Politechnika Gdańska

Wydział Elektroniki,
Telekomunikacji i Informatyki

Katedra: Algorytmów i Modelowania Systemów

Imię i nazwisko dyplomanta: Marcin Jurczak

Nr albumu: 171641

Forma i poziom studiów: Stacjonarne jednolite studia magisterskie

Kierunek studiów: Informatyka

Specjalność: Algorytmów i Technologii Internetowych

Praca dyplomowa magisterska

Temat pracy:

Wykorzystanie języka Elm do tworzenia aplikacji frontendowych.

Title of thesis:

Programming the front-end applications with Elm language.

Opiekun pracy:

dr inż. Krzysztof Manuszewski

Data ostatecznego zatwierdzenia raportu podobieństw w JSA: TBA

Gdańsk, 2022

Streszczenie

Celem niniejszej pracy magisterskiej jest zbadanie przydatności języka Elm oraz stojącego za nim ekosystemu w kontekście wytwarzania aplikacji internetowych. W ramach pracy stworzona została aplikacja typu *startpage*, czyli strona startowa przeglądarki, zawierająca najpotrzebniejsze informacje, takie jak czas, pogoda oraz odnośniki do wyszukiwarki i najczęściej odwiedzanych stron. Ponadto przygotowana została instrukcja laboratoryjna, która mogłaby zostać wykorzystana w ramach zajęć *„Współczesne Aplikacje Programowania Funkcyjnego"* przeprowadzanych na Wydziale Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej.

Słowa kluczowe: Elm, programowanie funkcyjne, wytwarzane aplikacji internetowych

Dziedzina nauki i techniki: Nauki inżynieryjne i techniczne, inżynieria informatyczna.

Abstract

The goal of this master thesis is to examine a usefulness of the Elm programming language and it's ecosystem in the context of web applications development. As part of thesis a *startpage* web application was created, meaning a starting page of a web browser consisting of the most useful information, such as time, weather and references to the search engine and the most visited websites. Moreover, a laboratory instruction was prepared, which could be used at *„Modern applications of functional programming"* class at Gdańsk University of Technology's Faculty of Electronics, Telecommunications and Informatics.

Keywords: Elm, functional programming, web development

Field of Science and Technology: Engineering and Technology, Information engineering.

Spis treści

Wykaz najważniejszych skrótów

API	—	ang. Application Programming Interface, pol. interfejs programowania aplikacji
CD	—	ang. Continuous Delivery, pol. ciągłe dostarczanie
CI	—	ang. Continuous Integration, pol. ciągła integracja
CLI	—	ang. Command Line Interface, pol. interfejs wiersza poleceń
CSS	—	ang. Cascading Style Sheets, pol. kaskadowe arkusze stylów
HTML	—	ang. Hypertext Markup Language, pol. hipertekstowy język znaczników
HTTP	—	ang. Hypertext Transfer Protocol, pol. protokół przesyłania hipertekstu
JSON	—	ang. JavaScript Object Notation, pol. tekstowy format zapisu danych
MVC	—	ang. Model-View-Controller, pol. Model-Widok-Kontroler
SPA	—	ang. Single Page Application, pol. jednostronicowa aplikacja internetowa
DOM	—	ang. Document Object Model, pol. obiektowy model dokumentu

Wstęp i cel pracy

Głównym celem niniejszej pracy magisterskiej było zbadanie przydatności języka Elm, a więc języka typowo funkcyjnego, oraz dedykowanych dla niego narzędzi programistycznych do tworzenia nowoczesnych rozwiązań związanych z wytwarzaniem aplikacji internetowych. W ramach pracy stworzona została frontendowa aplikacja internetowa typu *startpage*, czyli strona startowa przeglądarki, zawierająca najpotrzebniejsze informacje, takie jak czas, pogoda oraz odnośniki do wyszukiwarki i najczęściej odwiedzanych stron.

Logo Elma

Drugi rozdział przedstawia przykłady technologii powszechnie używanych do tworzenia frontendowych aplikacji internetowych, t.j. React.js, Angular oraz Vue.js. Zaprezentowane zostały ich cechy charakterystyczne oraz proste przykłady tworzenia oprogramowania z ich użyciem. Ponadto zostały pokazane podobieństwa i różnice między tymi rozwiązaniami.

Trzeci rozdział został poświęcony na wysokopoziomowe wprowadzenie do języka Elm. Opowiada o idei jaka przyświecała autorowi podczas tworzenia tego języka, jakie są jego potencjalne zastosowania i gdzie sprawdza się najlepiej. Wprowadzony zostaje koncept *The Elm Architecture*, składnia oraz podstawy języka, a także narzędzia wspomagające tworzenie oprogramowania z użyciem tejże technologii, włączając w to dokumentację.

W czwartym rozdziale opisana została implementacja przygotowanej aplikacji frontendowej napisanej w Elmie. Jest to poniekąd rozwinięcie poprzedniego rozdziału, ponieważ głównym celem jest dalej zapoznanie się z Elmem, jednak tutaj uwagę skupiam na przedstawieniu konkretnych rozwiązań technicznych, jakie zostały wykorzystane do osiągnięcia wybranego celu.

Piąty rozdział został poświęcony na omówienie zagadnień związanych z ciągłą integracją oraz ciągłym dostarczaniem, a także przedstawienie narzędzi wykorzystanych w tym celu podczas tworzenia omawianej aplikacji. Celem jest pokazanie, że Elm nie stanowi przeszkody w wykorzystywaniu tych technologii i całkowicie nadaje się do użytku produkcyjnego.

Ostatni rozdział dotyczy przede wszystkim podsumowania niniejszej pracy magisterskiej. Zostaje przedstawiony produkt dwóch semestrów działań autora oraz wyciągnięte wnioski na temat Elma jako języka przeznaczonego do tworzenia aplikacji frontendowych.

Na końcu dokumentu znajdują się spisy użytych rysunków i listingów, a także bibliografia, która została wykorzystana podczas pracy nad niniejszym dokumentem oraz w czasie zapoznawania się z tematem wytwarzania aplikacji internetowych z wykorzystaniem języka Elm.

W ramach pracy powstała również pomoc dydaktyczna w postaci instrukcji laboratoryjnej,

która przeprowadza czytelnika przez cały proces tworzenia oprogramowania w języku Elm, od przygotowania środowiska, przed podstawy języka, po stworzenie aplikacji frontendowej. Stworzona instrukcja mogłaby zostać potencjalnie wykorzystana w ramach przedmiotu Współczesne Aplikacje Programowania Funkcyjnego, prowadzonego przez mojego promotora, dra inż. Krzysztofa Manuszewskiego.

Powszechne rozwiązania

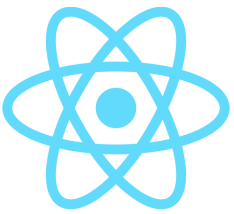
W poniższym rozdziale chciałbym przedstawić najpopularniejsze rozwiązania do tworzenia frontendowych stron internetowych, które są powszechnie stosowane zarówno przez największych gigantów technologicznych, tj. Google, Facebook, Netflix, ale także małe, dopiero wchodzące na rynek firmy startupowe.

Według ankiety przeprowadzonej w 2021 roku przez StackOverflow [1], najczęściej wybieranym przez programistów rozwiązaniem do tworzenia stron internetowych była biblioteka React.js (40,14% odpowiedzi), a na czwartym i piątym miejscu znajdowały się odpowiednio frameworki Angular (22,96% odpowiedzi) oraz Vue.js (18,97% odpowiedzi).

Ze względu na popularność wspomnianych rozwiązań, są to biblioteki i frameworki, na których chciałbym się skupić w poniższych podrozdziałach. Opiszę, czym się charakteryzują, jakie są ich najważniejsze funkcjonalności, a także co je ze sobą łączy oraz w jaki sposób są od siebie różne.

2.1 React.js

React.js [2] jest otwartoźródłową biblioteką języka programowania JavaScript, której głównym przeznaczeniem jest tworzenie interfejsów graficznych aplikacji internetowych, w większości wykorzystywana jest do aplikacji typu SPA.



Logo Reacta

Została stworzona w 2013 roku przez ówczesnego programistę Facebook'a, Jordana Walke. Rozwój Reacta utrzymywany jest po dzień dzisiejszy przez „matczyną" firmę Meta (dawniej znaną jako Facebook), a także przez społeczność indywidualnych programistów i innych organizacji ze względu na swoją otwartoźródłową naturę.

Tworzenie oprogramowania z użyciem tej biblioteki odbywa się poprzez budowanie nowych komponentów, które za pośrednictwem metody `render()` decydują, co ma zostać wyświetlane na ekranie użytkownika. Odnosząc się do dokumentacji Reacta [3], komponenty mogą być zdefiniowane na dwa sposoby: poprzez stworzenie JavaScriptowej funkcji lub wykorzystując klasę bazującą na `React.Component`.

