Zaawansowane Techniki Programowania

Przetwarzanie Obrazów

Projekt

Prowadzący: dr hab. inż. Mirosław Kordos, prof. ATH
Autor: Rafał Klinowski
Informatyka II st., sem. 1., gr. A

Spis treści

1. Temat projektu

Projekt nosi nazwę “Przetwarzanie Obrazów”. Jest to aplikacja webowa, stworzona przy pomocy technologii ASP.NET, w której użytkownicy mogą w różny sposób edytować przesłane przez nich obrazy.

Przykładami działań, jakie można uzyskać przy pomocy aplikacji, są:

- Skala szarości
- Filtry obrazu: filtr Laplace, filtr górnoprzepustowy
- Zmiana jasności obrazu
- Zmiana kontrastu obrazu
- Konwersja obrazu na obraz binarny

Z powodu dużej otwartości projektu łatwe jest dodawanie nowych algorytmów związanych z przetwarzaniem danych.

Dostęp do projektu odbywa się poprzez stronę internetową – aplikację ASP.NET. Projekt udostępnia również API, które jest wewnętrznie wykorzystywane w celu przetwarzania obrazów.

Projekt został zaimplementowany w wersji środowiska .NET 6.

2. Specyfikacja zewnętrzna i wewnętrzna

2.1. Specyfikacja zewnętrzna

Użytkownik postępuje zgodnie z instrukcjami na ekranie dotyczącymi przesłania pliku.

Na stronie do przesyłania znajduje się formularz – użytkownik po przesłaniu pliku i rozpoczęciu przetwarzania czeka na otrzymanie wyniku.

W pierwotnej wersji, program obsługuje jedynie obrazy w formacie PNG, możliwe jest jednak dodanie obsługi również innych formatów.

Gdy wynik jest gotowy i zostanie odebrany od API, zostanie on automatycznie wyświetlony na ekran po przekierowaniu na odpowiednią podstronę.

Rysunek 1. Strona główna aplikacji internetowej. Większość styli jest domyślna.

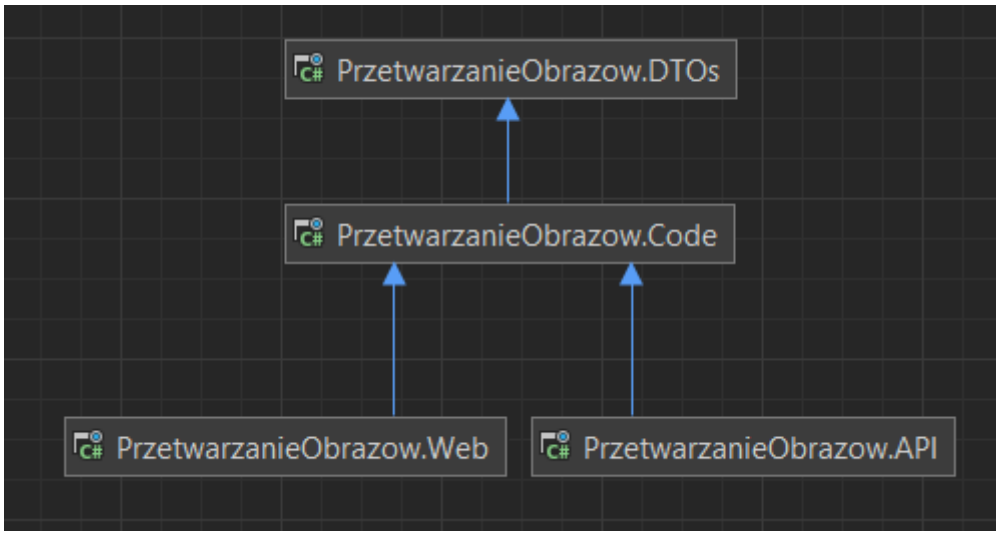
Rysunek 2. Podstrona prezentująca wynik wybranego przez użytkownika algorytmu.

