Politechnika Gdańska

Wydział Elektroniki,
Telekomunikacji i Informatyki

Katedra: Algorytmów i Modelowania Systemów

Imię i nazwisko dyplomanta: Marcin Jurczak

Nr albumu: 171641

Forma i poziom studiów: Stacjonarne jednolite studia magisterskie

Kierunek studiów: Informatyka

Specjalność: Algorytmów i Technologii Internetowych

Praca dyplomowa magisterska

Temat pracy:

Wykorzystanie języka Elm do tworzenia aplikacji frontendowych.

Title of thesis:

Programming the front-end applications with Elm language.

Opiekun pracy:

dr inż. Krzysztof Manuszewski

Data ostatecznego zatwierdzenia raportu podobieństw w JSA: TBA

Gdańsk, 2022

Streszczenie

Celem niniejszej pracy magisterskiej było stworzenie frontend'owej aplikacji internetowej z wykorzystaniem funkcyjnego języka Elm, porównanie tejże technologii z istniejącymi, bardziej powszechnymi rozwiązaniami tego typu, a także przygotowanie instrukcji laboratoryjnej, która mogłaby zostać wykorzystana w ramach zajęć *„Współczesne Aplikacje Programowania Funkcyjnego"* przeprowadzanych na Wydziale Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej. Wytworzona aplikacja to strona internetowa typu *startpage*, czyli startowa strona przeglądarki, zawierająca najpotrzebniejsze informacje, takie jak czas, pogoda oraz odnośniki do wyszukiwarki i najczęściej odwiedzanych stron.

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 use the Elm language to create a frontend web application, comparing this technology to existing, more popular solutions, as well as preparing a lab instruction, which could be used at *„Modern applications of functional programming"* class at Gdańsk University of Technology's Faculty of Electronics, Telecommunications and Informatics. The created application is a *startpage*, meaning a starting page of a web browser consisting of the most useful information, such as time, weather and references to search engine and the most visited websites.

Keywords: Elm, functional programming, web development

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

Spis treści

Wykaz najważniejszych skrótów

API	—	ang. Application Programming Interface, pol. interfejs programowania aplikacji
CD	—	ang. Continuous Deployment, pol. ciągłe wdrażanie
CI	—	ang. Continuous Integration, pol. ciągła integracja
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
SPA	—	ang. Single Page Application, pol. jednostronicowa aplikacja internetowa
DOM	—	ang. Document Object Model, pol. obiektowy model dokumentu

Wstęp i cel pracy

Głównym celem niniejszej pracy jest zapoznanie się z funkcyjnym językiem programowania Elm oraz stworzenie przykładowej frontendowej aplikacji internetowej. Ponadto chciałbym przeprowadzić porównanie tej technologii z innymi, bardziej powszechnie używanymi rozwiązaniami do tworzenia aplikacji internetowych. Ostatnim celem pracy jest przygotowanie części dydaktycznej w postaci instrukcji laboratoryjnej, która mogłaby zostać potencjalnie wykorzystana w ramach przedmiotu Współczesne Aplikacje Programowania Funkcyjnego, prowadzonego przez mojego promotora, dra inż. Krzysztofa Manuszewskiego.

Logo Elma

W drugim rozdziale skupiam się na przedstawieniu technologii powszechnie używanych do tworzenia frontendowych aplikacji internetowych, t.j. React.js, Angular oraz Vue.js. Prezentuję ich zalety i wady względem siebie, a także przedstawiam cechy charakterystyczne dla każdego z nich oraz co je odróżnia między sobą.

Trzeci rozdział poświęcam na wysokopoziomowe wprowadzenie do języka Elm. Mówię o idei jaka przyświęcała autorowi podczas tworzenia tego języka, jakie są jego potencjalne zastosowania, gdzie sprawdza się najlepiej oraz przedstawiam narzędzia wspomagające tworzenie oprogramowania z użyciem tejże technologii, włączając w to dokumentację.

W czwartm rozdziale przedstawiam implementację przygotowanej aplikacji frontendowej napisanej w Elmie. Jest to poniekąd rozwinięcie poprzedniego rozdziału, ponieważ głównym celem jest dalej zapoznanie się z Elmem, jednak tutaj uwagę skupiam na przedstawieniu konkretnych rozwiązań, jakie zostały wykorzystane do osiągniecia wybranego celu.