Przykłady wspomnianych definicji zostały przedstawione odpowiednio na listingach 2.1 oraz 2.2, a ich funkcjonalność jest jednakowa — wyświetlenie napisu w formie nagłówka tagu `<h1>` o treści „*Hello,* " + imię zawarte w zmiennej `props`.

Listing 2.1: Funkcyjny komponent

```
function Hello(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Listing 2.2: Klasowy komponent

```
class Hello extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Jedną z cech kodu tworzonego z użyciem biblioteki React jest jego deklaratywność. Aby użyć stworzonego komponentu wystarczy go zaimportować i wykorzystać za pomocą tagu, dla powyższego przykładu tagu `<Hello name="Marcin"/>`, który dodatkowo zawiera pojedynczy argument `props`, będący obiektem z danymi, w tym przypadku zawierający imię „Marcin". Dzięki temu kod pisany przy pomocy biblioteki React jest bardzo reużywalny i pozwala na wykorzystanie komponentów nawet pomiędzy odrębnymi aplikacjami.

Jedną z zalet tego rozwiązania jest jego niesamowita popularność oraz dostępność gotowych rozwiązań. W Internecie znajduje się mnóstwo poradników i instrukcji pomagających nowym użytkownikom zapoznać się z biblioteką i stworzyć pierwsze proste projekty. Istnieje także dostatek pomocy dla bardziej zaawansowanych programistów Reacta oraz szeroka społeczność użytkowników chętnych do pomocy przy rozwiązywaniu bardziej skomplikowanych problemów. Dostępność gotowych rozwiązań pozwala na relatywnie łatwy rozwój oprogramowania poprzez ich wykorzystanie.

Warto także wspomnieć, że React jest jedynie biblioteką, a nie pełnoprawnym frameworkiem. Daje to programiście większą kontrolę nad tworzonym oprogramowaniem, ale także wymaga od niego większej odpowiedzialności w organizacji i utrzymaniu architektury aplikacji.

Istnieją także różne zestawy narzędzi do Reacta, które pozwalają na łatwiejsze tworzenie aplikacji z użyciem tej biblioteki, w zależności od potrzeb. *Create React App* pozwala na szybkie stworzenie prostej aplikacji jednostronicowej i zapewnia dogodne środowisko do nauki Reacta. Do stworzenia strony internetowej renderowanej po stronie serwera z użyciem Node.js można skorzystać z *Next.js*. Natomiast do tworzenia statycznych stron internetowych przydatny może się okazać framework *Gatsby*. Istnieją także bardziej elastyczne zestawy narzędziowe oferujące większą dowolność konfiguracji, czego przykładami mogą być *Neutrino*, *Nx*, *Parcel* czy *Razzle*.

2.2 Angular

Angular [4] został stworzony przez firmę Google w 2016 roku z wykorzystaniem języka TypeScript — nadzbioru języka JavaScript, który dodatkowo udostępnia takie funkcjonalności jak statyczne typowanie czy programowanie zorientowane obiektowo. Pojęcie „nadzbioru" oznacza w tym przypadku, że każdy program napisany w JavaScript jest także prawidłowym programem TypeScript. Angular początkowo miał być drugą wersją biblioteki AngularJS, jednak w ostateczności Google zdecydował się wydać tę bibliotekę jako osobny produkt.

Logo Angulara

Angular, w przeciwieństwie do biblioteki React, jest pełnoprawnym *frameworkiem*. Biblioteka jest jedynie zbiorem funkcji i klas, które odpowiadają za konkretne zagadnienie i programista używa ich w celu rozwiązania określonego problemu. Krótko mówiąc, to programista ma kontrolę nad wykonywanym kodem.

Framework często określa się jako „szkielet" programu. Udostępnia programiście gotowe środowisko do tworzenia aplikacji, dzięki czemu nie musi się martwić takimi procesami jak obsługiwanie żądań, różnych adresów URL, czy zajmowanie się ciasteczkami. Wszystkie te rzeczy robi za programistę framework. Znacznie ułatwia to proces tworzenia aplikacji, gdyż umożliwia uniknąć programowania powtarzalnych rzeczy wymaganych do prawidłowego działania aplikacji, a pozwala programiście skupić się na wdrażaniu logiki biznesowej. Oznacza to także, że framework sprawuje większą kontrolę nad aplikacją niż programista, który „dowodzi" jedynie chwilowo, po czym kontrola wraca do frameworka.

Głównym przeznaczeniem Angulara, podobnie jak w przypadku React.js, jest tworzenie aplikacji typu SPA. Kolejnym podobieństwem jest jego deklaratywność, modularność i zorientowanie na komponenty, które mogą być reużywalne pomiędzy różnymi projektami.

Do stworzenia nowego środowiska i projektu bardzo pomocne jest wykorzystanie narzędzia CLI od Angulara — `ng`. Komenda `ng new hello-world` pobierze wymagane dependencje przy użyciu menadżera paczek `npm`, utworzy nowe środowisko o wybranej nazwie (tutaj *hello-world*), stworzy szkielet aplikacji w podfolderze `src/app`, a także doda odpowiednie pliki konfiguracyjne.

Program jest od razu gotowy do uruchomienia w lokalnym środowisku przy pomocy polecenia `ng serve -open`. Jest to prosta aplikacja powitalna, zawierająca jeden komponent `AppComponent`. Kolejne komponenty mogą być dodawane z użyciem polecenia `ng generate component xyz`. Na listingu 2.3 został przedstawiony domyślny komponent w Angularze utworzony przy pomocy wspomnianego narzędzia.

Listing 2.3: Domyślny komponent w Angularze

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'hello-world';
}
```

W trakcie generowania kodu zostają także dodane szablonowe pliki HTML oraz arkusze stylów CSS. Komponent, podobnie jak w przypadku Reacta, może zostać deklaratywnie wywołany w głównym pliku `main.ts` za pomocą zdefiniowanego selektora (tutaj `app-root`).

2.3 Vue.js

Vue.js [5] jest otwartoźródłowym frameworkiem JavaScript, którego celem jest, podobnie jak w poprzednich przykładach, tworzenie interfejsów użytkownika w aplikacjach internetowych.

Logo Vue

Wykorzystuje wirtualny DOM pozwalający na szybsze renderowanie stron. Dzieje się to dzięki wirtualnej reprezentacji drzewa DOM jako obiekt JavaScriptowy, gdzie po każdej naniesionej zmianie tworzony jest nowy obiekt, a następnie są one między sobą porównywane i na koniec faktyczny DOM jest aktualizowany, co pozwala na znacznie przyspieszenie procesu renderowania strony, w porównaniu do każdorazowego nadpisywania drzewa DOM.

Aplikacje w Vue także buduje się z użyciem komponentów, którego prosty przykład typu *Hello, World!* został przedstawiony na listingu 2.4.

Listing 2.4: Komponent w Vue

```
<template>
  <h1>{{ greeting }} World!</h1>
</template>
<script>
module.exports = {
  data: function () {
    return {
```

```
    greeting: "Hello",
   };
  },
};
</script>
```

Jak wcześniej zostało wspomniane, Vue jest frameworkiem, podobnie jak Angular. Jednakże w porównaniu do Angulara jest to bardzo lekkie rozwiązanie umożliwiające programiście dużo więcej swobody w tworzeniu aplikacji. Angular zdecydowanie bardziej wymusza na użytkowniku pewne rozwiązania architektoniczne, gdzie w Vue jest to jak najbardziej możliwe, ale niekonieczne. Twórca aplikacji może wykorzystywać, jeśli tylko chce, jedynie lekkie funkcjonalności pozwalające na odpowiednie dostosowanie wyglądu strony, które są dostarczane przez framework.

Kolejnym podobieństwem do obu poprzednich rozwiązań jest możliwość wykorzystania narzędzi CLI umożliwiających zarządzanie środowiskiem deweloperskim i projektem, a także generowanie kodu, np. szablonów nowych komponentów. W przypadku Vue narzędzie to nazywa się *vue-cli*.

Elm

Elm [6] jest czysto funkcyjnym językiem programowania przeznaczonym do tworzenia graficznych interfejsów użytkownika. Powstał w roku 2012 wraz z opublikowaniem przez Evana Czaplickiego pracy „Elm: Concurrent FRP for Functional GUIs" [7]. Podczas jego tworzenia nacisk został położony na użyteczność, wydajność oraz niską podatność na błędy.

Składnia Elma jest mocno zbliżona do Haskella, a sam język jest kompilowany do JavaScriptu. Wykorzystuje wirtualny DOM, czyli wirtualną reprezentację „prawdziwego" drzewa DOM w postaci obiektu JavScriptowegoco pozwala na znacznie szybsze renderowanie stron internetowych w porównaniu do „regularnego" DOM.

Największym atutem tego języka jest zdecydowanie silnie promowany przez autora brak występowania wyjątków w czasie działania programu (tzw. *runtime exception*), co jest możliwe dzięki statycznemu sprawdzaniu typów przez kompilator Elma.

Celem kolejnych sekcji jest pokazanie na prostym przykładzie czym jest *The Elm Architecture*, a także przedstawienie i opisanie narzędzi dostępnych podczas pracy z językiem Elm, zarówno tych dostarczanych domyślnie z platformą jak i tych od niezależnych twórców.

3.1 Składnia i podstawy języka

W celu nauki podstaw języka Elm użyte zostanie narzędzie `elm repl`, pozwalające na korzystanie z interaktywnej sesji programistycznej.

3.1.1 Wartości

Najmniejszym budulcem aplikacji w Elmie są **wartości**. Mogą to być liczby, ciągi znaków, czy typy logiczne, np. 10, „Hello", True. Po wpisaniu do okna `elm repl` danej wartości, na ekranie powinna zostać pokazana powtórzona wartość, a po dwukropku jej typ.

Wartości można także łączyć z operatorami. Dla liczb będą to typowe operatory matematyczne, jak +, −, ⋆, /, dla ciągów znaków operatorem konkatenacji jest ++, a dla typów logicznych dostępne są operatory logiczne „`&&`" (AND) oraz „`||`" (OR).

Na rysunku 3.1 pokazane zostały przykłady efektów takich wywołań.