2.2. Specyfikacja wewnętrzna

Aplikacja składa się z kilku połączonych ze sobą projektów:

- **PrzetwarzanieObrazow.Code** – projekt zawiera implementacje poszczególnych algorytmów, sposoby ich wywoływania oraz inne niezbędne klasy i struktury danych
- **PrzetwarzanieObrazow.API** – jest to projekt odpowiedzialny za wykorzystywane w programie API, obsługuje zapytania, wywołuje odpowiednie algorytmy i zwraca wyniki w odpowiednim formacie
- **PrzetwarzanieObrazow.DTOs** – zawiera klasę służącą do przesyłu danych między aplikacją webową a API, projekt jest wspólny dla obu tych projektów

- **PrzetwarzaniaObrazow.Web** – zawiera webowy interfejs użytkownika, pobieranie danych (obrazu) od użytkownika oraz wysyłanie poszczególnych zapytań do API



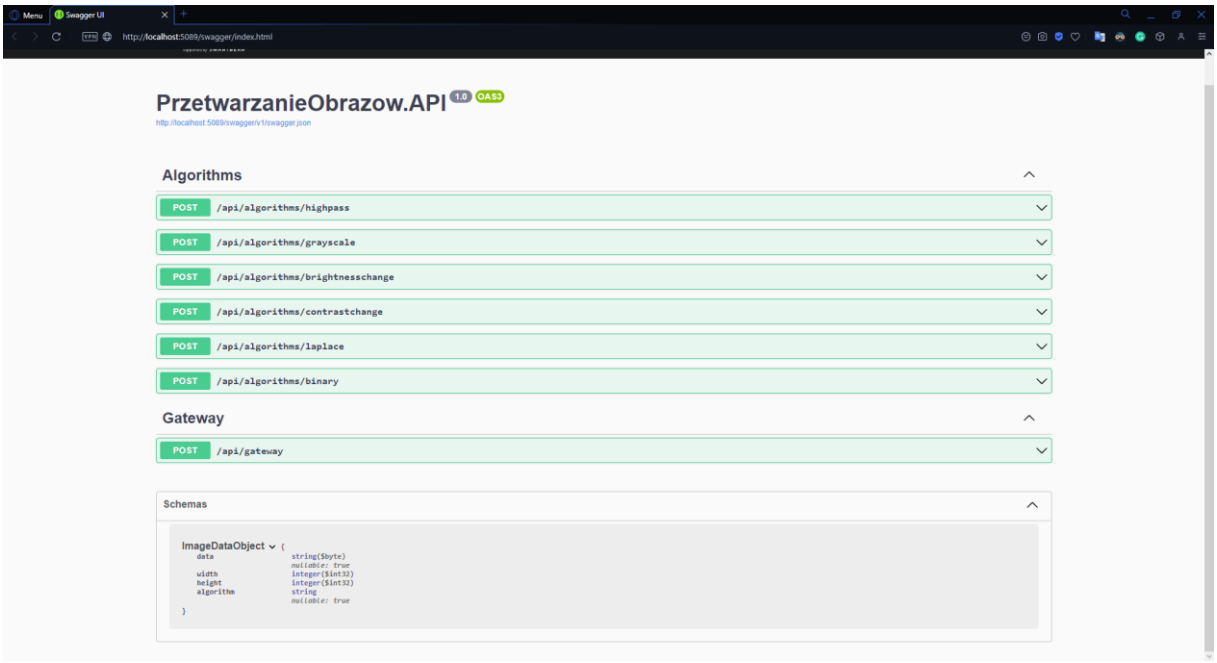
Rysunek 3. Schemat połączenia projektów – projekt PrzetwarzanieObrazow.DTOs jest zależnością dla wszystkich pozostałych. Projekty API i Web są niezależne.

W implementacji wykorzystano następujące wzorce projektowe:

- Budowniczy („builder”) – obsługuje konfigurację poszczególnych algorytmów w taki sposób, że z perspektywy projektu wywołującego algorytmy wymagane jest jedynie przekazanie obrazu wejściowego, a sam ustawia wszystkie niezbędne parametry i odpowiednio łączy funkcjonalność algorytmu z m.in. maskami
- Serializacja („serialization”) – umożliwia zapisanie stanu wewnętrznego (wejścia, konfiguracji, wyjścia) dla każdego z algorytmów do pliku tekstowego, po modyfikacjach również np. do bazy danych
- Data Transfer Object („DTO”) – pozwala na prostą komunikację między kilkoma projektami za pomocą wspólnych obiektów, w przypadku aplikacji są to obiekty zawierające dane i informacje o wywoływanym algorytmie
- Singleton – pojedynczy obiekt w całej aplikacji, w przypadku projektu jest to klasa Settings, zawierająca m.in. adres URL API oraz port połączenia
- API Gateway – definiuje jeden punkt wejściowy, wspólny dla wszystkich zapytań do API, a następnie sam odpowiednio przekierowuje zapytania do odpowiednich węzłów końcowych („endpointów”); z punktu widzenia aplikacji wysyłającej zapytania konieczne jest więc wysłanie odpowiednio przygotowanego zapytania w jedno, stałe miejsce

Interfejs użytkownika został stworzony przy pomocy „Razor Pages”, a więc stron stworzonych przy pomocy HTML+CSS+JavaScript ale z obsługą kodu C#. Strony takie umożliwiają np. dynamiczne wyświetlanie danych, prostą obsługę formularzy czy warunkowe wyświetlanie niektórych danych przy pomocy instrukcji ‘if’.

