

Streszczenie

Celem niniejszej pracy magisterskiej jest zbadanie przydatności języka Elm oraz stojącego za nim ekosystemu w kontekście wytwarzania aplikacji internetowych. W ramach części eksperymentalnej niniejszej 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 the usefulness of the Elm programming language and its ecosystem in the context of web applications development. As part of this 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

1	Wstęp i cel pracy	5
2	Najpopularniejsi rywale Elma	7
2.1	React.js	7
2.2	Angular	9
2.3	Vue.js	10
3	Elm	12
3.1	Cechy języka	12
3.2	Składnia i podstawy języka	13
3.2.1	Wartości	13
3.2.2	Funkcje	14
3.2.3	Listy	15
3.2.4	Rekordy	15
3.3	The Elm Architecture	17
3.3.1	Model	18
3.3.2	Update	18
3.3.3	View	18
3.4	Narzędzia	19
4	Eksperyment implementacyjny	21
4.1	Model	21
4.2	Update	24
4.3	View	27
4.4	Wnioski	30
5	Porównanie Elma z konkurencyjnymi rozwiązaniami	32
5.1	Wydażność	32
5.2	Narzędzia	33
5.3	Opinia	34

Spis treści	3
6 Nowoczesny proces wytwórczy aplikacji w języku Elm	35
6.1 CI/CD	35
6.1.1 Ciągła integracja	35
6.1.2 Ciągłe dostarczanie	37
6.2 GitHub Actions	37
6.3 GitHub Pages	38
6.4 Wnioski	39
7 Podsumowanie	40
Spis rysunków	41
Spis listingów	42
Bibliografia	43
Dodatek A: Instrukcja laboratoryjna	44
Dodatek B: Kod źródłowy do instrukcji laboratoryjnej	60

Wykaz najważniejszych skrótów

API	—	ang. Application Programming Interface, pol. interfejs programowania aplikacji
CD	—	ang. Continuous Delivery, pol. ciągle 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

1. 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 przeznaczonych do tworzenia nowoczesnych rozwiązań związanych z wytwarzaniem aplikacji internetowych. W ramach części eksperymentalnej niniejszej pracy stworzona została frontendowa aplikacja internetowa typu *startpage*, czyli strona startowa przeglądarki, zawierająca najważniejsze informacje, takie jak czas, pogoda oraz odnośniki do wyszukiwarki i najczęściej odwiedzanych stron. Na tej podstawie dokonano ewaluacji Elma, jak również stało się to punktem wyjścia do przygotowania instrukcji laboratoryjnej, która może zostać wykorzystana w dydaktyce.



Logo Elma

Drugi rozdział przedstawia przykłady technologii powszechnie używanych do tworzenia frontendowych aplikacji internetowych, które stanowią bezpośrednią konkurencję dla języka Elm, 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 wprowadzeniu 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 opisany został eksperyment implementacyjny, polegający na przygotowaniu aplikacji frontendowej z użyciem języka Elm. Jest to poniekąd rozwinięcie poprzedniego rozdziału, ponieważ głównym celem jest dalej zapoznanie się z Elmem, jednak tutaj uwaga została skupiona na przedstawieniu konkretnych rozwiązań technicznych, jakie zostały wykorzystane do osiągnięcia wybranego celu.

Piąty rozdział zawiera porównanie języka Elm z innymi technologiami używanymi do tworzenia aplikacji internetowych, które zostały przedstawione w rozdziale drugim. Porównanie dotyczy aspektów wydajnościowych tychże rozwiązań, a także ich ekosystemu, czyli narzędzi, które są dostępne do pracy z nimi. Na końcu rozdziału znajduje się subiektywna opinia autora niniejszej pracy magisterskiej na temat tego języka.

Szósty rozdział został poświęcony na omówienie zagadnień związanych z ciągłą integracją oraz ciągłym dostarczaniem, a także przedstawienie narzędzi wykorzystanych w tym celu podczas two-

rzenia 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.

2. Najpopularniejsi rywale Elma

W poniższym rozdziale przedstawiono 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.

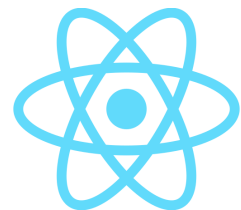
Według ankiety przeprowadzonej w 2021 roku przez StackOverflow [1], najczęściej wybieranym przez programistów rozwiązaniem do tworzenia stron internetowych była biblioteka React.js (40,14% odpowiedzi), a na czwartym i piątym miejscu znajdowały się odpowiednio frameworki Angular (22,96% odpowiedzi) oraz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

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.



Logo Reacta

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 zadanie 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.

Podobnie jak React, 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.



Logo Vue

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*.

3. 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.

Celem kolejnych sekcji jest przedstawienie najważniejszych cech języka Elm i jego podstaw oraz składni, a także pokazanie na prostym przykładzie czym jest *The Elm Architecture*. Na końcu zostają opisane narzędzia dostępne podczas pracy z językiem Elm, zarówno te dostarczane domyślnie z platformą jak i część tych stworzonych przez zewnętrznych twórców.

3.1 Cechy języka

Składnia Elma jest mocno zbliżona do Haskell, a sam język jest kompilowany do JavaScriptu. Wykorzystuje wirtualny DOM, czyli wirtualną reprezentację „prawdziwego” drzewa DOM w postaci obiektu JavaScriptowegoco pozwala na znacznie szybsze renderowanie stron internetowych w porównaniu do „regularnego” DOM. Jest to ta sama metoda, która została także wykorzystana w opisanym wcześniej rozwiązaniu Vue.js.

Wiele cech języka obraca się wokół kompilatora Elma, który jest bardzo pomocnym narzędziem ze względu na niezwykle przemyślane komunikaty o błędach i ostrzeżeniach. Według twórcy języka, kompilator powinien być asystentem programisty, nie jego przeciwnikiem, a jego rola nie powinna ograniczać się jedynie do wykrywania błędów, ale także na pomocy w zrozumieniu, dlaczego przede wszystkim dany błąd wystąpił.

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. Należy podkreślić, że jest to doświadczenie typowe dla języków funkcyjnych, a z drugiej strony nieznane większości programistom zajmującym się wytwarzaniem frontendowych aplikacji internetowych. Wynika to z faktu, że przeważająca część rozwiązań w tej domenie wykorzystuje języki obiektowe lub posługujące się paradygmatami obiektowymi.

Kolejnym atutem języka jest bezbolesna refaktoryzacja kodu. Jest to „efekt uboczny” działania kompilatora i jego przyjaznych komunikatów, co zostało wspomniane w poprzednim akapicie. Programista jest bezpiecznie prowadzony przez wszystkie wprowadzone zmiany, w razie potrzeby otrzymuje przydatne wskazówki, często pozwalające rozwiązać występujące problemy, a kiedy wskazówki okażą się niewystarczające — kompilator udostępnia odnośniki do konkretnej części dokumentacji

technicznej, która może okazać się najbardziej przydatna.