```
> 2 + (2 * 2)
6 : number
```

(a) Liczba

```
> "Hello " ++ "World!"
"Hello World!" : String
```

(b) Ciąg znaków

```
> True && False
False : Bool
```

(c) Typ logiczny

Rysunek 3.1: Wartości w Elmie

3.1.2 Funkcje

Funkcje w Elmie określają, w jaki sposób wartości mogą zostać przetworzone. Na rysunku 3.2 pokazana została przykładowa funkcja `hello`, która przyjmuje argument `name` i zwraca nowy ciąg znaków.

```
> hello name =
|   "Hello " ++ name ++ "!"
|
<function> : String → String
> hello "Bob"
"Hello Bob!" : String
> hello "Alice"
"Hello Alice!" : String
>
```

Rysunek 3.2: Definicja funkcji w Elmie

Można zauważyć, że typ argumentu `name` nie został sprecyzowany. Elm sam potrafi określić, czy dana funkcja wykona się poprawnie na podstawie operacji w niej zawartych. W pokazanym przykładzie argument `name` jest wykorzystany jako operand operatora konkatenacji „++”, który potrzebuje dwóch operandów typu `String` do prawidłowego działania programu. Jeśli użytkownik zamiast ciągu znaków podałby jako argument inny typ, np. liczbę, to kompilator Elma zwróciłby na to uwagę i wystosował użytkownikowi odpowiedni komunikat.

Przykład takiego komunikatu został przedstawiony na rysunku 3.3.

```
> hello 42
-- TYPE MISMATCH ----------------------------------------

The 1st argument to `hello` is not what I expect:

6|   hello 42
           ^^

This argument is a number of type:

    number

But `hello` needs the 1st argument to be:

    String

Hint: Try using String.fromInt to convert it to a string?

>
```

Rysunek 3.3: Komunikat kompilatora o błędzie

3.1.3 Listy

Listy są jednymi z najczęściej używanych struktur danych w Elmie. Ich przeznaczeniem jest trzymanie sekwencji wielu elementów tego samego typu.

Na rysunku 3.4 zostały pokazane przykłady użycia list. Została zdefiniowana lista `names`, zawierająca trzy elementy typu `String`, a także tablica `numbers`, która zawiera 4 liczby (typ `Int`).

```
> names = ["Bob", "Alice", "John"]
["Bob","Alice","John"] : List String
> List.length names
3 : Int
> List.reverse names
["John","Alice","Bob"] : List String
> List.isEmpty names
False : Bool
> numbers = [4,3,2,1]
[4,3,2,1] : List number
> List.sort numbers
[1,2,3,4] : List number
> List.map negate numbers
[-4,-3,-2,-1] : List number
>
```

Rysunek 3.4: Operacje na listach

3.1.4 Rekordy

Rekordy służą do trzymania wielu wartości, gdzie każda z nich jest przypisana do konkretnej nazwy. Na rysunku 3.5 pokazane zostały przykładowe operacje związane z rekordami. Zdefiniowany został rekord `bob`, który zawiera informacje o imieniu, nazwisku oraz wieku.

(a) Definicja rekordu i dostęp do jednego z pól

```
> bob =
|   { first = "Robert"
|   , last = "California"
|   , age = 61
|   }
|
{ age = 61, first = "Robert", last = "California" }
    : { age : number, first : String, last : String }
> bob.last
"California" : String
>
```

(b) Nadpisanie zawartości rekordu

```
> List.map .last [bob, alice, john]
["California","Cooper","Lennon"] : List String
> { bob | last = "Pattinson" }
{ age = 61, first = "Robert", last = "Pattinson" }
    : { age : number, first : String, last : String }
>
```

Rysunek 3.5: Przykłady pracy z rekordami

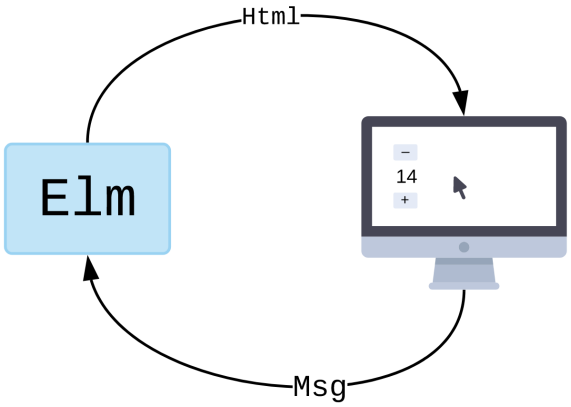
W przypadku rekordów, które zawierają wiele pól, praca z nimi może stawać się problematyczna. Wygodne może być wtedy wykorzystanie tzw. „aliasów typów", które pozwalają na definicję typu rekordu i korzystanie z niego w skróconej wersji.

Na rysunku 3.6 zdefiniowany został nowy typ `Person`, który jest równoznaczny ze zdefiniowanym na rys. 3.5a rekordem `bob`. Jednakże tak zdefiniowany typ sprawia, że kod staje się krótszy, bardziej czytelny, a praca z nim dużo wygodniejsza.

```
> type alias Person = { first: String, last: String, age: Int }
> Person
<function> : String → String → Int → Person
> Person "Bob" "Dylan" 81
{ age = 81, first = "Bob", last = "Dylan" }
    : Person
>
```

Rysunek 3.6: Definicja aliasu typu `Person`

3.2 The Elm Architecture



Rysunek 3.7: Diagram działania programu w Elmie

The Elm Architecture jest schematem tworzenia interaktywnych aplikacji internetowych lub gier. Zgodnie z rysunkiem 3.7 typowa aplikacja Elm działa w następujący sposób: Program generuje pewien kod HTML, który zostaje wyświetlony na ekranie, a następnie komputer zwraca wiadomości informujące o tym co się dzieje, np. użytkownik wcisnął guzik.

A co się dzieje wewnątrz wspomnianego programu Elmowego? Zawsze składa się z trzech podstawowych elementów:

- Model — opisujący stan aplikacji
- Update — opisujący logikę aplikacji
- View — opisujący wygląd aplikacji

W kolejnych podrozdziałach przedstawiam powyższe elementy architektury Elma na podstawie prostego programu, którego zadaniem jest wyświetlenie na ekranie dwóch guzików oraz licznika, który może się zwiększać i zmniejszać, w zależności od tego, który guzik zostanie naciśnięty przez użytkownika.

3.2.1 Model

Celem modelu przedstawionego na listingu 3.1 jest zdefiniowanie danych w naszej aplikacji.
W tym przypadku model będzie bardzo prosty — jest to rekord zawierający jedną wartość całkowitoliczbową (typ `Int`), która będzie mogła zostać zwiększona lub zmniejszona poprzez naciśnięcie odpowiedniego guzika.

Listing 3.1: *The Elm Architecture* — Model

```
type alias Model = Int


init : Model
init =
  0
```

3.2.2 Update

Listing 3.2: *The Elm Architecture* — Update