W programie każdy z algorytmów jest zaimplementowany jako osobna usługa, dostępna za pomocą API – każda niezależnie od siebie – a więc projekt implementuje architekturę WebServices. Poza dostępem do algorytmów w aplikacji, możliwy jest bezpośredni dostęp do poszczególnych usług za pomocą odpowiednio utworzonych zapytań HTTP.



Rysunek 4. Dokumentacja API. Z punktu aplikacji korzystamy tylko z endpointu /api/gateway – pozostałe endpointy są obsługiwane bezpośrednio przez projekt API.

Zastosowanie platformy .NET 6, posiadającej wieloplatformowe możliwości, umożliwia uruchomienie projektu-serwera na serwerze zdalnym (np. Azure), w tym na maszynach z systemem operacyjnym Linux. Jednak w ramach projektu, o ile nie jest to konieczne, proces ten nie będzie wykonywany, a testy będą przeprowadzanie na serwerze lokalnym.

3. Kluczowe fragmenty kodu

3.1. Implementacja poszczególnych algorytmów

Podstawą dla wszystkich klas poszczególnych algorytmów jest abstrakcyjna klasa `ImageAlgorithm`, zawierająca niezbędne właściwości, konstruktor oraz metodę `Process` przetwarzającą obraz wejściowy i zwracającą wynik.

```
public abstract class ImageAlgorithm
{
   public Bitmap InputImage { get; set; }
   public int Width { get; set; }
   public int Height { get; set; }

   public Bitmap OutputImage { get; set; }
   public IAlgorithmMask Mask { get; set; }

   public abstract Bitmap Process();

   public ImageAlgorithm(Bitmap inputImage, int width, int height)
   {
      InputImage = inputImage;
      Width = width;
      Height = height;

      OutputImage = new Bitmap(Width, Height);
   }
}
```

Rysunek 5. Kod źródłowy klasy bazowej.

Klasy, które implementują powyższą abstrakcyjną klasę, wywołują bazowy konstruktor i nadpisują funkcjonalność metody przetwarzającej.

```
public class ImageToBinary : ImageAlgorithm
{
   public ImageToBinary(Bitmap inputImage, int width, int height) :
base(inputImage, width, height)
   {
   }

   public override Bitmap Process()
   {
      const double threshold = 0.50;

      for (int i = 0; i < InputImage.Width; i++)
      {
         for (int j = 0; j < InputImage.Height; j++)
         {
            var pixel = InputImage.GetPixel(i, j);
            var grayScale = (pixel.R + pixel.G + pixel.B) / 3;

            if (grayScale > threshold * 256)
            {
               OutputImage.SetPixel(i, j, Color.White);
            }
            else
            {
               OutputImage.SetPixel(i, j, Color.Black);
            }
         }
      }

      return OutputImage;
   }
}
```

Rysunek 6. Przykład klasy pochodnej na przykładzie algorytmu konwersji obrazu do postaci binarnej.

Korzyścią z takiej implementacji hierarchii klas jest to, że wszystkie algorytmy można wywoływać w taki sam sposób, wykonując metodę `Process` na dowolnym obiekcie klasy pochodnej.

3.2. Wzorzec „budowniczy”

Wzorzec ten polega na utworzeniu klasy (lub wielu takich klas), których zadaniem jest tworzenie bardziej skomplikowanych obiektów, na przykład obiektów zawierających w sobie właściwości będące innymi, generycznymi obiektami.

W przypadku projektu Przetwarzanie Obrazów, jest to klasa `ImageFilterBuilder` zawierająca odpowiednie metody do budowania odpowiednio skonfigurowanych wersji algorytmów.

```
public ImageAlgorithm BuildLaplaceFilterAlgorithm(Bitmap inputImage)
{
   var laplaceFilter = new LaplaceFilter(inputImage, inputImage.Width,
inputImage.Height);
   laplaceFilter.Mask = new LaplaceMask();

   return laplaceFilter;
}
```

Rysunek 7. Metoda tworząca obiekt wykonujący algorytm „Filtr Laplace” dla danego obrazu wejściowego. Warto zauważyć, że każda metoda klasy „builder” zwraca typ bazowy ImageAlgorithm.