Ważną cechą języka jest także jego czytelność. W przypadku pracy z innymi rozwiązaniami, np. językiem JavaScript, często zdarza się, że po pewnym czasie programista przestaje rozumieć działanie kodu, nawet swojego! W Elmie ten problem jest zminimalizowany poprzez wykorzystanie tzw. „The Elm Architecture”. Oznacza to, że każdy program pisany w tym języku tworzony jest według ustalonego schematu, który zostanie opisany w kolejnych sekcjach. Pozwala to na szybkie zrozumienie działania kodu, nawet jeżeli widzi się go dopiero pierwszy raz.

Jako że Elm jest językiem kompilowanym do JavaScriptu, umożliwia to jego interoperacyjność. Aby zacząć używać Elma w swoim istniejącym projekcie, nie jest konieczne całkowite przepisanie kodu do tego języka, gdyż mogłoby się to okazać bardzo kosztowne i czasochłonne. Lepszym rozwiązaniem może być przepisanie do Elma niewielkiej części projektu, nawet pojedynczego elementu, a następnie załączenie skompilowanego pliku `.js` do całości programu. Takie podejście pozwala na stopniowe zapoznanie się z językiem, jego możliwościami i ograniczeniami, bez podejmowania zbędnego ryzyka.

3.2 Składnia i podstawy języka

W celu nauki podstaw języka Elm bardzo przydatne okazuje się być narzędzie `elm repl`, pozwalające na korzystanie z interaktywnej sesji programistycznej. Zostanie ono wykorzystane w dalszych sekcjach niniejszego rozdziału w celu przedstawienia składni oraz podstaw języka Elm.

3.2.1 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 konkatencji jest `++`, a dla typów logicznych dostępne są operatory logiczne `„&&”` (AND) oraz `„| |”` (OR).

Na rysunku 3.1 pokazane zostały przykłady efektów takich wywołań.

<pre>> 2 + (2 * 2) 6 : number</pre>	<pre>> "Hello " ++ "World!" "Hello World!" : String</pre>	<pre>> True && False False : Bool</pre>
(a) Liczba	(b) Ciąg znaków	(c) Typ logiczny

Rysunek 3.1: Wartości w Elmie

3.2.2 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.

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 konkatencji „`++`”, 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.

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

Rysunek 3.2: Definicja funkcji w Elmie

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

3.2.3 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

3.2.4 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.

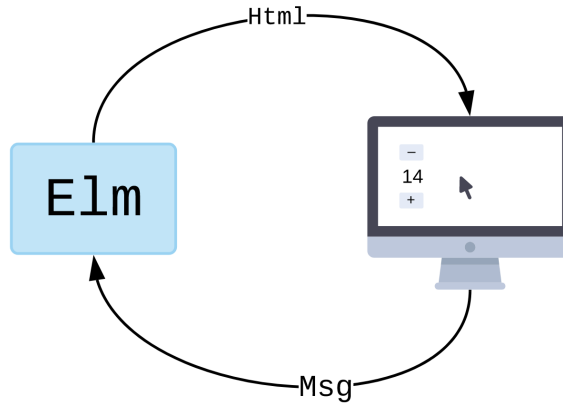
```

> 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`

3.3 The Elm Architecture



Rysunek 3.7: Diagram działania programu w Elmie

The Elm Architecture (znane także jako MVU — Model-View-Update) 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.

3.3.1 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
```

3.3.2 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.

3.3.3 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 "+" ]  
  ]
```

3.4 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.

<code>elm repl</code>	—	otwiera interaktywną sesję programistyczną.
<code>elm init</code>	—	inicjalizuje bieżący katalog jako nowy projekt Elma poprzez stworzenie pliku <code>elm.json</code> opisującego projekt i jego zależności, a także tworzy katalog <code>src/</code> , w którym będą znajdowały się pliki <code>.elm</code> .
<code>elm reactor</code>	—	uruchamia serwer deweloperski, który poprzez przeglądarkę pozwala wybrać dany plik źródłowy, skompilować go i sprawdzić jak wygląda po zbudowaniu.
<code>elm make</code>	—	pozwalą 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 <code>elm reactor</code> .
<code>elm install</code>	—	pozwalą instalować paczki dostępne na stronie <code>package.elm-lang.org</code> , które udostępniają nowe funkcjonalności, jak np. obsługa plików JSON czy praca z zapytaniami HTTP.
<code>elm-format</code> [8]	—	formater kodu „upiększający” kod źródłowy Elm'a zgodnie z oficjalną dokumentacją opisującą styl jego tworzenia
<code>elm-live</code> [9]	—	podobnie jak <code>elm reactor</code> , 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.

4. Eksperyment implementacyjny

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 zaimplementowane zostały 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 frontendowych 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 sekcji 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.

Do porównania, w aplikacji stworzonej z użyciem biblioteki `React` stan przykładowego komponentu jest trzymany w sposób przedstawiony na listingu 4.3.

Listing 4.3: Stan komponentu `Weather`

```
class Weather extends Component {  
  constructor() {  
    super();  
    this.state = { desc: "", temp: 0 };  
  }  
}
```

W konstruktorze każdego z komponentów należy zdefiniować i zainicjalizować potrzebne dane przy użyciu zmiennej `this.state`. Jest to wygodne rozwiązanie, pozwalające na łatwe i szybkie stwo-

rzenie działającej aplikacji. Jednakże, nie gwarantuje to poprawności jej działania. W przypadku odebrania nieprawidłowych danych React mimo wszystko spróbuje je przetworzyć, co może skutkować ciężkimi do znalezienia błędami, a nawet naruszeniem bezpieczeństwa.

Elm poprzez silne typowanie **wymaga**, aby dane aplikacji zostały prawidłowo przetworzone, a w przypadku niezgodności typów kompilacja nie powiedzie się. Pozwala to na szybkie zdiagnozowanie problemów z aplikacją i uniemożliwia wystąpienie błędów w czasie działania programu.

4.2 Update

Na listingu 4.4 znajduje się implementacja typu `Msg` definiującego rodzaje wiadomości, jakie mogą zostać odebrane i obsłużone przez funkcję `update`.

Listing 4.4: 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 4.5, 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 dekodery (`weatherDecoder`), użyty do wydobycia konkretnych informacji.

Listing 4.5: 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 4.6, 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 4.7. 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 4.6: Implementacja dekodera JSON

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

Listing 4.7: 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 4.8 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 przekazany jako argument, a następnie funkcja `update` na podstawie tej wiadomości odpowiednio zaktualizuje wartość `clockTime.time` w zdefiniowanym modelu.

Listing 4.8: Implementacja funkcji subscriptions

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

W stworzonej analogicznej aplikacji w React, stan programu aktualizowany jest z użyciem funkcji `setState()`. Przykład aktualizacji stanu komponentu `Clock` został przedstawiony na listingu 4.9. Metoda `componentDidMount()` jest uruchamiana po wyrenderowaniu komponentu do drzewa DOM. W tym miejscu należy zainicjalizować aktualny czas, a także zdefiniować jednosekundowy interwał, kiedy to komponent ma aktualizować czas z użyciem metody `currentTime()`.

Listing 4.9: Aktualizacja czasu w komponencie `Clock`