```
type Msg
  = Increment
  | Decrement


update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1


    Decrement ->
      model - 1
```

Funkcja `update`, która została przedstawiona na listingu 3.2, ma za zadanie opisywać jak nasz model będzie się zmieniał w czasie.
Jako argument przyjmuje nowo zdefiniowany typ `Msg`, który ma dwa warianty — `Increment` i `Decrement`. Typ zostaje dopasowany i w zależności od otrzymanego wariantu, model zostanie odpowiednio zaktualizowany (zmniejszony lub zwiększony) i zwrócony z funkcji.

3.2.3 View

Funkcja `view`, która została zaprezentowana na listingu 3.3, jako argument przyjmuje model i zwraca kod HTML. Wykorzystany został tutaj handler `onClick` z biblioteki `Html.Events`, który po kliknięciu guzika, do którego został przypisany, generuje odpowiednią wiadomość. Znak plusa generuje wiadomość `Increment`, znak minusa `Decrement`. Następnie wybrana wiadomość trafia do funkcji `update`.

Listing 3.3: *The Elm Architecture* — View

```
view : Model -> Html Msg
view model =
```

```
div []
  [ button [ onClick Decrement ] [ text "-" ]
  , div [] [ text (String.fromInt model) ]
  , button [ onClick Increment ] [ text "+" ]
  ]
```

3.3 Narzędzia

Platforma Elm jest dostarczana wraz z zestawem narzędzi pozwalających m.in. na kompilację plików źródłowych czy instalację dodatkowych modułów. Poniżej postaram się opisać większość z tych narzędzi, tj. dostarczanych przez Elma, ale wskazać również te dostarczane przez zewnętrznych twórców, a które znacząco ułatwiły mi pracę z tym językiem.

`elm repl` — otwiera interaktywną sesję programistyczną.

`elm init` — inicjalizuje bieżący katalog jako nowy projekt Elma poprzez stworzenie pliku `elm.json` opisującego projekt i jego zależności, a także tworzy katalog `src/`, w którym będą znajdowały się pliki `.elm`.

`elm reactor` — uruchamia serwer deweloperski, który poprzez przeglądarkę pozwala wybrać dany plik źródłowy, skompilować go i sprawdzić jak wygląda po zbudowaniu.

`elm make` — pozwala na kompilację kodu źródłowego do HTML'a lub JavaScriptu. Jest to najbardziej ogólna forma kompilacji, jaką udostępnia Elm, ale jest to niezwykle przydatne narzędzie, kiedy projekt stanie się zbyt skomplikowany na korzystanie z `elm reactor`.

`elm install` — pozwala instalować paczki dostępne na stronie `package.elm-lang.org`, które udostępniają nowe funkcjonalności, jak np. obsługa plików JSON czy praca z zapytaniami HTTP.

`elm-format` [8] — formater kodu „upiększający" kod źródłowy Elm'a zgodnie z oficjalną dokumentacją opisującą styl jego tworzenia

`elm-live` [9] — podobnie jak `elm reactor`, uruchamia serwer deweloperski, jednak jest to znacznie bardziej rozbudowane narzędzie, oferujące m.in. takie funkcjonalności jak kompilowanie Elm'a do JavaScriptu, załączanie go do pliku HTML i wyświetlanie w przeglądarce stworzonej strony

Ponadto, Elm posiada szeroką dokumentację, która w znacznym zakresie została wykorzystana podczas pracy nad niniejszym dokumentem. Składa się z dwóch głównych części — oficjalnego poradnika języka oraz technicznego opisu paczek, zarówno tych podstawowych jak i tych stworzonych i udostępnionych przez użytkowników.

Pierwsza część składa się z poradnika, który opisuje m.in. podstawowe mechanizmy języka, przedstawia przykłady prostych aplikacji wraz z ćwiczeniami pozwalających na rozwijanie swojej znajomości Elma oraz umiejętności tworzenia oprogramowania z jego wykorzystaniem, a także oferuje wskazówki odnośnie dobrych praktyk pisania kodu w tym języku.

Druga część dokumentacji zawiera techniczny opis zarówno tych fundamentalnych modułów języka, niezbędnych do tworzenia aplikacji internetowych, takich jak `elm/core` i `elm/browser`, przez te mniej niezbędne, jednakowoż w wielu przypadkach wymagane do spełnienia podstawowych funkcjonalności tworzonego oprogramowania, np. `elm/http` i `elm/json`, aż po opis paczek stworzonych i udostępnionych przez członków społeczności Elma, np. `elm-community/json-extra` i `elm-community/graph`.

Dokumentacja techniczna zawiera takie informacje, jak ogólne przeznaczenie danej paczki, jakie moduły są w niej zawarte, a także szczegółowy opis funkcji mieszczących się w ramach danego modułu wraz z przykładami ich użycia.

Implementacja

W ramach części eksperymentalnej niniejszej pracy, mającej na celu ewaluację języka Elm, stworzona została aplikacja internetowa typu *startpage*, czyli spersonalizowanej strony startowej przeglądarki zawierającej najpotrzebniejsze i najczęściej używane elementy oraz skróty. W ramach tejże aplikacji postanowiłem zaimplementować następujące funkcjonalności:

- Cyfrowy zegar wskazujący aktualny czas w strefie czasowej użytkownika,
- Aktualna data,
- Pogoda w Gdańsku przedstawiona w formie krótkiego opisu tekstowego i temperatury,
- Wyszukiwarka Google,
- Zakładki zawierające odnośniki do wybranych stron internetowych.

Powyżej wymienione elementy wykorzystują różne mechanizmy języka, dodatkowe biblioteki Elm'a oraz uwzględniają pracę z najpopularniejszymi sposobami przekazywania informacji w aplikacjach internetowych, takich jak przetwarzanie plików JSON, wysyłanie zapytań HTTP oraz praca z plikami.

W poniższych podrozdziałach skupię się na opisie wymienionych wyżej mechanizmów, bazując bezpośrednio na kodzie źródłowym stworzonej aplikacji. Przejdę przez każdy fragment architektury Elma, tj. `Model`, `View` i `Update`, opisując działanie najważniejszych według mnie fragmentów kodu oraz wyjaśniając decyzje stojące za wyborem danych rozwiązań.

Ponadto zaimplementowany został odpowiednik aplikacji z użyciem biblioteki React, udostępniający identyczne funkcjonalności jak w przypadku aplikacji w Elmie. Celem stworzenia drugiego programu było zaznajomienie się z bardziej powszechnym rozwiązaniem do tworzenia aplikacji frontedowych i następnie porównanie go z Elmem.

4.1 Model

Na listingu 4.1 przedstawiony został zaimplementowany model aplikacji, którego celem jest reprezentacja aktualnego stanu programu, a także pokazane zostają stworzone typy pomocnicze wykorzystane w głównym typie Model.

Listing 4.1: Pełen model aplikacji

```
type alias Model =
   { clockTime : ClockTime
   , weatherStatus : WeatherStatus
```

```
   , searchText : String
   , bookmarks : List Bookmark
   }
type alias ClockTime =
   { zone : Time.Zone
   , time : Time.Posix
   }
type WeatherStatus
   = Failure String
   | Loading
   | Success Weather
type alias Weather =
   { description : String
   , temperature : Float
   }
type alias Bookmark =
   { name : String
   , url : String
   }
```

W rozdziale powyżej wymienione zostały główne funkcjonalności aplikacji, które bezpośrednio przekładają się na implementację modelu. Informacje potrzebne do przedstawienia daty i godziny zawarte zostały w zmiennej typu `ClockTime`, który zawiera w sobie dane wykorzystujące bibliotekę `elm/time` do określenia strefy czasowej oraz aktualnego czasu, tj. daty i godziny.

Kolejnym elementem modelu jest zmienna typu `WeatherStatus`, która może przyjąć jedną z trzech wartości — `Failure`, `Loading` lub `Success`. Odpowiada ona za trzymanie informacji o statusie zapytania HTTP do API dostarczanego przez serwis OpenWeather [10]. Początkowo inicjalizowany jest jako wariant `Loading`, w przypadku niepowodzenia zapytania przypisywany jest wariant `Failure` wraz z tekstem opisującym błąd, w stworzonej aplikacji jest to. *„Error: Couldn't retrieve weather data"*. W przypadku powodzenia typ `WeatherStatus` przyjmuje wariant `Success` wraz z otrzymaną pogodą. Typ pogody `Weather` zawiera przeparsowane informacje z pliku JSON odebranego z zapytania HTTP, czyli krótki opis słowny pogody oraz aktualna temperatura w Gdańsku.

Następny element o nazwie `searchText` odpowiada za trzymanie informacji o frazie wpisanej przez użytkownika w dedykowanym polu tekstowym wyszukiwarki, która po wciśnięciu klawisza

Enter ma zostać wyszukana w Internecie, wykorzystując do tego wyszukiwarkę Google.

Ostatni zdefiniowany element modelu jest listą elementów typu `Bookmark`. Zadaniem elementu `bookmarks` jest trzymanie informacji o zakładkach zdefiniowanych przez użytkownika w oddzielnym pliku. Lista przekazywana jest do programu przy pomocy mechanizmu flag, co zostanie opisane w kolejnych podrozdziałach. Typ zakładki składa się z dwóch elementów — nazwy, która ma zostać wyświetlona na stronie oraz adresu URL, do którego prowadzi kliknięcie hiperlinku.

Listing 4.2: Funkcja inicjalizująca model