3.3. Wzorzec „API Gateway”

API Gateway to wzorzec, który przekierowuje wszystkie zapytania przychodzące do serwera do poszczególnych, odpowiednich punktów końcowych („endpoint”) odpowiednich mikroserwisów. W wyniku działania tego wzorca projektowego, klient (w przypadku niniejszego projektu – aplikacja internetowa) wysyła wszystkie zapytania na jeden, stały adres, a API Gateway zajmuje się odpowiednim ich przekierowywaniem i zwracaniem poprawnych wyników.

Początkowo w projekcie wykorzystano zewnętrzne narzędzie służące do konfiguracji takiej bramki, jednak ostatecznie implementacja została wykonana „ręcznie”.

```
[Route("api/gateway")]
[ApiController]
public class GatewayController : ControllerBase
{
    [HttpPost]
    public async Task<IActionResult> Post([FromBody]ImageDataObject obj)
    {
        string algorithmName = obj.Algorithm;

        // Wywołanie odpowiedniego algorytmu.
        // Wysłanie zapytania do /api/algorithms/[nazwa algorytmu].
        using (var client = new HttpClient())
        {
            var content = new
StringContent(System.Text.Json.JsonSerializer.Serialize(obj),
Encoding.UTF8, "application/json");

            var response = await
client.PostAsync($"{Settings.Instance.ApplicationUrl}/api/algorithms/{algor
ithmName}", content);

            string result = await response.Content.ReadAsStringAsync();

            return Ok(result);
        }
    }
}
```

Rysunek 8. Kod źródłowy klasy GatewayController, będący implementacją wzorca API Gateway. Wykonuje on zapytania do poszczególnych endpointów, zdefiniowanych w klasie AlgorithmsController.

3.4. Data Transfer Object („DTO”)

DTO to specjalny typ obiektu używany do transferu danych pomiędzy aplikacjami, bądź – w naszym przypadku – między aplikacją a API. Obiekty takie są uniwersalne i interpretowane przez każdy projekt, który z nich korzysta, w taki sam sposób. Ponadto, obiekty takie muszą być łatwe do serializacji, zazwyczaj w pliku typu JSON czy XML.

W Przetwarzaniu Obrazów, projekt zawierający DTO jest implementowany przez każdy pozostały projekt.

```
public class ImageDataObject
{
    public byte[] Data { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public string Algorithm { get; set; }
}
```

Rysunek 9. DTO implementowany w aplikacji – zawiera dane obrazu (nie ma znaczenia, czy wejściowego, czy wyjściowego) oraz typ wywołanego algorytmu.

3.5. Wysyłanie zapytań do API

Z poziomu aplikacji webowej, zapytania do aplikacji API wysyłane są poprzez kod obsługujący formularz. Pobierane są wtedy dane wprowadzone przez użytkownika – wybrany obraz oraz typ algorytmu – który pakowany jest jako DTO i przekazywany do API Gateway.

```
<form method="post" enctype="multipart/form-data">
    <div>
        <label for="fileUpload">Wybierz obraz PNG (na ten moment inne
formaty nie są obsługiwane), typ filtru do nałożenia i prześlij obraz. Po
chwili na ekranie zostanie wyświetlony wynik.</label>
        <br/>
        <input type="file" id="fileUpload" name="fileUpload"
accept="image/*" required>
        <br/>
        <select name="algorithm">
            <option value="highpass">High Pass Filter</option>
            <option value="grayscale">Grayscale</option>
            <option value="contrastchange">Contrast Change</option>
            <option value="brightnesschange">Brightness Change</option>
            <option value="laplace">Laplace Filter</option>
            <option value="binary">Image to binary</option>
        </select>
    </div>
    <button type="submit">Prześlij</button>
</form>
```

Rysunek 10. Formularz napisany w CSHTML.

```
public async Task<IActionResult> OnPostAsync(IFormFile fileUpload)
{
   Bitmap bitmap;

   using (var memoryStream = new MemoryStream())
   {
      await fileUpload.CopyToAsync(memoryStream);
      byte[] fileBytes = memoryStream.ToArray();

      bitmap = new Bitmap(new MemoryStream(fileBytes));
   }

   string algorithm = Request.Form["algorithm"];

   var dto = new ImageDataObject()
   {
      Width = bitmap.Width,
      Height = bitmap.Height,
      Algorithm = algorithm
   };

   using (var memoryStream = new MemoryStream())
   {
      bitmap.Save(memoryStream, System.Drawing.Imaging.ImageFormat.Png);
      dto.Data = memoryStream.ToArray();
   }

   using (var client = new HttpClient())
   {
      // Wysyłanie DTO do API.
      var content = new
StringContent(System.Text.Json.JsonSerializer.Serialize(dto),
Encoding.UTF8, "application/json");
      var response = await
client.PostAsync("http://localhost:5089/api/gateway", content);

      var resultString = await response.Content.ReadAsStringAsync();
      var resultParsed =
System.Text.Json.JsonSerializer.Deserialize<ImageDataObject>(resultString);

      Result.ImageResult = resultParsed;
      return Redirect("/result");
   }
}
```

Rysunek 11. Obsługa formularza w C# - wysłanie danych i odebranie wyniku.