```
currentTime() {  
    this.setState({ date: new Date() });  
}  
componentDidMount() {  
    setInterval(() => this.currentTime(), 1000);  
}
```

4.3 View

Na listingu 4.10 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 4.10: 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 4.11.

Listing 4.11: 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`.

W aplikacji stworzonej w `React` wyświetlanie programu odbywa się za pomocą metody `render()`, która powinna znajdować się w każdym komponencie. Główny komponent `App` ma na celu wywołanie pozostałych komponentów aplikacji, a jego metoda `render()` została przedstawiona na listingu 4.12 poniżej. Zawarte są tutaj podstawowe elementy `HTML` oraz deklaratywne wywołania komponentów `Clock`, `Weather`, `Search` oraz `Bookmarks`.

Listing 4.12: Metoda `render()` komponentu `App`

```
render() {  
  return (  
    <div class="container">  
      <Clock />
```

```
        <div class="weather-container">
            <Weather />
        </div>

        <div id="search">
            <Search />
        </div>

        <div id="bookmark-container">
            <Bookmarks />
        </div>
    </div>

    );
}
```

Przechodząc do implementacji metody `render` jednego z komponentów, tutaj `Bookmarks` (listing 4.13), można zauważyć, że sposób tworzenia takiej aplikacji jest dość prosty. Lista zakładek została zaimportowana z oddzielnego pliku i jest trzymana w stanie komponentu w zmiennej `bookmarks`. Następnie z użyciem funkcji mapującej każdy element listy zostaje odpowiednio wyświetlony wykorzystując do tego odwołanie do aktualnej zawartości stanu komponentu. Metoda `render` ma za zadanie zwrócić kod HTML, a fragmenty kodu JavaScript wykonywane są wewnątrz nawiasów klamrowych.

Listing 4.13: Metoda `render()` komponentu `Bookmarks`

```
render() {
    const listBookmarks = this.state.bookmarks.map((b) => (
        <li class="bookmark">
            <a class="bookmark" href={b.url}>
                {b.name}
            </a>
        </li>
    ));
    return (
        <div class="bookmark-set">
            <div class="bookmark-title">{this.state.title}</div>
            <div class="bookmark-inner-container">
                <ul>{listBookmarks}</ul>
            </div>
        </div>
    );
}
```