```
init bookmarks =
    ( Model (ClockTime Time.utc (Time.millisToPosix 0)) Loading ""
        bookmarks
    , Cmd.batch [ Task.perform AdjustTimeZone Time.here, getWeather ]
    )
```

Na listingu 4.2 przedstawiona została funkcja `init`, której zadaniem jest wstępne zainicjalizowanie modelu odpowiednimi wartościami.

Przyjmuje jeden argument, którym jest lista zakładek przekazana jako flaga, a zwraca tuplę zawierającą nowy, zainicjalizowany model oraz komendy, jakie mają zostać wykonane przez program. Inicjalizacja modelu jest trywialna, należy jedynie w odpowiedniej kolejności przekazać wartości dla każdego z elementów. Zegar zostaje wyzerowany, status pogody przyjmuje wartość `Loading`, a zakładki zostają przypisane bezpośrednio z argumentu funkcji.

Następnie należy wskazać wiadomości, które mają zostać przetworzone przez funkcję `update` i wykonane przez program na początku działania programu. W przypadku stworzonej aplikacji są to dwie wiadomości — jedna odpowiedzialna za dostosowanie odpowiedniej strefy czasowej, druga za wysłanie zapytania HTTP w celu odebrania aktualnego stanu pogody.

4.2 Update

Na listingu 4.3 znajduje się implementacja typu `Msg` definiującego rodzaje wiadomości, jakie mogą zostać odebrane i obsłużone przez funkcję `update`.

Listing 4.3: Implementacja typu `Msg` i funkcji `update`

```
type Msg
    = Tick Time.Posix
    | AdjustTimeZone Time.Zone
    | UpdateWeather
    | GotWeather (Result Http.Error Weather)
```

```
    | UpdateField String
    | Search

update msg model =
  case msg of
    Tick newTime ->
      ( { model | clockTime = ClockTime model.clockTime.zone newTime }
      , Cmd.none )
    AdjustTimeZone newZone ->
      ( { model | clockTime = ClockTime newZone model.clockTime.time }
      , Cmd.none )
    UpdateWeather ->
      ( { model | weatherStatus = Loading }
      , getWeather )
    GotWeather result ->
      case result of
        Ok weather ->
          ( { model | weatherStatus = Success weather }
          , Cmd.none )
        Err _ ->
          ( { model | weatherStatus = Failure "Error: Couldn't retrieve
              weather data" }
          , Cmd.none )
    UpdateField searchText ->
      ( { model | searchText = searchText }
      , Cmd.none )
    Search ->
      ( model
      , Nav.load ("https://google.com/search?q=" ++ model.searchText) )
```

Przedstawiona powyżej funkcja `update` w przypadku odebrania wiadomości `UpdateWeather` wykorzystuje pomocniczą funkcję `getWeather`, przedstawioną na listingu 4.4, w celu wykonania zapytania HTTP do API usługi OpenWeather, aby odebrać aktualny stan pogody w Gdańsku. Aby wysłać prawidłowe zapytanie GET do wspomnianego API, potrzebne są dodatkowe dane. W przypadku stworzonej aplikacji trzymane są one w oddzielnym pliku o nazwie `Config.elm` i są

to:

- Prywatny klucz API do OpenWeather
- Miasto — Gdańsk
- Jednostki — Stopnie Celsjusza

Do samego wysłania zapytania została użyta biblioteka `elm/http` i funkcja `get`, która wymaga adresu URL, na który ma zostać wysłane zapytanie oraz wskazania funkcji dekodującej otrzymaną odpowiedź. Funkcja dekodująca jest konieczna do wyłuskania potrzebnych informacji ze względu na statyczne typowanie języka Elm. Program *nie wie* jaki typ ma odebrane zapytanie, więc obowiązkiem programisty jest jego odpowiednie przetworzenie.

W przypadku stworzonej aplikacji jako odpowiedź zapytania GET spodziewanym formatem pliku jest JSON (`Http.expectJson`), następnie zostaje przekazany typ odebranej wiadomości (`GotWeather`), a na końcu dekoder (`weatherDecoder`), użyty do wydobycia konkretnych informacji.

Listing 4.4: Implementacja funkcji `getWeather`

```
getWeather =
  Http.get
    { url = Config.weatherApi ++ ("&q=" ++ Config.city) ++ ("&units=" ++
        Config.unit) ++ ("&appid=" ++ Config.apiKey)
    , expect = Http.expectJson GotWeather weatherDecoder
    }
```

Do implementacji dekodera użyta została biblioteka `elm/json`. Funkcja `weatherDecoder`, przedstawiona na listingu 4.5, ma na celu przetworzenie odebranego pliku w formacie JSON tak, aby pasował do zdefiniowanego przez nas typu `Weather`, który ma zawierać krótki opis pogody (typ `String`) oraz temperaturę powietrza (typ `Float`).

Przykład odpowiedzi w formacie JSON, który na potrzeby prezentacji został zmodyfikowany tak, aby zawierał tylko potrzebne nam informacje, został pokazany na listingu 4.6. Implementacja funkcji dekodującej wynika bezpośrednio ze struktury odebranej odpowiedzi. Funkcja `Json.field` pozwala na wyciągnięcie wartości pola o danej nazwie, a `Json.index` wartości znajdującej się pod wskazanym indeksem. Na końcu, funkcja `Json.map2` pozwala na przypisanie wyłuskanych wartości do wskazanego typu, tutaj `Weather`.

Znając działanie tych funkcji można zauważyć że implementacja funkcji dekodującej jest dość prosta: „sprecyzuj typ i miejsce potrzebnych pól z odpowiedzi JSON, a następnie zmapuj je do wskazanego typu”.

Listing 4.5: Implementacja dekodera JSON

```
weatherDecoder =
 map2 Weather
   (field "weather" (index 0 (field "description" string)))
   (field "main" (field "temp" float))
```

Listing 4.6: Odebrane zapytanie GET w formacie JSON

```
"weather": [ { "description": "broken clouds", ... } ],
"main": { "temp": 20.58, ... }
```

Przechodząc do implementacji cyfrowego zegara, do jego prawidłowego działania potrzebne jest wykorzystanie funkcji `subscriptions`, której zadaniem jest cykliczne generowanie wiadomości typu `Msg`. Na listingu 4.7 przedstawiona została implementacja tej funkcji dla stworzonej aplikacji. Wykorzystanie biblioteki `elm/time` pozwala na użycie funkcji `Time.every` z argumentem `10`. W praktyce oznacza to, że co 10 *ms* zostanie wygenerowana wiadomość typu `Tick` wraz z aktualnym czasem POSIX przekazanym jako argument, a następnie funkcja `update` na podstawie tej wiadomości odpowiednio zaktualizuje wartość `clockTime.time` w zdefiniowanym modelu.

Listing 4.7: Implementacja funkcji `subscriptions`

```
subscriptions _ =
   Time.every 10 Tick
```

4.3 View

Na listingu 4.8 przedstawiona została implementacja głównej funkcji wyświetlającej elementy na ekranie użytkownika — `view`. Dla lepszej czytelności kodu wykorzystuje ona wiele funkcji pomocniczych. Każda z nich wyświetla jeden z pięciu elementów funkcjonalnych, które zostały wymienione na początku tego rozdziału.

Listing 4.8: Implementacja funkcji `view`

```
view model =
   { title = "Startpage"
   , body =
      [ div []
         [ viewTime model.clockTime
         , viewDate model.clockTime
```

```
        , viewWeather model.weatherStatus
        , viewSearchBar
        , viewBookmarks model.bookmarks
        ]
    ]
}
```

Działanie funkcji odpowiedzialnych za widok jest bardzo podobne, dlatego opisana zostanie jedynie funkcja `viewSearchBar`, która została przedstawiona na listingu 4.9.

Listing 4.9: Implementacja funkcji `viewSearchBar`