Piąty rozdział zawiera instrukcję laboratoryjną, w której przeprowadzam czytelnika nieposiadającego żadnego doświadczenia z Elmem przez proces tworzenia oprogramowania z wykorzystaniem tej technologii, zaczynając od przygotowania środowiska, przez podstawy języka, aż po stworzenie

aplikacji frontendowej przedstawionej we wcześniejszym rozdziale.

Szósty rozdział poświęcam na omówienie zagadnień związanych z ciągłą integracją oraz ciągłym wdrażaniem, a także przedstawiam narzędzia wykorzystywane przeze mnie w tym celu podczas tworzenia omawianej aplikacji. Pokazuję, 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. Przedstawiam produkt dwóch semestrów moich działań oraz wyciągam 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.

Powszechne rozwiązania

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

Według ankiety przeprowadzonej w 2021 roku przez StackOverflow [1], najczęściej wybieranym przez programistów frameworkiem do tworzenia stron internetowych był React.js (40,14% odpowiedzi), a na czwartym i piątym miejscu znajdowały się odpowiednio biblioteki Angular (22,96% odpowiedzi) oraz Vue.js (18,97% odpowiedzi). Są to biblioteki, na których chciałbym się skupić w poniższych podrozdziałach. Opiszę, czym się charakteryzują, jakie są ich wady i zalety, 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, 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.

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. Według dokumentacji Reacta [3], komponenty mogą być zdefiniowane na dwa sposoby: poprzez stworzenie JavaScript'owej 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 tre-

ści „*Hello,* ” + imię zawarte w zmiennej `props`.

Listing 2.1: Przykład funkcyjnego komponentu

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

Listing 2.2: Przykład klasowego komponentu

```
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 zaawansowych 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 określany jest jako „biblioteka”, a nie framework. Oznacza to

2.2 Angular

Biblioteka Angular [4] została stworzona 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. Bycie „nadzbiorem” oznacza, 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.

Głównym przeznaczeniem Angulara, podobnie jak w przypadku React.js, jest tworzenie aplikacji typu SPA. Jednak w przeciwieństwie do Reacta, jest to pełnoprawny *framework*. Oznacza to, że [4]

2.3 Vue.js

Biblioteka Vue.js [5]

2.4 Podobieństwa i różnice

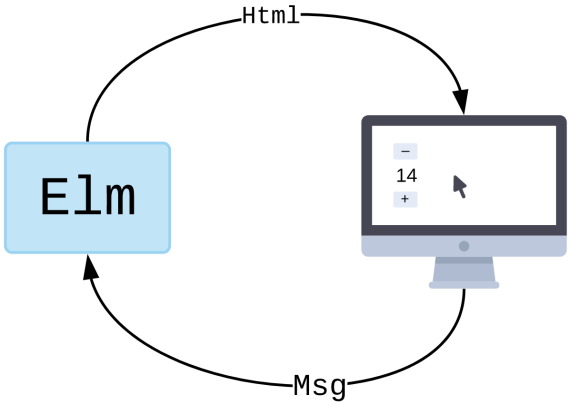
Elm

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.

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.

3.1 Programowanie funkcyjne

3.2 The Elm Architecture



Rysunek 3.1: Diagram działania programu w Elmie

The Elm Architecture jest schematem tworzenia interaktywnych aplikacji internetowych lub gier. Zgodnie z rysunkiem 3.1 typowa aplikacja Elm działa w następujący sposób: Program generuje pewien kod HTML, który zostaje wyświetlony na ekranie, a następnie komputer zwraca wiadomości

informujące o tym co się dzieje, np. użytkownik wcisnął guzik.

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

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

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

3.2.1 Model

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

Listing 3.1: *The Elm Architecture* — Model

```
type alias Model = Int



init : Model
init =
  0
```

3.2.2 Update

Listing 3.2: *The Elm Architecture* — Update

```
type Msg
  = Increment
  | Decrement



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


    Decrement ->
      model - 1
```

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

3.2.3 View

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

Listing 3.3: *The Elm Architecture* — View

```
view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

3.3 Narzędzia

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

`elm repl` — otwiera interaktywną sesję programistyczną.

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

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

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

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

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

`elm-live` [9] — podobnie jak `elm reactor`, uruchamia serwer deweloperski, jednak jest to znacznie bardziej rozbudowane narzędzie, oferujące m.in. takie funkcjonalności jak kompilowanie Elm'a do JavaScript'u, 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`.