3.6. Wyświetlanie danych na interfejsie użytkownika

Po uzyskaniu wyniku algorytmu z API w kroku wyżej następuje przekierowanie na podstronę „Result”, wyświetlającą otrzymane wyniki (przekonwertowane z tablicy bajtów na obraz) na ekran jako obraz („img”).

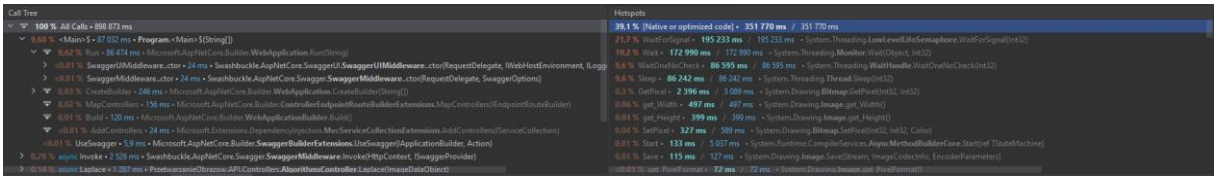
```
@page
@model Result

<h1>Wynik algorytmu</h1>

<p>API zwróciło następujący wynik:</p>

<p>Algorytm: @Model.CurrentImageResult.Algorithm</p>
<p>Szerokość: @Model.CurrentImageResult.Width</p>
<p>Wysokość: @Model.CurrentImageResult.Height</p>

<p>
    @{
        var base64 = Convert.ToBase64String(Model.CurrentImageResult.Data);
        var imageSource = String.Format("{0}",
base64);
    }
    <img src="@imageSource" alt="Wynik algorytmu" />
</p>
```

Rysunek 12. Kod źródłowy podstrony Result – wykorzystanie CSHTML do konwersji obrazu wyjściowego z DTO na źródło znacznika „img”.

4. Optymalizacja pamięci i CPU

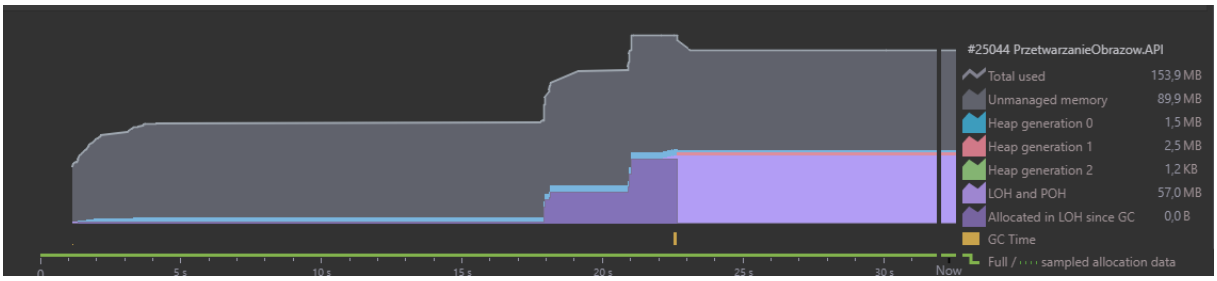
W pierwszym stadium realizacji projektu wykonano proste testy dotyczące optymalizacji pamięciowej i wydajnościowej projektu. Zbadano w ten sposób mało optymalne algorytmy wykorzystane w projekcie, w celu identyfikacji możliwych do poprawienia fragmentów kodu.

Na początku zaobserwowano, że algorytmy High Pass Filter i Laplace Filter są znacznie wolniejsze od pozostałych, dla takich samych danych testowych. Następnie przeprowadzono analizę za pomocą narzędzi dostępnych w środowisku JetBrains Rider – dotTrace i dotMemory.