```
viewSearchBar =
   div []
      [ input
         [ type_ "text"
         , placeholder "Search"
         , onInput UpdateField
         , onEnter Search
         ] []
      ]
```

Wykorzystany został tutaj formularz typu `text` wyświetlający okno z polem tekstowym. Najważniejszymi elementami są użyte tutaj funkcje `onInput` oraz `onEnter`. Funkcja `onInput` jest częścią biblioteki `Html.Events` i pozwala na wysłanie do funkcji `update` wiadomości `UpdateField` z zawartością wpisaną do pola tekstowego. Dzięki temu model zostaje odpowiednio zaktualizowany. Funkcja `onEnter` została zaimplementowana w aplikacji i pozwala na wykrycie wciśniętego klawisza Enter. Kiedy takie zdarzenie zostanie wykryte, wysłana zostaje wiadomość `Search`. Następnie wiadomość jest odpowiednio przetwarzana przez funkcję `update` i w wyniku aktualna strona zostaje przekierowana do wyszukiwarki wraz z aktualnym stanem modelu, czyli wartością `searchText`.

Porównanie

Celem tego rozdziału jest przeprowadzenie porównania języka Elm z najpopularniejszymi bibliotekami do tworzenia interfejsów użytkownika. Skupiam się przede wszystkim na porównaniu wydajności Elma w stosunku do innych rozwiązań, a także dokonuję porównania narzędzi deweloperskich dostarczanych przez Elma z tymi, które są dostępne w przypadku pracy z biblioteką React. Na końcu rozdziału przedstawiam swoją subiektywną opinię na temat języka.

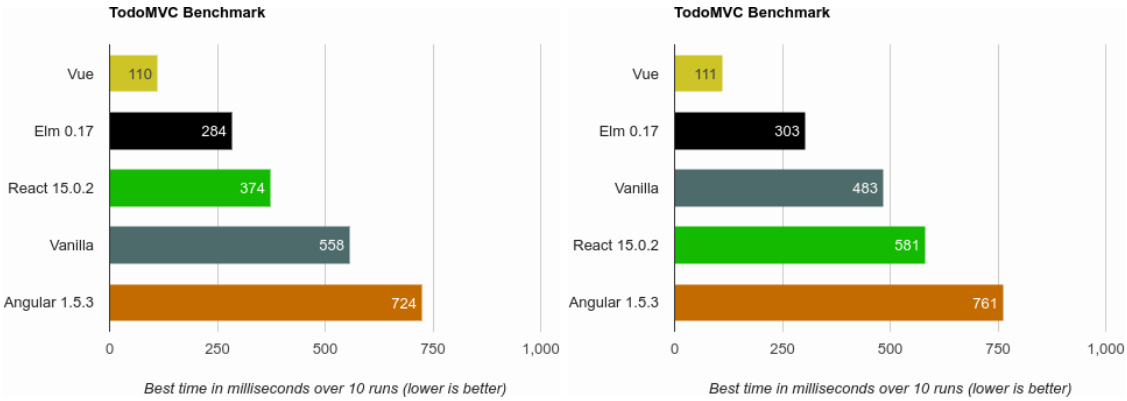
5.1 Wydajność

Celem tego podrozdziału jest porównanie wydajności wybranych bibliotek/frameworków na przykładzie TodoMVC [11]. Jest to projekt, którego celem jest implementacja tej samej aplikacji typu Todo (listy rzeczy do zrobienia) z użyciem różnych bibliotek JavaScriptowych (lub kompilujących się do JavaScriptu, tak jak ma to miejsce w przypadku Elma). W momencie pisania tej pracy dostępnych jest ponad 40 aplikacji stworzonych z użyciem różnych rozwiązań.

Na potrzeby niniejszego porównania zdecydowałem się wybrać biblioteki React, Angular i Vue, jak i aplikację napisaną w „czystym" języku JavaScript. Aplikacja w języku Elm jest oczywiście także częścią testu.

Celem testu było dodanie 200 nowych elementów do listy rzeczy do zrobienia, następnie wszystkie z nich zostały oznaczone jako wykonane, a na koniec każdy element został usunięty. Pomiar składał się z 10 uruchomień testu i wybrane zostały wyniki z najkrótszym czasem. Jako, że obiektem pomiaru jest aplikacja internetowa, do przeprowadzenia testu zostały wykorzystane przeglądarki Chromium oraz Firefox.

Ostateczne wyniki pomiaru zostały przedstawione na rys. 5.1.



(a) Test w przeglądarce Chromium 102.0.5005.61 **(b)** Test w przeglądarce Firefox 101.0

Rysunek 5.1: Testy wydajnościowe aplikacji TodoMVC

Można zauważyć, że wyniki różnią się w zależności od użytej przeglądarki internetowej. W obu przypadkach najszybsza okazała się implementacja w Vue, a najwolniejsza implementacja z użyciem Angulara. Aplikacja w Elmie uplasowała się na drugim miejscu. We wszystkich przypadkach aplikacje uruchamiane w przeglądarce Chromium działały szybciej niż w konkurencyjnym Firefoxie. Największą różnicę można zauważyć w przypadku Reacta, gdzie aplikacja poradziła sobie w Chromium o ok. 35% lepiej niż w przeglądarce od Mozilli.

5.2 Narzędzia

W kolejnej części pracy chciałbym porównać narzędzia dostępne w Elmie z zestawami narzędzi oferowanymi wraz przedstawionymi wcześniej powszechnymi rozwiązaniami tego typu.

Pierwszą rzeczą, na którą warto zwrócić uwagę jest różnica w ilości dostępnych narzędzi wspomagających pracę z tymi rozwiązaniami. Jest to spodziewana sytuacja ze względu na ich rozbieżną popularność — Elm jest dość niszowym językiem, więc dużo więcej dostępnych jest zestawów narzędziowych oferujących pomoc do pracy z rozwiązaniami takimi jak React, Angular czy Vue.

Elm domyślnie udostępnia jedynie podstawowe narzędzia, które zostały opisane w sekcji 3.3. Istnieją proste frameworki stworzone przez niezależnych programistów oferujące funkcjonalności generowania w Elmie szablonów aplikacji typu SPA, jednak nie są one tak zaawansowane jak narzędzia stworzone np. dla Angulara.

5.3 Opinia

Celem tej sekcji jest przedstawienie mojej subiektywnej opinii na temat Elma, biorąc pod uwagę moje doświadczenia zarówno z tym językiem, jak i innymi, z którymi miałem możliwość pracować.

Przed rozpoczęciem pracy nad aplikacją miałem jedynie podstawowe doświadczenie z językami funkcyjnymi, jednakże nauka Elma okazała się bardzo przyjemna. Oficjalny poradnik języka okazał się być bardzo przydatny i zdecydowanie pomógł mi zrozumieć składnię języka i architekturę tworzenia aplikacji w Elmie.

Całe doświadczenie z językiem oceniam jako bardzo pozytywne. Wielkim atutem okazał się tutaj kompilator Elma, który dostarczał wnikliwe uwagi na temat pisanego kodu i proponował rozsądne propozycje rozwiązań problemów, jeżeli takie wystąpiły. Było to szczególnie pożyteczne w przypadku refaktoryzacji kodu, gdzie jako programista nie musiałem się martwić, czy program po wszystkich zmianach będzie dalej działał prawidłowo. Wystarczyło zastosować się po kolei do wszystkich komunikatów kompilatora i pozwoliło to przeprowadzić efektywną i bezbolesną refaktoryzację.

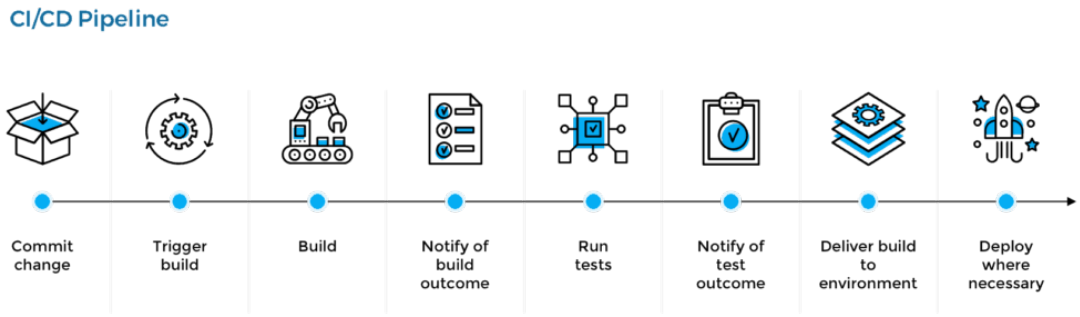
Jednakże dużym problemem Elma jest częsty brak kompatybilności wstecznej pomiędzy wersjami platformy. Aplikacje napisane z użyciem Elma w wersji 0.18 zwyczajnie nie będą działać po aktualizacji kompilatora do wersji 0.19. Wymaga to dodatkowego nakładu pracy związanego z refaktoryzacją, która jak często wspomniałem jest bezbolesna, jednakowoż czasami potrzebne jest niestety całkowite przepisanie wielu funkcji i szukanie nowych rozwiązań dla powstałych problemów.