```

    </div>
  );
}

```

4.4 Wnioski

Implementacja aplikacji z użyciem biblioteki React wymagała mniej czasu i wysiłku niż jej odpowiednik w języku Elm. Wynika to między innymi z dostępności materiałów w sieci jak, prostoty implementowanego rozwiązania, a także w dużej mierze doświadczenia autora z językami funkcyjnymi, które było ograniczone do wykonania kilku ćwiczeń laboratoryjnych z języka F# w ramach zajęć na drugim semestrze studiów magisterskich.

W poniższej tabeli zostały zestawione wybrane metryki dotyczące kodu źródłowego dwóch implementacji — z użyciem języka Elm oraz biblioteki React.

<div> <div>Technologia</div> <div>Metryka</div> </div>	Aplikacja w Elmie	Aplikacja w React.js
Liczba plików źródłowych	4	8
Linie kodu (bez plików <code>css</code>)	361	178
Liczba klas	0	5
Liczba funkcji/metod	22	12
Rozmiar pliku wynikowego przed kompresją	153 kB	143 kB
Rozmiar pliku wynikowego po kompresji <code>gzip</code>	35 kB	47 kB
Rozmiar całego projektu	2.2 MB	372 MB
Szacowany czas implementacji	10h	2h

React pozwala na łatwe i szybkie stworzenie działającego programu, który będzie spełniał podstawowe wymagania, gdzie Elm wymagał więcej pracy i zaangażowania. Jednakże, ze względu na naturę Elma, jego kod jest czytelniejszy, a sama aplikacja w tym języku jest bezpieczniejsza, zarówno pod względem występowania wyjątków w czasie działania programu, jak i odpowiedniego przetwarzania danych wejściowych, co zdecydowanie zmniejsza ryzyko naruszenia zabezpieczeń programu.

Wadą tworzenia aplikacji w React.js jest rozmiar dodatkowych dependencji. W przypadku omawianych programów, katalog `node_modules/` stworzony przez polecenie `npm install` zajmuje aż 372 MB w porównaniu do znikomych 2.2 MB w przypadku katalogu `elm-stuff/`. Jest to ogromna różnica i podczas pracy w ograniczonych warunkach sieciowych, np. w podróży, pobranie takiej ilości danych może być zdecydowanie kłopotliwe.

Ważnym elementem jest także refaktoryzacja kodu bardziej zaawansowanych aplikacji. W przypadku dużych projektów stworzonych z pomocą biblioteki React, refaktoryzacja może być bardzo czasochłonna i nieprzewidywalna. Może okazać się, że po jej skończeniu program uruchamia się i na pierwszy rzut oka wszystko działa bez zastrzeżeń, jednak w specyficznych sytuacjach niektóre funkcjonalności pracować prawidłowo. Takie sytuacje są często ciężkie do zdiagnozowania i w wielu przypadkach wymagają np. całościowego pokrycia kodu testami jednostkowymi.

Podczas refaktoryzacji aplikacji w Elmie takie sytuacje nie mają miejsca, gdyż w przypadkach występowania jakichkolwiek niezgodności program zwyczajnie nie skompiluje się i nie zostawia wtedy wątpliwości, że kod wymaga dodatkowej pracy. Jeżeli kod kompiluje się prawidłowo, wtedy programista może „spać spokojnie” i nie obawiać się o działanie jego programu. Przez cały proces prowadzi go także kompilator oferujący pomoc w postaci odnośników do dokumentacji oraz wskazówek, jak dany problem może zostać rozwiązany.

5. Porównanie Elma z konkurencyjnymi rozwiązaniami

Oprócz własności języka, czytelności kodu i innych cech języka, które widzi programista, niezwykle ważne jest także zachowanie aplikacji w trakcie pracy. Poniższy rozdział ma na celu przeprowadzenie porównania języka Elm z najpopularniejszymi bibliotekami i frameworkami do tworzenia interfejsów użytkownika. Nacisk został położony przede wszystkim na porównaniu wydajności Elma w stosunku do innych rozwiązań, a także dokonanie porównania narzędzi deweloperskich dostarczanych przez Elma z tymi, które są dostępne w przypadku pracy z biblioteką React. Na końcu rozdziału przedstawiona została subiektywna opinia autora niniejszej pracy na temat języka Elm.

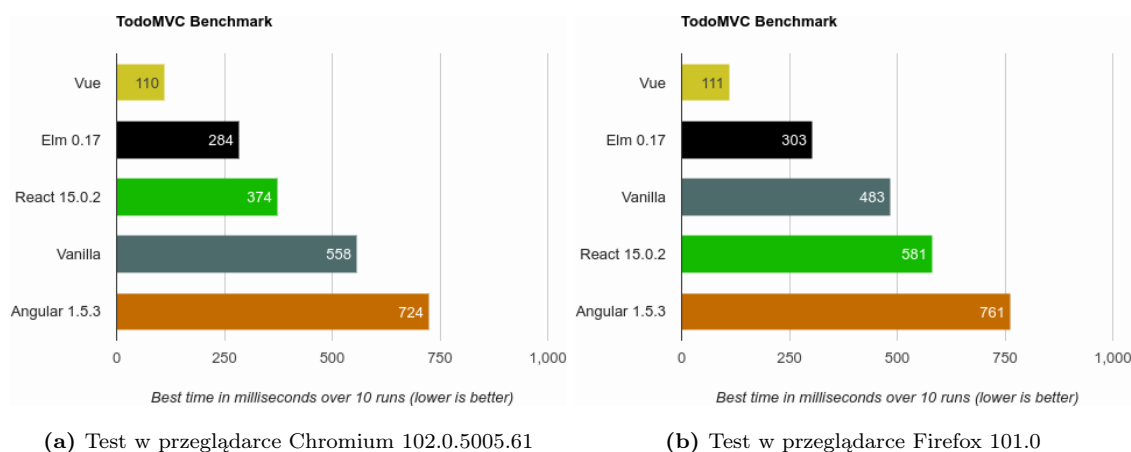
5.1 Wydajność

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 wybrane zostały technologie React, Angular i Vue, jak i aplikacja napisana 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.



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 3.4. 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 wspominałem jest bezbolesna, jednakowoż czasami potrzebne jest niestety całkowite przepisanie wielu funkcji i szukanie nowych rozwiązań dla powstałych problemów.

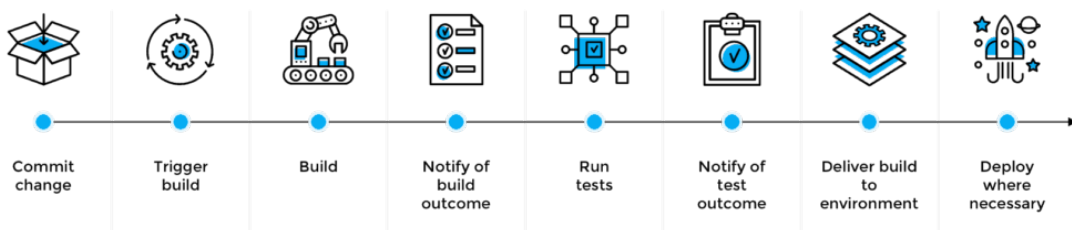
6. Nowoczesny proces wytwórczy aplikacji w języku Elm

Trudno wyobrazić sobie pracę nad nowoczesnymi rozwiązaniami dot. wytwarzania aplikacji internetowych bez wsparcia CI/CD. W poniższym rozdziale opisane zostały procesy CI/CD oraz korzyści płynące z ich użytkowania, a także ujęte zostały wnioski związane z wdrożeniem 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.

CI/CD Pipeline



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 najpopularniejszych 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, została podjęta decyzja o wykorzystaniu 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 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.

The screenshot shows a GitHub Actions workflow run titled "Update Main.elm build-elm #112". The workflow was triggered by a push to the main branch 28 days ago. The status is "Success" with a total duration of 37s and 1 artifact. The workflow consists of two jobs: "build" (9s) and "deploy" (11s). The "build" job produces an artifact named "main.js" with a size of 152 KB.

Update Main.elm build-elm #112 Re-run all jobs ...

Summary

Jobs

- build
- deploy

Triggered via push 28 days ago

marcinjurczak pushed · 7ed0bf9 · main

Status: **Success** Total duration: **37s** Artifacts: **1**

build_elm.yml
on: push

build (9s) → deploy (11s)

Artifacts
Produced during runtime

Name	Size
main.js	152 KB

Rysunek 6.2: Widok CI/CD aplikacji w Elmie

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

6.3 GitHub Pages

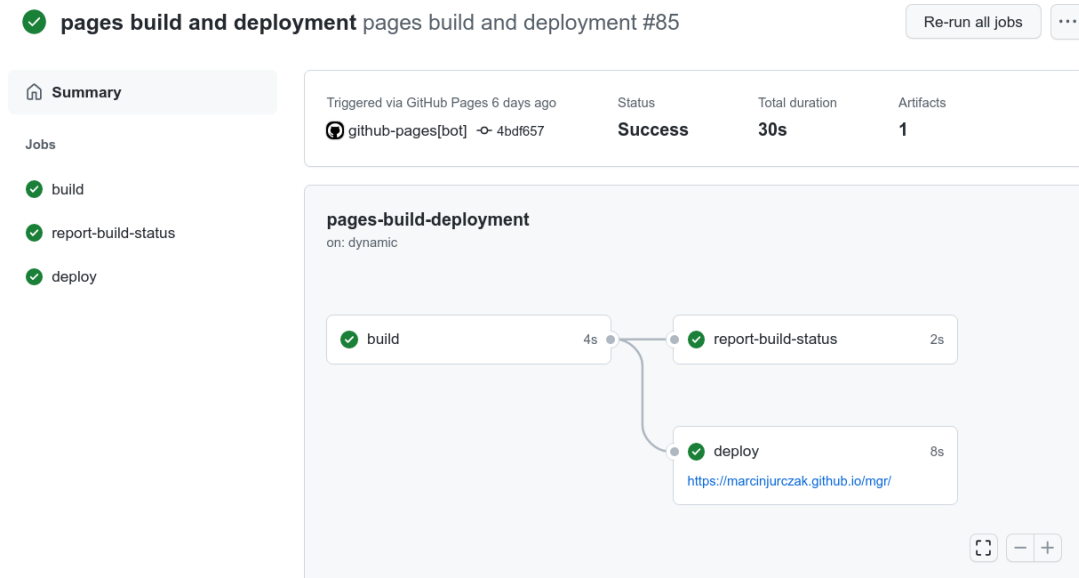
Serwis GitHub udostępnia możliwość hostowania stron internetowych prosto z repozytorium kodu. Warto zaznaczyć, że ta opcja jest dostępna za darmo, pod warunkiem, że hostowana strona znajduje się w **publicznym** repozytorium kodu. Aby skorzystać z usług GitHub Pages dla prywatnego repozytorium konieczne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

6.4 Wnioski

Język Elm nie stanowi żadnych przeszkód w automatyzacji procesów CI/CD. Wynika to częściowo z interoperacyjności tego języka, czyli faktu kompilowania do JavaScriptu. Po kompilacji program Elmowy może być obsługiwany tak jak program napisany w języku JavaScript, a do tej technologii dostępnych jest wiele rozwiązań CI/CD.

7. Podsumowanie

Elm jest zdecydowanie bardzo ciekawym rozwiązaniem do tworzenia aplikacji frontendowych. Jest językiem czysto funkcyjnym, co przed rozpoczęciem przygody z nim może wymagać wcześniejszego zapoznania się z najważniejszymi zagadnieniami w tym temacie. Jednakże język zawiera świetną dokumentację i zestaw najpotrzebniejszych narzędzi, które umożliwiają relatywnie szybkie i przyjemne tworzenie oprogramowania.

Aplikacje stworzone z użyciem tej technologii są niezwykle wydajne i szybkie w porównaniu do rozwiązań konkurencyjnych, takich jak React i Angular. Największą zaletą jest zdecydowanie brak występowania wyjątków w czasie działania programu, co jest skutkiem działania kompilatora Elma oraz silnego typowania. Dlatego w wielu przypadkach programowania w Elmie można krótko powiedzieć — jeżeli kod Elma się kompiluje, to znaczy, że działa.

Elm stanowczo sprawdzi się także 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 się ustalonej architektury, jednak w przypadku Elma przychodzi to dość naturalnie i powoduje, że kod jest bardzo czytelny i łatwy do zrozumienia.

Stworzona w ramach niniejszej pracy instrukcja laboratoryjna może stanowić wprowadzenie do języka Elm dla osób, które dopiero zaczynają w nim programować, gdyż zawiera wszystkie niezbędne elementy takie jak instrukcja przygotowania środowiska deweloperskiego i opis podstaw języka i składni.

Spis rysunków

3.1	Wartości w Elmie	13
3.2	Definicja funkcji w Elmie	14
3.3	Komunikat kompilatora o błędzie	14
3.4	Operacje na listach	15
3.5	Przykłady pracy z rekordami	16
3.6	Definicja aliasu typu <code>Person</code>	16
3.7	Diagram działania programu w Elmie	17
5.1	Testy wydajnościowe aplikacji <code>TodoMVC</code>	33
6.1	Przykładowy potok <code>CI/CD</code>	35
6.2	Widok <code>CI/CD</code> aplikacji w Elmie	38
6.3	Publikacja aplikacji z użyciem <code>GitHub Pages</code>	39
7.1	Wyjście instrukcji <code>elm</code>	45
7.2	Otwarta interaktywna sesja <code>elm repl</code>	47
7.3	Wartości w Elmie	48
7.4	Definicja funkcji w Elmie	48
7.5	Komunikat kompilatora o błędzie	49
7.6	Operacje na listach	49
7.7	Przykłady pracy z rekordami	50
7.8	Definicja aliasu typu <code>Person</code>	50

Spis listingów

2.1	Funkcyjny komponent	7
2.2	Klasowy komponent	8
2.3	Domyślny komponent w Angularze	9
2.4	Komponent w Vue	10
3.1	<i>The Elm Architecture</i> — Model	18
3.2	<i>The Elm Architecture</i> — Update	18
3.3	<i>The Elm Architecture</i> — View	18
4.1	Pełen model aplikacji	21
4.2	Funkcja inicjalizująca model	23
4.3	Stan komponentu Weather	23
4.4	Implementacja typu Msg i funkcji update	24
4.5	Implementacja funkcji getWeather	25
4.6	Implementacja dekodera JSON	26
4.7	Odebrane zapytanie GET w formacie JSON	26
4.8	Implementacja funkcji subscriptions	26
4.9	Aktualizacja czasu w komponencie Clock	27
4.10	Implementacja funkcji view	27
4.11	Implementacja funkcji viewSearchBar	28
4.12	Metoda render() komponentu App	28
4.13	Metoda render() komponentu Bookmarks	29

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).
- [15] Isaac Z. Schlueter. *npm documentation*. URL: <https://docs.npmjs.com/> (term. wiz. 17.05.2022).
- [16] Evan Czaplicki. *Elm installer*. 2019. URL: <https://github.com/elm/compiler/releases>.
- [17] *Visual Studio Code documentation*. Microsoft. URL: <https://code.visualstudio.com/docs> (term. wiz. 17.05.2022).

Dodatek A: Instrukcja laboratoryjna

Poniższa instrukcja laboratoryjna ma na celu krok po kroku przeprowadzić czytelnika przez proces tworzenia aplikacji w Elmie, zaczynając od przygotowania środowiska deweloperskiego, przez podstawy języka wraz z ćwiczeniami pozwalającymi na lepsze zrozumienie składni, aż po stworzenie większej aplikacji frontendowej.

Przygotowanie środowiska

Pierwszą rzeczą, którą należy się zająć przed rozpoczęciem nowego projektu jest przygotowanie odpowiedniego środowiska deweloperskiego. Należy upewnić się, że wszystkie narzędzia potrzebne do wykonania pracy są zainstalowane i prawidłowo skonfigurowane. W przypadku pracy z Elm'em zalecane będzie korzystanie przede wszystkim z platformy dostarczanej przez autora, edytora tekstu wspierającego podświetlanie składni, a także innych narzędzi wspomagających proces tworzenia oprogramowania z użyciem tej technologii.

Platforma Elm

Najważniejszą rzeczą, jaka będzie potrzebna podczas pracy z Elm'em, będzie platforma języka zawierająca m.in. takie narzędzia jak kompilator oraz menadżer bibliotek. Poniżej przedstawiam instrukcję instalacji tej platformy na systemach operacyjnych Linux i Windows. Po przejściu tych kroków, w wierszu poleceń należy wykonać instrukcję `elm`. Jeśli wszystko zostało prawidłowo zainstalowane i skonfigurowane, na ekranie powinien pojawić się widok podobny do przedstawionego na rysunku 7.1.