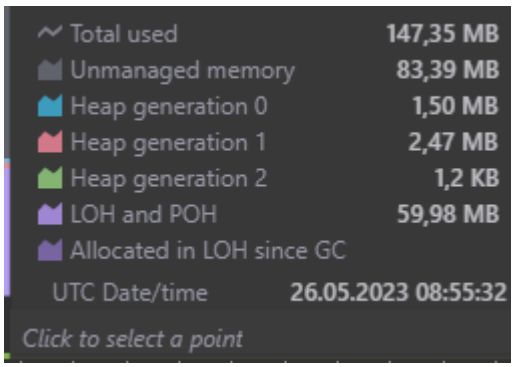


Rysunek 13. Wyniki z dotTrace dla programu API (zajmującego się przetwarzaniem obrazów). Widać m.in. problemy z nieoptymalnym przepisywaniem obrazu wejściowego na wyjście oraz z dostępem do bitmapy.

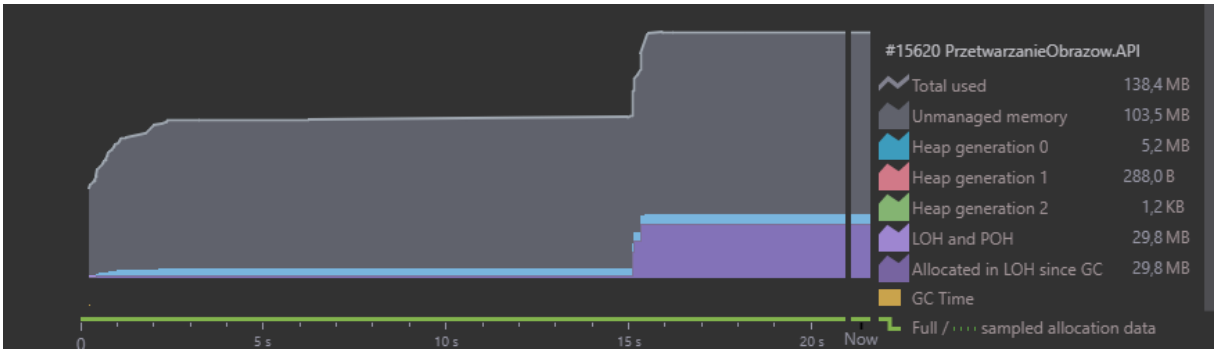
Uwaga: w projekcie Web nie ma sensu wykonywać takiej analizy – projekt jest odpowiedzialny jedynie za pobranie danych od użytkownika, oraz asynchroniczne wysłanie i odebranie wyników od API. Skupimy się więc na projekcie API oraz ściśle z nim powiązanym Code.



Rysunek 14. Wyniki z dotMemory po wykonaniu algorytmu High Pass Filter. Widzimy, że ilość pamięci użytej przez program wzrasta z ok. 80MB do ponad 160MB, a następnie spada do około 150MB. Znacznie zwiększa się również rozmiar Large Object Heap (LOH).



Rysunek 15. Ilość wykorzystanej pamięci minutę po wykonaniu algorytmu – nadal duża.



Rysunek 16. Wyniki dla algorytmu Image to Binary dla takiego samego obrazu. Nadal spore wartości, jednak znacznie mniejsze niż dla poprzedniego.

Można więc zauważyć następujące problemy:

- Alokowanie dużych obiektów typu `Bitmap`, co zajmuje sporo czasu i wymaga dużej ilości pamięci
- Wolne uzyskiwanie dostępu pojedynczych pikseli metodami `GetPixel` i `SetPixel`
- Mała optymalność algorytmów High Pass Filter i Laplace Filter w stosunku do pozostałych (są one znacznie wolniejsze)

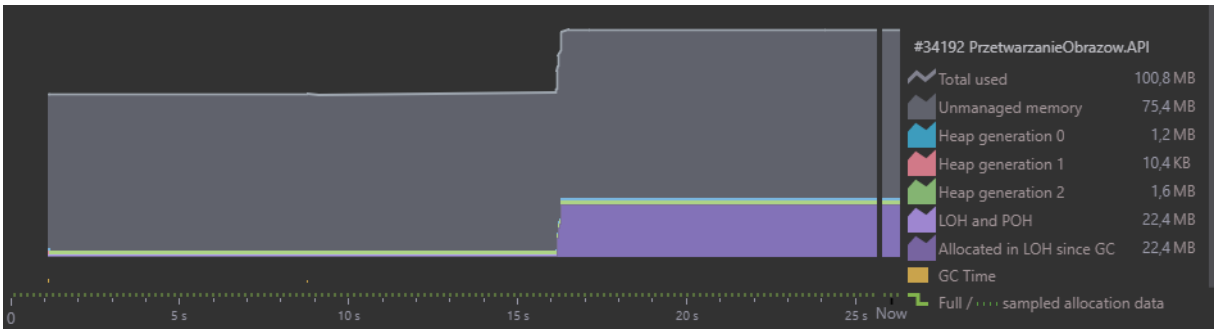
Proponowane rozwiązania:

- Zamiast korzystania z Bitmap, będziemy korzystać z tablic wartości Pixel (struktura zamiast klasy), zawierającej trzy pola: R, G i B

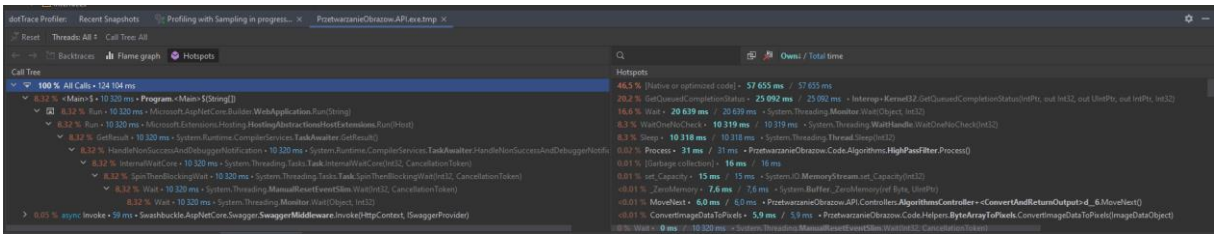
- Uzyskanie dostępu do poszczególnych pikseli w tablicy będzie optymalniejsze niż uzyskanie dostępu do pikseli w bitmapie
- Tworzenie jako obraz wyjściowy tablicy wyżej wymienionych pikseli – ułatwi to również konwersję w drugą stronę (wcześniej: `Bitmap -> byte[]`)
- Przebudowanie algorytmów High Pass Filter i Laplace Filter w oparciu o maski, zamiast ręcznego obliczania wartości dla każdego piksela po kolei

```
public override Pixel[,] Process()
{
   int sumR, sumG, sumB;
   byte byteR, byteG, byteB;

   for (int j = 1; j < Height - 1; j++)
   {
      for (int i = 1; i < Width - 1; i++)
      {
         sumR = 0;
         sumG = 0;
         sumB = 0;

         for (int x = -1; x <= 1; x++)
         {
            for (int y = -1; y <= 1; y++)
            {
               sumR += InputImage[i + x, j + y].R * Mask.Mask[x + 1, y +
1];
               sumG += InputImage[i + x, j + y].G * Mask.Mask[x + 1, y +
1];
               sumB += InputImage[i + x, j + y].B * Mask.Mask[x + 1, y +
1];
            }
         }

         byteR = sumR switch
         {
            >255 => 255,
            <0 => 0,
            _ => (byte)sumR
         };
         byteG = sumG switch
         {
            >255 => 255,
            <0 => 0,
            _ => (byte)sumG
         };
         byteB = sumB switch
         {
            >255 => 255,
            <0 => 0,
            _ => (byte)sumB
         };

         OutputImage[i, j] = new Pixel(byteR, byteG, byteB);
      }
   }

   return OutputImage;
}
```

Rysunek 17. Kod źródłowy algorytmu High Pass Filter po optymalizacjach. Wyeliminowano typ Bitmap, zastosowano maskę.



Rysunek 18. Wykonanie algorytmu High Pass Filter dla tych samych danych po zastosowaniu optymalizacji. Wykorzystana pamięć rośnie z ok. 80MB do niewiele ponad 100MB, co jest znaczącą różnicą do wersji przed optymalizacją.



Rysunek 19. Wynik z dotTrace – po zastosowaniu optymalizacji, cała metoda Process zajmuje mniej czasu, niż same wywołania metody GetPixel wcześniej.

Drobną optymalizacją w niektórych algorytmach było również pozbycie się ciągłej konwersji typu `int` na `double`, oraz ciągłego przeliczania tych samych wyrażeń. Na przykład, w algorytmie Image to Binary, zamiast ciągłego obliczania wyrażenia `threshold * 256`, zdefiniowano teraz nową stałą:

```
const double threshold = 0.50;
const int thresholdInt = (int)(threshold * 256);
```

Rysunek 20. Użycie stałej zamiast ciągłego obliczania tej samej wartości.

Konsekwencją takiego podejścia było również przyspieszenie pozostałych algorytmów, które – zamiast pojedynczego dostępu do Bitmapy i długiego kopiowania wyniku – operują teraz na tablicy pikseli.