Automatyzacja

W poniższym rozdziale chciałbym opisać procesy CI/CD oraz korzyści płynące z ich użytkowania, a także zaprezentować implementację takiego rozwiązania na przykładzie stworzonej wcześniej aplikacji w Elm'ie.

6.1 CI/CD

Mianem CI/CD określa się zbiór praktyk pozwalających na ciągłą integrację oraz ciągłe dostarczanie projektów informatycznych.



Rysunek 6.1: Przykładowy potok CI/CD

Na rysunku 6.1 przedstawiony został przykład potoku CI/CD. Jego kroki zostaną omówione w kolejnych podrozdziałach.

6.1.1 Ciągła integracja

„Ciągłą integracją" (CI) nazywa się praktyki wykorzystywane przy tworzeniu oprogramowania, polegające na regularnym kontrybuowaniu do zdalnego repozytorium kodu przez zespół deweloperski, gdzie za każdym razem następuje weryfikacja wprowadzonych zmian. Dzieje się to poprzez automatyczne budowanie projektu oraz wykonanie testów jednostkowych, a na końcu udostępnienie artefaktów

Korzyści wynikające z używania ciągłej integracji obejmują między innymi:

- wczesne wykrywanie błędów
- zmniejszenie kosztów i ilości pracy manualnej

Na rynku dostępnych jest wiele narzędzi oferujących usługi wspierające CI. Kilka najpopular-

niejszych z nich to m.in. Jenkins, TeamCity i CircleCI. Są to potężne narzędzia, umożliwiające tworzenie i zarządzanie potokami ciągłej integracji nawet w przypadku bardzo rozbudowanych projektów prowadzonych przez największe firmy informatyczne.

Ciągła integracja skupia się głównie na pracy zespołu deweloperskiego i to właśnie jego dotyczy informacja zwrotna przekazywana przez wyniki potoku. Mogą to być błędy kompilacji, problemy z łączeniem gałęzi repozytorium (*merge conflicts*) czy wskazanie testów, które zakończyły się niepowodzeniem.

Martin Fowler w swojej książce „Continuous Integration" [12] przedstawia i opisuje następujące praktyki:

- Utrzymuj jedno repozytorium kodu — systemy kontroli wersji są integralną częścią większości projektów deweloperskich. Repozytorium powinno zawierać wszystkie pliki źródłowe i każdy członek zespołu powinien mieć do niego dostęp.
- Zautomatyzuj budowanie projektu — często jest to skomplikowany proces, jednak jak większość zadań w procesie deweloperskim może zostać zautomatyzowany, i w rezultacie powinien.
- Przygotuj testy — pozwala to bardzo szybko i efektywnie wychwycić błędy, szczególnie jeśli będą uruchamiane przy każdym budowaniu projektu.
- Każdy deweloper codziennie nanosi zmiany w głównej gałęzi — codzienne zmiany, które są automatycznie budowane i testowane pozwalają na szybkie wykrycie błędów, nawet tych potencjalnych, które mogą wynikać z konfliktu pracy dwóch deweloperów.
- Każda zmiana powinna zostać zbudowana na maszynie integracyjnej — zmiana jest uznana za „gotową" dopiero wtedy, gdy powiedzie się na maszynie integracyjnej.
- Naprawiaj nieudane buildy natychmiastowo — głównym zamysłem ciągłej integracji jest praca na stabilnej gałęzi repozytorium. Jeżeli budowa projektu się nie powiodła, naprawienie zmian powinno być najważniejszym i najpilniejszym zadaniem do wykonania.
- Budowa ma odbywać się szybko — ciągła integracja ma zapewniać błyskawiczną informację zwrotną. Czas jest mocno zależny od danego projektu, ale 10–15 minutowa budowa projektu jest uznawana za odpowiednią długość.
- Testuj w odpowiedniku środowiska produkcyjnego — celem testów jest upewnienie się, że system działa prawidłowo w docelowym środowisku, więc uruchamianie ich w środowisku innym niż produkcyjne mija się z celem.
- Ułatw dostęp do ostatnich plików wynikowych — po zakończeniu budowania wszystkie artefakty powinny być dostępne do pobrania.
- Wszyscy widzą co się dzieje — użytkownicy ciągłej integracji powinni mieć możliwość zobaczenia co się dzieje w systemie. Oznacza to wprowadzenie znaczników określających, czy dany build się powiódł.

6.1.2 Ciągłe dostarczanie

„Ciągłe dostarczanie" to praktyka pozwalająca na automatyzację procesu dostarczania (publikowania) oprogramowania na podstawie nowych artefaktów kompilacji. Oznacza to, że jest ściśle związana z ciągłą integracją, której założeniem jest posiadanie jednej głównej gałęzi w repozytorium kodu, która jest zawsze stabilna i gotowa do publikacji.

Skrót *CD* powoduje nieraz pewne zamieszanie, gdyż często używany jest zamiennie dla dwóch pojęć — ciągłego dostarczania oraz ciągłego wdrażania. Są to procesy bardzo zbliżone, gdyż oba polegają na automatyzacji procesu dostarczania oprogramowania do środowiska produkcyjnego. Różnica polega na tym, że ciągłe dostarczanie wymaga ręcznej interakcji człowieka, który określi kiedy ma zostać wykonane wdrożenie, gdzie w ciągłym wdrażaniu cały proces dzieje się automatycznie, bez wpływu człowieka.

W praktyce ciągłe wdrażanie jest zdecydowanie rzadziej spotykane, gdyż pełna automatyzacja, obejmująca m.in. testy integracyjne, wydajnościowe i akceptacyjne, jest niesamowicie trudna do osiągnięcia.

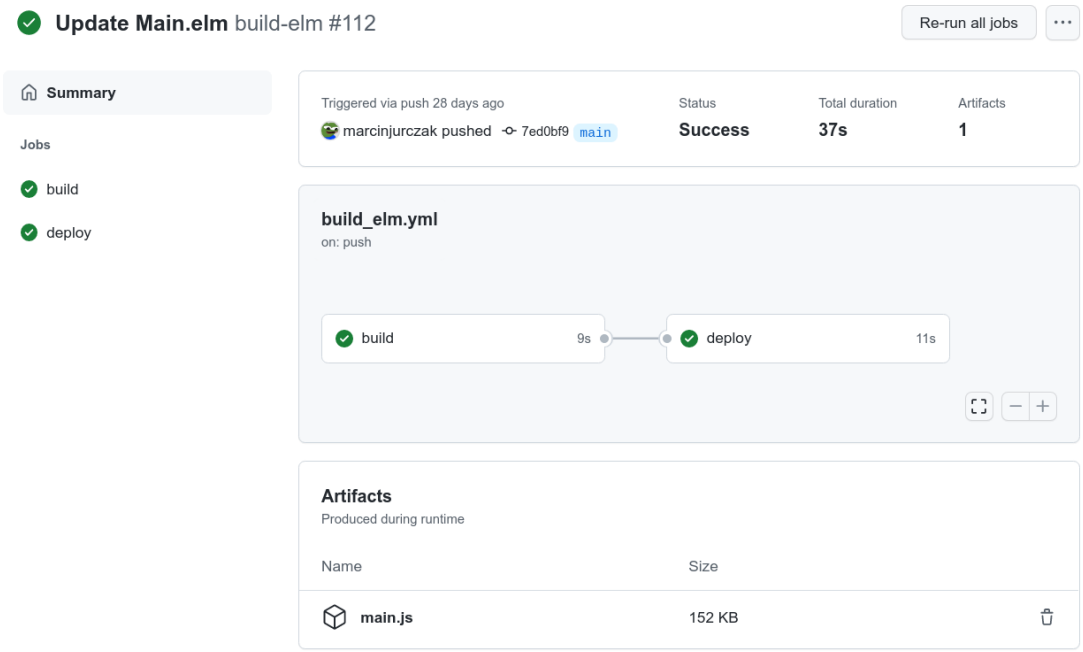
6.2 GitHub Actions

Od samego rozpoczęcia pracy nad aplikacją w Elmie, cały kod źródłowy przechowywany był z użyciem systemu kontroli wersji Git [13], a wybór serwisu hostującego zdalne repozytorium kodu padł na GitHub [14]. Ze względu na ten wybór jak i niewielki rozmiar stworzonej w Elmie aplikacji, zdecydowałem się wykorzystać GitHub Actions do zbudowania potoków CI/CD.

W odróżnieniu od standardowych przypadków użycia potoków CI/CD, gdzie oprogramowanie jest tworzone przez zespół deweloperski składający się z wielu osób, aplikacja powstała w ramach niniejszej pracy magisterskiej została stworzona przez jedną osobę. Jest to specyficzny przypadek, gdyż w takiej sytuacji nie ma możliwości wystąpienia problemów, które często mogą pojawić się w zespołach deweloperskich podczas próby integracji, na przykład konflikty łączenia gałęzi.