```
→ ~ elm
Hi, thank you for trying out Elm 0.19.1. I hope you like it!
```

I highly recommend working through <<https://guide.elm-lang.org>> to get started. It teaches many important concepts, including how to use `elm` in the terminal.

The most common commands are:

```
elm repl
  Open up an interactive programming session. Type in Elm expressions like
  (2 + 2) or (String.length "test") and see if they equal four!

elm init
  Start an Elm project. It creates a starter elm.json file and provides a
  link explaining what to do from there.

elm reactor
  Compile code with a click. It opens a file viewer in your browser, and
  when you click on an Elm file, it compiles and you see the result.
```

There are a bunch of other commands as well though. Here is a full list:

```
elm repl    --help
elm init    --help
elm reactor --help
elm make    --help
elm install --help
elm bump    --help
elm diff    --help
elm publish --help
```

Adding the `--help` flag gives a bunch of additional details about each one.

Be sure to ask on the Elm slack if you run into trouble! Folks are friendly and happy to help out. They hang out there because it is fun, so be kind to get the best results!

Rysunek 7.1: Wyjście instrukcji `elm`

Linux

Najprostszym sposobem instalacji platformy Elm na systemie operacyjnym Linux jest wykorzystanie narzędzia `npm` – powszechnie używanego menadżera pakietów służącego do zarządzania warstwą frontendową aplikacji internetowych. Aby zainstalować `npm` należy skorzystać z systemowego menadżera pakietów. Na przykładzie dystrybucji Ubuntu będą to komendy:

```
$ sudo apt update
$ sudo apt install npm
```

Kiedy narzędzie zostanie już pomyślnie zainstalowane, można przejść do instalacji platformy Elm. Posłuży do tego polecenie:

```
$ npm install -g elm
```

Zgodnie z dokumentacją npm [15], flaga `-g` oznacza, że pakiet zostanie zainstalowany globalnie, dzięki czemu będzie dostępny z każdego miejsca z systemu. Aby sprawdzić, czy rzeczywiście tak się stało, wystarczy w wierszu poleceń uruchomić komendę `elm`. Jeżeli wszystkie kroki przebiegły pomyślnie, na ekranie powinien ukazać się widok podobny do przedstawionego na rysunku 7.1, tak jak zostało to już wcześniej wspomniane.

Windows

Osoby korzystające z systemu operacyjnego Windows mogą skorzystać z npm, tak jak to było opisane w powyższej sekcji dotyczącej Linuxa lub posłużyć się dedykowanym instalatorem Elm'a [16] na system Windows. W tym drugim przypadku wystarczy przejść przez wszystkie kroki zostawiając opcje domyślne i w rezultacie Elm zostanie pomyślnie zainstalowany i będzie gotowy do użytkowania. W celu sprawdzenia czy faktycznie tak się stało, należy uruchomić wiersz poleceń oraz wykonać instrukcję `elm`. Wyjście komendy powinno być podobne do tego przedstawionego na rysunku 7.1, tak jak zostało to już wcześniej wspomniane.

Edytor

Ważnym elementem tworzenia oprogramowania jest wyposażenie się w odpowiedni edytor tekstowy, który jest w stanie podświetlać składnię języka, z którego aktualnie korzystamy. Żeby osiągnąć ten cel, w przypadku Elm'a potrzebna będzie instalacja dodatkowej wtyczki do jednego z następujących edytorów:

- Atom
- Emacs
- IntelliJ
- Light Table
- Sublime Text
- Vim
- VS Code

Powyższa lista zawiera odnośniki do wspomnianych wtyczek dla danego edytora, wystarczy kliknąć nazwę swojego ulubionego edytora i pobrać odpowiedni dodatek. Na potrzeby niniejszej instrukcji przedstawię proces instalacji wtyczki dla edytora Visual Studio Code [17], ponieważ jest to jedno z najbardziej powszechnie używanych narzędzi do pracy z kodem źródłowym.

W tym celu należy otworzyć edytor VS Code, otworzyć konsolę *Quick Open* przy pomocy skrótu `Ctrl+P` i wpisać następującą komendę:

```
ext install Elmtooling.elm-ls-vscode
```

Wtyczka powinna zostać automatycznie zainstalowana i dalsza konfiguracja nie jest konieczna, edytor powinien być gotowy do pracy. Po instalacji można przejść do kolejnego kroku.

Tworzenie projektu

Elm jest dostarczany wraz z zestawem bardzo przydatnych narzędzi. Jednym z nich jest `elm init`, które posłuży nam do stworzenia nowego projektu. W tym celu należy otworzyć wiersz poleceń i wykonać następujące instrukcje:

```
$ mkdir lab
$ cd lab
$ elm init
```

Po wypisaniu zawartości katalogu `lab` z użyciem polecenia `ls` powinny pojawić się dwa nowe elementy:

- Plik `elm.json` opisujący projekt oraz jego zależności
- Katalog `src/` zawierający nasze przyszłe pliki Elm'a

Następnym krokiem będzie utworzenie nowego pliku `Main.elm` w nowo utworzonym katalogu `src/`. Będzie się tam znajdował kod aplikacji, która zostanie stworzona w kolejnych krokach.

Podstawy języka Elm

W celu nauki podstaw języka Elm użyte zostanie narzędzie `elm repl`, pozwalającego na korzystanie z interaktywnej sesji programistycznej. Należy otworzyć wiersz poleceń i wpisać polecenie `elm repl`. Powinien ukazać się widok podobny do przedstawionego na rysunku 7.2 poniżej.

```
→ ~ elm repl
—— Elm 0.19.1 ——
Say :help for help and :exit to exit! More at <https://elm-lang.org/0.19.1/repl>
>
```

Rysunek 7.2: Otwarta interaktywna sesja `elm repl`

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 konkatencji jest ++, a dla typów logicznych dostępne są operatory logiczne „&&” (AND) oraz „||” (OR).

Na rysunku 7.3 pokazane zostały przykłady efektów takich wywołań.

<code>> 2 + (2 * 2)</code> <code>6 : number</code>	<code>> "Hello " ++ "World!"</code> <code>"Hello World!" : String</code>	<code>> True && False</code> <code>False : Bool</code>
(a) Liczba	(b) Ciąg znaków	(c) Typ logiczny

Rysunek 7.3: Wartości w Elmie

Funkcje

Funkcje w Elmie określają, w jaki sposób wartości mogą zostać przetworzone. Na rysunku 7.4 pokazana została przykładowa funkcja `hello`, która przyjmuje argument `name` i zwraca nowy ciąg znaków.

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 konkatencji „++”, 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 7.5.

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

Rysunek 7.4: Definicja funkcji w Elmie


```
> 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 7.5: Komunikat kompilatora o błędzie

Listy

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

Na rysunku 7.6 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 7.6: Operacje na listach

Rekordy

Rekordy służą do trzymania wielu wartości, gdzie każda z nich jest przypisana do konkretnej nazwy. Na rysunku 7.7 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 7.7: 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 7.8 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 7.8: Definicja aliasu typu Person

Aplikacja frontendowa

W ramach większego projektu zbudowana zostanie strona internetowa typu `startpage`, czyli strona startowa przeglądarki zawierająca najważniejsze i najczęściej używane elementy. Zaimplementowane zostaną cztery moduły — Data i czas, pogoda, wyszukiwarka oraz pasek zakładek.

Każda z funkcjonalności będzie dodawana inkrementalnie poprzez dokładanie kolejnych fragmentów kodu do każdej z części architektury Elma, tj. `Model`, `Update` oraz `View`, czyli elementów odpowiedzialnych odpowiednio za stan, logikę i wygląd aplikacji.

Plik źródłowy `Main.elm` został stworzony w jednym z poprzednich kroków — należy go otworzyć w swoim wybranym edytorze i przejść do kolejnych kroków.

Szkielet aplikacji

W pierwszej kolejności należy uruchomić aplikację typu „Hello World!”, aby upewnić się, że środowisko zostało prawidłowo skonfigurowane. W załączonym pliku `Main.elm` znajdują się wszystkie potrzebne do działania klauzule importujące biblioteki podstawowe Elma. Ponadto zaimplementowana została podstawowa architektura aplikacji. Aby uruchomić aplikację „Hello, World!” należy otworzyć wiersz poleceń i wykonać polecenie:

```
$ elm reactor
```

Powinien pokazać się następujący widok:

```
→ lab git:(main) ✕ elm reactor
Go to http://localhost:8000 to see your project dashboard.
```

Po otwarciu w przeglądarce adresu `http://localhost:8000` należy znaleźć plik `Main.elm` i go otworzyć. Powinien wyświetlić się napis „Hello, World!” oraz guzik `Click me!`, który zamienia ciąg tekstowy na „Hello, again!”.

Celem dalszych sekcji będzie rozwinięcie tego programu poprzez implementację wymaganych funkcjonalności omówionych na początku instrukcji.

Czas

Do wydobycia informacji o aktualnym czasie wykorzystana zostanie biblioteka `elm/time`. W celu zainstalowania tej biblioteki w wierszu poleceń należy wykonać komendę `$ elm install elm/time`, a następnie załączyć bibliotekę do tworzonego programu używając klauzuli `import Time exposing (..)`.

Przechodząc do edycji kodu, pierwszą rzeczą jest zdefiniowanie modelu programu. Potrzebne będą informacje o strefie czasowej oraz aktualny czas. W tym celu należy stworzyć nowy typ `DateTime`, który będzie przechowywał te dane, a następnie dodać go do modelu.

```
type alias Model =  
    { dateTime : DateTime }  
type alias DateTime =  
    { zone : Time.Zone, time : Time.Posix }
```

Kolejnym krokiem będzie zdefiniowanie typów wiadomości, które może odebrać program. Do prawidłowego działania zegara potrzebne będzie dostosowanie strefy czasowej oraz pobranie aktualnego czasu. Oznacza to, że należy zdefiniować dwie wiadomości — `AdjustTimeZone` oraz `Tick`.

```
type Msg  
    = Tick Time.Posix  
    | AdjustTimeZone Time.Zone
```

Oba typy wiadomości muszą zostać odpowiednio przetworzone w funkcji `update`. W obu przypadkach wystarczające będzie nadpisanie stworzonego modelu nowymi wartościami. Przykład nadpisanie rekordu został przedstawiony wcześniej na rys. 7.7b.