Kolejną konsekwencją było całkowite wyeliminowanie zależności `System.Drawing.Common` z projektu API oraz Code, co m.in. zmniejsza rozmiar plików wykonywalnych.

Warto również zauważyć, że niektóre komponenty klasy Bitmap działają poprawnie tylko na systemie Windows. Oznacza to, że pierwotna wersja projektu API mogła być uruchamiana wyłącznie na serwerze z systemem Windows, podczas gdy wersja poprawiona – korzystająca z własnych typów – jest wieloplatformowa.

Poza usunięciem typu Bitmap i przebudowaniem algorytmów w oparciu o to, zastosowano następujące techniki optymalizacji:

- Pozbycie się zmiennych lokalnych, definiowanych w pętli, zamiast tego tworzenie ich w wyższym zakresie
- Ograniczenie ilości operacji arytmetycznych wewnątrz algorytmów (m.in. zamiana niektórych obliczanych wartości na stałe)

- Przeprowadzenie bardziej optymalnych konwersji między typami (na typ `byte`), w tym pozbycie się wykorzystania metody `Math.Clamp` na rzecz „ręcznego” ograniaczania wartości do zakresu 0-255

Po naniesieniu poprawek, algorytmy są wykonywane widocznie szybciej (i wciąż poprawnie), a sam serwer zużywa mniej pamięci.

5. Wnioski

Udało się zrealizować założenia projektu. W wyniku prac powstała aplikacja webowa, stworzona przy pomocy ASP.NET, będąca interfejsem dla osobnego projektu-serwera API, odbierającego dane wejściowe i przetwarzające wynik dzięki algorytmom zaimplementowanym jako mikroserwisy. Udało się również zoptymalizować wyniki prac tak, aby algorytmy działały sprawniej i wymagały mniej pamięci.

Stworzona aplikacja jest bardzo łatwo rozszerzalna. Sposób, w jaki zaimplementowane są poszczególne algorytmy oraz kontroler API pozwalają na bardzo łatwe dodanie kolejnych algorytmów, w tym również takich opartych o maski różnego rozmiaru (niekoniecznie 3x3). Wystarczy jedynie zaimplementować klasę bazową algorytmów oraz utworzyć nową konfigurację w klasie budowniczego.

Jednym z zastosowanych w projekcie wzorców jest API Gateway. Korzyścią z takiego podejścia jest to, że aplikacja będąca interfejsem wysyła wszystkie zapytania w jedno miejsce, a odpowiednie endpointy i algorytmy są dobierane automatycznie. Jest to dość komfortowe i ponownie zwiększa rozszerzalność aplikacji. Na tym etapie algorytmy są dostępne również bezpośrednio (za pomocą narzędzi takich jak Postman można wysłać zapytania do odpowiednich endpointów), jednak można temu zapobiec dodając np. pewną formę autoryzacji zapytań, tak aby mogły one być wykonywane tylko z poziomu API Gateway.

Innym przydatnym wzorcem jest Builder, który ułatwia budowanie skomplikowanych obiektów – w naszym przypadku są to obiekty realizujące algorytmy. Obiekty te wymagają odpowiedniej konfiguracji, m.in. w przypadku algorytmów korzystających z masek, co dzieje się wewnątrz specjalnie do tego przygotowanej klasy. Alternatywą byłoby „ręczne” tworzenie obiektu w każdym miejscu, w którym jest on używany, co jest znacznie gorszym i podatnym na błędy podejściem.

Podczas optymalizacji projektów API i Code nieocenioną pomocą okazały się być narzędzia dotTrace i dotMemory, wbudowane w środowisko JetBrains Rider. dotTrace pomógł odnaleźć kilka kluczowych miejsc, które wyróżniały się swoją nieoptymalnością na tle całego kodu. dotMemory pomógł natomiast zmierzyć realny impakt zmian poprzez dokładną analizę wykorzystanej pamięci, w tym pamięci LOH (Large Object Heap), gdzie prawdopodobnie wcześniej znajdowały się nasze bitmapy. Zintegrowanie tych programów w środowisko JetBrains Rider sprawia, że są one bardzo proste w użyciu i nietrudno wykorzystać ich pomoc.

6. Źródła

1. Opisy algorytmów: http://www.algorytm.org/przetwarzanie-obrazow/
2. Dokumentacja środowiska i języka: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-7.0
3. Wzorce projektowe: https://refactoring.guru/pl/design-patterns
4. Dokumentacja dotMemory: https://www.jetbrains.com/dotmemory/documentation/
5. Andrey Akinshin, „Pro .NET Benchmarking: The Art of Performance Measurement”: https://aakinshin.net/prodotnetbenchmarking/