa autora, nie ma się wrażenia przebywania z wykładowcą zanudzającym tematem poczatkującego adepta danego przedmiotu, jednak widać ogromną wiedzę i lata praktyki (Frankowski jest znany i aktywnym członkiem polskiej społeczności Joomla!).

Wracając do samej książki, zawiera ona sześć rozdziałów:

- » Rozdział 1: *Strony internetowe w obliczu zagrożeń* – w rozdziale tym autor omawia przyczyny atakowania stron, ich skutki, potencjalne źródła ataków, porusza także kwestie prawne dotyczące administratorów. Zapoznając się z treścią tej części książki, można odnieść wrażenie, że w brutalny sposób obdziela Internet z magicznego kocyka „fałszywego poczucia bezpieczeństwa”.
- » Rozdział 2: *Kopia zapasowa* – szkoda, że tych prawie 40 stron nie można wypiąć z książki i nosić ze sobą – wiele osób może w końcu zrozumiałoby, jak ważne jest wykonywanie kopii zapasowych. Autor w tym rozdziale opisuje praktycznie wszystko, co wiąże się z wykonywaniem kopii zapasowych od teorii do praktyki – czyli najbardziej przydatnie :)
- » Rozdział 3: *Pierwsza linia obrony* – jeśli chcielibyśmy mieć nieco spokojniejsze noce, utrudnić życie domorosłym „hakierom”, to ten rozdział jest w sam raz dla nas – znajdziemy w nim porady od najprostszych do bardziej złożonych, dowiemy się, jak ukryć informacje o używanej platformie, i poznamy ciekawe sposoby uprzyskrzenia się „tym złym”.
- » Rozdział 4: *Jak zabezpieczyć WordPressa* – jest to pierwszy z rozdziałów, które można pominąć, jeśli nie ma się styczności z danym systemem. Jednak sam, będąc wielkim fanem Joomla!, z nieukrywaną przyjemnością przeczytałem o sposobach walki z atakującymi w konkurencyjnym „obozie”.
- » Rozdział 5: *Jak zabezpieczyć system Joomla!* – ten rozdział, tak jak poprzedni, można pominąć, gdy nie używamy Joomla!. Będąc użytkownikiem tego systemu od czasów fork'a z Mamboo, był to najbardziej interesujący dla mnie element książki. Treści tutaj zamieszczone wystarczą dla większości administratorów tego CMS'a do uczynienia go odporniejszym na zautomatyzowane ataki.
- » Rozdział 6: *Oczyszczanie strony po włamaniu* – tak, zdarza się to każdemu. Nie znam osobiście administratora, który nie musiałby, choć raz w życiu, walczyć z zainfekowaną stroną klienta. Często jest to walka z wiatrakami, bo okazuje się (o czym pisze też Frankowski wielokrotnie), że agencja, która wykonała stronę, umywa ręce od jej aktualizowania, zrzucając wszystko na barki najczęściej nieświadomego niczego właściciela strony. Autor bardzo dogłębnie analizuje tę tematykę, przedstawia sposoby ułatwiające wykonanie tego niewdzięcznego zadania, zwracając uwagę na wiele powtarzających się detali.

Lektura książki może przytłaczać ilością materiału i wiedzy, jednak jest to wiedza bardzo cenna. Nie wiem, czy to jakieś magiczne zrządzenie losu, ale jak tylko dowiedziałem się o powstawaniu tej książki, zostałem poproszony o pomoc przy naprawie pewnego serwisu opartego na WordPress nękanego notorycznymi włamaniami. Żeby było ciekawiej, będąc w połowie rozdziału o zabezpieczaniu Joomla!, zadzwonił kolega z prośbą o pomoc w oczyszczaniu zainfekowanej Joomla! z rodziną 1.5 (data instalacji: połowa grudnia 2008 r. i nigdy nie aktualizowana!), na której bazowała strona dużego podmiotu. Te dwa przypadkowe zdarzenia umożliwiły, poza teoretycznym zapoznaniem się z poradami autora, wdrożyć część jego sugestii jako uzupełnienie własnych rozwiązań, a także porównać i uzupełnić własną listę poszukiwań śladów włamań oraz uświadomić żyjących w blogie nieświadomości właścicielom o konieczności dbania o serwis i regularnego jego aktualniania (mających nastawienie „nie ma o tym książek, to temat nie istnieje”).

Oczywiście osoby bardziej zaznajomione z tą tematyką, będące praktykami, mające na koncie wiele oczyszczonych serwisów i interesujące się bezpieczeństwem wykorzystywanych narzędzi z większością wiedzy przekazanej przez autora będzie zaznajomiona, jednak każdy znajdzie coś ciekawego i – bardzo prawdopodobne – nowego dla siebie. Samemu znając tę problematykę z wieloletnią praktyką, czytałem książki z nieskrywaną przyjemnością, a niektóre pomysły autora zainspirowały do uzupełnienia własnych unikalnych rozwiązań.

Mówiąc krótko: w mojej opinii omawiana tu książka jest jak pyszny tort ze smacznymi niespodziankami w środku i taką symboliczną wisienką na jego szczytce pod postacią dodatków znajdujących się na końcu książki, a szczególnie jednego – co to za dodatek, jaka jest jego przydatność, tego czytelnik dowie się sam po lekturze *WordPress i Joomla! Zabezpieczanie i ratowanie stron WWW*, do której z całego serca namawiam.

Mariusz "maryush" Witkowski

Tytuł:	<i>WordPress i Joomla! Zabezpieczanie i ratowanie stron WWW</i>
Autor:	Paweł Frankowski
Stron:	248
Wydawnictwo:	Helion
Data wydania:	2017-02-14

# UCZENIE FAKTÓW W EPOCE GOOGLE NIE MA SENSU

**MY UCZYMY  
MYŚLEĆ**

[ SZKOLENIA PROFILOWANE  
I WARSZTATY EKSPERCKIE ]





Mały zespół z wielkimi możliwościami

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

# CONTMAN

Normal developer 7000 - 9000 + VAT Senior developer 9000 - 13000 + VAT

## CZŁOWIEK POMÓŻ!

### SZUKAMY NAJLEPSZYCH PROGRAMISTÓW

OD  
ZARAZ!

Sprawdź oferty pracy w Contman:  
[contman.pl/kariera](http://contman.pl/kariera)



#### U MNIE DZIAŁA

CR, testy automatyczne  
i współtworzenie produktu

#### Z WIDOKIEM NA WARTĘ

Poznaj korzyści płynące  
z pracy w centrum Poznania

#### CZY DZIADEK MOŻE BYĆ SM?

Spotkaj się z naszym  
Scrum Masterem i oceń sam

#### A CO PO PRACY?

Multisport, pizza, squash,  
wspinaczka, siłownia...