Następnie należy zdefiniować funkcję `init`, gdzie wskazany zostanie sposób inicjalizacji modelu oraz operacje, jakie mają zostać wykonane na początku programu. W przypadku daty i czasu należy początkowo wyzerować obie wartości, a następnie pobrać aktualne informacje wykorzystując stworzone przed chwilą wiadomości.

```
init _ =  
    ( Model (DateTime Time.utc (Time.millisToPosix 0))  
    , Cmd.batch [ Task.perform AdjustTimeZone Time.here, Task.perform  
        Tick Time.now ]  
    )
```

Jednakże czas zmienia się z sekundy na sekundę, więc jednorazowe ustawienie wartości modelu niestety nie jest wystarczające — trzeba aktualizować model co sekundę. Aby to osiągnąć, wykorzystana zostanie funkcja `subscriptions`. Pozwala na nasłuchiwanie zewnętrznych zdarzeń, takich jak kliknięcie myszki, naciśnięcie klawisza na klawiaturze, zmiany w geolokalizacji lub — tykanie zegara.

W przypadku tworzonej aplikacji należy co sekundę generować nową wiadomość `Tick`, która po jej przetworzeniu w funkcji `update` zaktualizuje aktualny czas w modelu.

```
subscriptions _ =  
    Time.every 1000 Tick
```

Ostatnim elementem architektury Elma jest funkcja `view`, której zadaniem jest wyświetlanie programu na ekranie.

```
view model =  
    { title = "Hello"  
    , body =  
        [ viewTime model.dateTime  
        , viewDate model.dateTime  
        ]  
    }
```

W celu polepszenia czytelności kodu, funkcja `view` wykorzystuje dwie funkcje pomocnicze — `viewTime` oraz `viewDate`. Część implementacji jednej z nich została przedstawiona poniżej. W ramach ćwiczenia należy dokończyć implementację funkcji `viewDate` oraz stworzyć analogiczną funkcję `viewTime`. Ponadto przedstawiony został początek funkcji `toEnglishWeekday`, która przyjmuje wartość typu `Time.Weekday` i zwraca typ `String`, który może zostać następnie wyświetlony w funkcji `viewDate`. Implementację tej funkcji także należy dokończyć oraz stworzyć analogiczną funkcję pozwalającą na dekodowanie nazw miesięcy. Dla lepszego wyglądu zegara warto także stworzyć funkcję dodającą znak 0 dla liczb mniejszych od 10.

```
viewDate dateTime =  
    let  
        weekday =  
            Time.toWeekday dateTime.zone dateTime.time  
        ...  
    in  
    div [] [ text (toEnglishWeekday weekday) ]
```

```
toEnglishWeekday weekday =  
    case weekday of  
        Mon ->  
            "Monday"  
        ...
```

Jeżeli wszystko udało się pomyślnie, w przeglądarce internetowej powinien wyświetlić się aktualny czas, zmieniający się co sekundę, a pod nim data. Następnie można przejść do kolejnej sekcji.

Pogoda

Pogoda będzie pobierana wykorzystując zapytania HTTP do API portalu OpenWeatherMap. Potrzebne będą do tego dwie biblioteki — `elm/http` do wysłania zapytania oraz `elm/json` do odebrania i przetworzenia odpowiedzi w formacie JSON. Podobnie jak poprzednio, należy je zainstalować używając poleceń:

```
$ elm install elm/http  
$ elm install elm/json
```

Następnie należy dodać do pliku `Main.elm` następujące klauzule:

```
import Http  
import Json.Decode exposing (..)
```

Przed przejściem do edycji kodu programu należy założyć darmowe konto na stronie OpenWeatherMap, a następnie wygenerować klucz API, który będzie konieczny do prawidłowego działania aplikacji. Po stworzeniu prywatnego klucza API można przejść do kolejnych kroków.

W przypadku pogody potrzebne będzie stworzenie dwóch nowych typów: jeden będzie zawierał informacje o faktycznym stanie pogody, tj. temperatura i krótki opis słowny, a drugi informacje o statusie zapytania HTTP.

```
type WeatherStatus  
    = Failure String  
    | Loading  
    | Success Weather  
type alias Weather =  
    { description : String  
    , temperature : Float
```

```
}
```

Powyższy typ `WeatherStatus` należy dodać do głównego modelu. Ponadto należy zdefiniować w programie dane potrzebne do wysłania zapytania, a następnie wykorzystać je w funkcji odpowiedzialnej za wysłanie zapytania GET do OpenWeatherAPI. Są to:

- URL do wysłania zapytania
- Prywatny klucz API do OpenWeatherMap
- Miasto
- Jednostki

Przykład implementacji został przedstawiony poniżej:

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

Szczegóły użycia API opisane są w dokumentacji OpenWeatherMap.

Z perspektywy Elma najważniejsze jest sprecyzowanie rodzaju zapytania (tutaj GET) oraz adresu URL, na który ma zostać wysłane zapytanie. Konieczne jest także określenie formy spodziewanej odpowiedzi (JSON), a także przekazanie funkcji jej dekodującej. Na końcu należy przekazać rodzaj wiadomości do wygenerowania po odebraniu odpowiedzi (`GotWeather`).

A jak wygląda funkcja i dekodująca i do czego służy? Przykład prostego dekodera został przedstawiony poniżej. Celem dekodera jest przetworzenie pliku w formacie JSON w taki sposób, aby był zrozumiały dla Elma. Należy w nim określić nazwy oczekiwanych pól oraz opisać jak się do nich dostać.

Funkcja `personDecoder` spodziewa się trzech wartości — dwóch ciągów znaków i jednej liczby, które znajdują się w polach nazwanych odpowiednio `first`, `last` oraz `age`. Typ `Person` został stworzony na rys. 7.7b i zawiera w sobie takie same typy jak dekodery. Oznacza to, że taka odpowiedź JSON powinna zostać prawidłowo zmapowana na typ `Person`.

```
personDecoder =  
  map3 Person  
    (field "first" string)  
    (field "last" string)
```

```
(field "age" int)
```

W ramach ćwiczenia należy zaimplementować funkcję `weatherDecoder`, która zmapuje odpowiednie pola z odpowiedzi JSON na typ `Weather` zaimplementowany wcześniej w modelu.

W przypadku wiadomości `Msg` sytuacja będzie podobna jak przy tworzeniu typów — potrzebny będzie jeden rodzaj wiadomości odpowiedzialny za wysłanie zapytania oraz drugi odpowiedzialny za jego odebranie i odpowiednie zaktualizowanie modelu na podstawie danych odebranych z dekodera.

```
type Msg
  = ...
  | UpdateWeather
  | GotWeather (Result Http.Error Weather)
```

```
update msg model =
  case msg of
    ...
    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)
```

Wyszukiwarka

Wyszukiwarka jest jedną z prostszych funkcjonalności do implementacji. Potrzebne będzie jedynie stworzenie pola tekstowego, gdzie użytkownik może wpisać wybraną frazę, oraz jego odpowiednie obsłużenie. Po wpisaniu frazy i wciśnięciu klawisza `Enter` użytkownik powinien zostać przeniesiony do strony wyszukiwarki z wyszukany wpisany już wcześniej hasłem.