Jednakże, celem mojej implementacji było pokazanie, że techniki CI/CD mogą zostać efektywnie wykorzystane podczas pracy z językiem Elm.

Zaimplementowany potok CI/CD dla aplikacji w Elmie został przedstawiony na rys. 6.2. Składa się z kroku *build*, którego celem jest przygotowanie środowiska, tj. instalacja platformy Elm oraz pobranie kodu źródłowego, a następnie zbudowanie aplikacji i udostępnienie artefaktu kompilacji (plik `main.js`) oraz z kroku *deploy*, który przygotowuje oddzielną gałąź *gh-pages* poprzez pobranie i aktualizację udostępnionego wcześniej pliku wynikowego.



Rysunek 6.2: Widok CI/CD aplikacji w Elmie

Następnie przygotowana gałąź zostaje wystawiona przez GitHub Pages, co zostanie szczegółowiej opisane w kolejnej sekcji.

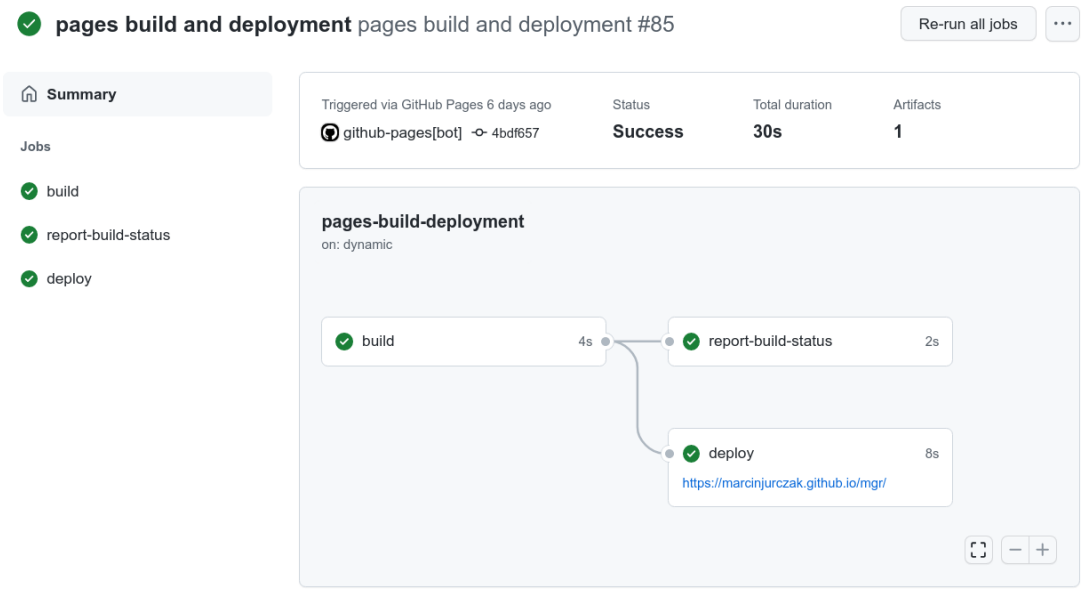
6.3 GitHub Pages

Serwis GitHub udostępnia możliwość hostowania stron internetowych prosto z repozytorium kodu. Warto zaznaczyć, że ta opcja jest dostępna za darmo, pod warunkiem, że hostowana strona znajduje się w **publicznym** repozytorium kodu. Aby skorzystać z usług GitHub Pages dla prywatnego repozytorium konieczne jest wykupienie jednej z płatnych opcji.

Wykorzystanie funkcjonalności Pages jest dość proste — w ustawieniach repozytorium należy ustawić gałąź (*branch*), która ma zostać wystawiona na stronie, a także wybrać odpowiedni katalog. Domyślnie mogą być to jedynie katalogi `/` (`root`) oraz `/docs`.

Jednakże wykorzystując GitHub Actions możliwe jest bardziej szczegółowe dostosowanie tej strony do swoich potrzeb. W przypadku stworzonej w ramach tej pracy aplikacji w Elmie, kod źródłowy znajdował się w podkatalogu `/elm`. W wyniku stworzonej konfiguracji, na gałęzi `gh-pages` zostaje opublikowana aktualna zawartość katalogu `/elm`, a następnie do tej samej gałęzi dodawany jest otrzymany podczas przeprowadzonej w poprzednim kroku fazy budowania wynikowy plik `main.js`. GitHub Pages znajduje w wybranym miejscu plik `index.html`, który zostaje wystawiony i strona staje się dostępna do oglądania w Internecie.

Na rysunku 6.3 został zaprezentowany domyślny widok potoku pozwalającego na publikację aplikacji z użyciem GitHub Pages.



Rysunek 6.3: Publikacja aplikacji z użyciem GitHub Pages

Podsumowanie

Niniejszy rozdział ma na celu wskazanie efektów końcowych pracy magisterskiej i przedstawienie wyciągniętych wniosków na temat tworzenia aplikacji frontendowych wykorzystując język programowania Elm.

7.1 Efekty

W ramach pracy magisterskiej powstały dwie implementacje tej samej aplikacji internetowej typu *startpage*, czyli strony startowej przeglądarki. Pierwsza, główna implementacja została stworzona z wykorzystaniem języka funkcyjnego Elm, a druga w celach porównawczych przy pomocy biblioteki React. Ponadto wdrożona została automatyzacja procesów ciągłej integracji i ciągłego dostarczania z wykorzystaniem zdalnego repozytorium kodu źródłowego GitHub i usług dostarczanych przez ten serwis, tj. GitHub Actions oraz GitHub Pages.

Kolejnym efektem pracy magisterskiej jest niniejszy dokument, zawierający przede wszystkim opis implementacji stworzonej aplikacji, porównanie Elma z innymi rozwiązaniami tego typu pod kątem wydajnościowym jak i uwzględniając subiektywne odczucia towarzyszące podczas pracy z językiem. Przygotowana została także instrukcja laboratoryjna przeprowadzająca czytelnika przez proces tworzenia oprogramowania z użyciem języka Elm, od przygotowania środowiska, przez podstawy języka, aż po stworzenie pełnej aplikacji frontendowej, wraz z ćwiczeniami.

7.2 Wnioski

Elm jest zdecydowanie bardzo ciekawym rozwiązaniem do tworzenia aplikacji frontedowych. Zawiera świetną dokumentację i zestaw najpotrzebniejszych narzędzi, które umożliwiają szybkie i przyjemne tworzenie oprogramowania.

Stanowczo sprawdzi się w środowiskach produkcyjnych ze względu na swój brak występowania wyjątków w czasie działania programu, a także na możliwość automatyzacji procesów CI/CD, co udało mi się pokazać w poprzednim rozdziale. Kod źródłowy wymaga trzymania sie ustalonej architektury, jednak w przypadku Elma przychodzi to dość naturalnie i powoduje, że kod jest bardzo czytelny i łatwy do zrozumienia.

Spis rysunków

Spis listingów

Bibliografia

[1] *Stack Overflow Developer Survey - Web frameworks*. StackOverflow, 2021.

[2] Alex Banks i Eve Porcello. *Learning React: functional web development with React and Redux*. "O'Reilly Media, Inc.", 2017.

[3] Jordan Walke. *React documentation*. URL: https://angular.io/docs (term. wiz. 17. 05. 2022).

[4] Deborah Kurata. *Angular documentation*. Google. URL: https://angular.io/docs (term. wiz. 17. 05. 2022).

[5] Evan You. *Vue.js documentation*. URL: https://vuejs.org/guide (term. wiz. 17. 05. 2022).

[6] Evan Czaplicki. *Elm documentation*. URL: https://elm-lang.org/docs (term. wiz. 17. 05. 2022).

[7] Evan Czaplicki. „Elm : Concurrent FRP for Functional GUIs". W: *Elm : Concurrent FRP for Functional GUIs*. 2012.

[8] Aaron VonderHaar. *elm-format*. 2022. URL: https://github.com/avh4/elm-format.

[9] Will King. *elm-live*. 2022. URL: https://www.elm-live.com/.

[10] *OpenWeather*. URL: https://openweathermap.org/guide (term. wiz. 17. 05. 2022).

[11] Addy Osmani. *TodoMVC*. URL: https://todomvc.com/ (term. wiz. 06. 06. 2022).

[12] Martin Fowler i Matthew Foemmel. *Continuous integration*. 2006.

[13] Linus Torvalds. *Git documentation*. URL: https://git-scm.com/docs (term. wiz. 17. 05. 2022).

[14] PJ Hyett Chris Wanstrath i Tom Preston-Werner. *GitHub documentation*. Microsoft. URL: https://docs.github.com/en (term. wiz. 17. 05. 2022).