Można zauważyć, że tym razem do modelu należy dołożyć jedynie jedną wartość — ciąg znaków wpisany przez użytkownika. Wiadomości typu `Msg` natomiast będą dwie — jedna standardowo odpowiedzialna za zaktualizowanie modelu, a druga za przekierowanie strony do wyszukiwarki. Przekierowanie odbywa się za pomocą funkcji `load` z biblioteki `Navigation`, która powinna już

być załączona do programu w początkowym szkielecie. Poniżej zostały przedstawione propozycje implementacji wiadomości `Search` obsługującej przekierowanie do wyszukiwarki oraz funkcji `viewSearchBar` wyświetlającej pole tekstowe. Dodatkowo zaimplementowana została funkcja `onEnter` pozwalająca na wykrycie czy użytkownik wcisnął klawisz `Enter`.

```
Search ->
  ( model
  , Nav.load ("https://google.com/search?q=" ++ model.searchText)
  )
```

```
onEnter msg =
  let
    isEnter code =
      if code == 13 then
        succeed msg
      else
        fail "not ENTER"
  in
    on "keydown" (andThen isEnter keyCode)
```

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

W ramach ćwiczenia należy zaimplementować model oraz rozszerzyć funkcję `update` o dwie nowe wiadomości, jak i główną funkcję `view` o przedstawioną wyżej propozycję `viewSearchBar`. Po prawidłowym złączeniu wszystkich fragmentów kodu program powinien się kompilować i wyświetlać nowy pasek pola tekstowego, a po wciśnięciu klawisza `Enter` strona powinna zostać przekierowana do wyszukiwarki Google.

Jeżeli wszystko udało się zaimplementować pomyślnie, można przejść do kolejnego kroku.

Zakładki

W dalszej części implementacji Do prawidłowego działania zakładek potrzebne będzie stworzenie dwóch dodatkowych plików — `bookmarks.js` oraz `index.html`. Pierwszy posłuży do trzymania listy zakładek, gdzie każda składa się z adresu URL oraz nazwy, która zostanie wyświetlona na tworzonej stronie. Natomiast drugi plik `index.html` potrzebny będzie do załadowania do programu poprzedniego pliku z zakładkami jak i pliku wynikowego stworzonego przez kompilator Elma z użyciem następującej komendy:

```
$ elm make src/MainBookmarks.elm --output=assets/elm.js
```

W sekcji `<head>` pliku `index.html` należy załączyć plik z zakładkami oraz wynikowy plik

kompilatora. W sekcji `<body>` należy dodać fragment kodu przedstawiony poniżej, który pozwoli na wykorzystanie programu w Elmie razem z zewnętrznym plikiem HTML.

```
<head>
  <script
    src="assets/elm.js">
  </script>
  <script
    src="assets/bookmarks.js">
  </script>
</head>
```

```
<body>
  <div id="app"></div>
  <script>
    var app = Elm.Main.init({
      node: document.getElementById("app
        "),
      flags: bookmarks,
    });
  </script>
</body>
```

Po otwarciu pliku `index.html` z użyciem programu `elm reactor` aplikacja wraz z zakładkami powinna pojawić się na ekranie.

Style

W załączonym pliku `styles.css` znajdują się kaskadowe arkusze stylów, które mogą zostać wykorzystane wraz ze stworzoną aplikacją. W tym celu należy odpowiednio zmienić w programie funkcje odpowiedzialne za wyświetlanie elementów, dodając do nich tagi `div[] []`. Funkcja `div` przyjmuje dwa argumenty: listę atrybutów oraz listę elementów HTML. W tej części należy skupić się na pierwszym argumencie, gdyż dotychczas nie był on jeszcze używany. Atrybuty HTML to np. `class`, `id` oraz `style`.

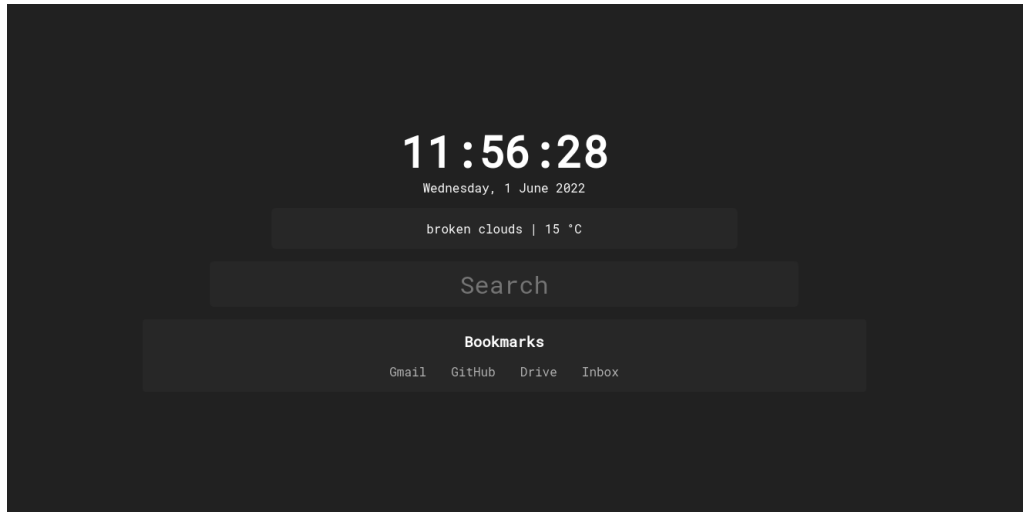
Na przykładzie głównej funkcji `view`, wykorzystanie klasy `container` wygląda następująco:

```
view model =
{ ...
, body =
  [ div [ class "container" ]
    [ viewTime model.clockTime
      ... ] ]
}
```

Jako, że plik `styles.css` został dostarczony w załączniku, w ramach ćwiczenia w aplikacji wystarczy jedynie dodać w odpowiednich miejscach funkcje `div` wraz z pasującymi atrybutami, które zostały zdefiniowane w arkuszu stylów.

Efekt

Jeżeli wszystkie funkcjonalności zostały prawidłowo zaimplementowane, a arkusze stylów właściwie użyte w programie, finalna aplikacja powinna wyglądać następująco:



Dodatek B: Kod źródłowy do instrukcji laboratoryjnej

Kod źródłowy do instrukcji laboratoryjnej znajduje się w repozytorium GitHub pod linkiem <https://github.com/marcinjurczak/mgr/tree/main/lab>.

Plik `src/MainHello.elm` zawiera wspomniany w instrukcji szkielet aplikacji, a pozostałe pliki źródłowe, tj. `src/MainClock.elm`, `src/MainWeather.elm`, `src/MainSearch.elm` oraz `src/MainBookmarks.elm` zawierają implementacje kolejnych funkcjonalności tworzonej aplikacji.