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

Implementacja

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

- Cyfrowy zegar wskazujący aktualny czas w strefie czasowej użytkownika
- Aktualna data
- Pogoda w Gdańsku przedstawiona w formie krótkiego opisu oraz temperatury w stopniach Celsjusza
- 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 jest pokazanie różnic w implementacji obu

4.1 Elm

4.1.1 Model

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

Listing 4.1: Pełen model aplikacji

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

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

Kolejnym elementem modelu jest zmienna typu `WeatherStatus`, która może przyjąć jedną z trzech wartości — `Failure`, `Loading` lub `Success`. Odpowiada ona za trzymanie informacji o statusie zapytania HTTP do API dostarczanego przez serwis OpenWeather [10]. Początkowo inicjalizowany jest jako wariant `Loading`, w przypadku niepowiedzenia 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 machanizmu 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 : List Bookmark -> ( Model, Cmd Msg )
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 przymuje wartość `Loading`, a zakładki zostają przypisane bezpośrednio z argumentu funkcji.

Następnie należy wskazać wiadomości, które mają zostać przetworzone przez funkcję `update` i wykonane przez program na początku działania programu. W przypadku stworzonej aplikacji są

to dwie wiadomości — jedna odpowiedzialna za dostosowanie odpowiedniej strefy czasowej, druga za wysłanie zapytania HTTP w celu odebrania aktualnego stanu pogody.

4.1.2 Update

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

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

```
type Msg
   = Tick Time.Posix
   | AdjustTimeZone Time.Zone
   | UpdateWeather
   | GotWeather (Result Http.Error Weather)
   | UpdateField String
   | Search
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    Tick newTime ->
      ( { model | clockTime = ClockTime model.clockTime.zone newTime }
      , Cmd.none )
    AdjustTimeZone newZone ->
      ( { model | clockTime = ClockTime newZone model.clockTime.time }
      , Cmd.none )
    UpdateWeather ->
      ( { model | weatherStatus = Loading }
      , getWeather )
    GotWeather result ->
      case result of
        Ok weather ->
          ( { model | weatherStatus = Success weather }
          , Cmd.none )
        Err _ ->
          ( { model | weatherStatus = Failure "Error: Couldn't retrieve
              weather data" }
          , Cmd.none )
```

```
  UpdateField searchText ->
    ( { model | searchText = searchText }
    , Cmd.none )
  Search ->
    ( model
    , Nav.load ("https://google.com/search?q=" ++ model.searchText) )
```

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

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

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

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

Listing 4.4: Implementacja funkcji `getWeather`

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

Do implementacji dekodera użyta została biblioteka `elm/json`. Funkcja `weatherDecoder`,

przedstawiona na listingu 4.5, ma na celu przetworzenie odebranego pliku w formacie JSON tak, aby pasował do zdefiniowanego przez nas typu `Weather`, który ma zawierać krótki opis pogody (typ `String`) oraz temperaturę powietrza (typ `Float`).

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

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

Listing 4.5: Implementacja dekodera JSON

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

Listing 4.6: Przykład odebranego zapytania GET w formacie JSON

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

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

wiadomości odpowiednio zaktualizuje wartość `clockTime.time` w zdefiniowanym modelu.

Listing 4.7: Implementacja funkcji `subscriptions`

```
subscriptions : Model -> Sub Msg
subscriptions _ =
   Time.every 10 Tick
```

4.1.3 View

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

Listing 4.8: Implementacja funkcji `view`

```
view : Model -> Browser.Document Msg
view model =
   { title = "Startpage"
   , body =
      [ div [ class "container" ]
         [ viewTime model.clockTime
         , viewDate model.clockTime
         , viewWeather model.weatherStatus
         , viewSearchBar
         , viewBookmarks model.bookmarks
         ]
      ]
   }
```

4.2 React

Instrukcja laboratoryjna

W poniższym rozdziale przedstawiam przykładową instrukcję laboratoryjną, która krok po kroku przeprowadza 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.

5.1 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.

5.1.1 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 instukcję 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 5.1.

```
→ ~ 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 5.1: 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` [11], 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 5.1, 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 [12] 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 5.1, tak jak zostało to już wcześniej wspomniane.

5.1.2 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 i konfiguracji wtyczki dla edytora Visual Studio Code [13], ponieważ jest to jeden z najbardziej powszechnie używanych narzędzi do pracy z kodem źródłowym.

5.1.3 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 elm_project
$ cd elm_project
$ elm init
```

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

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

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

5.2 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 5.2 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 5.2: Otwarta interaktywna sesja `elm repl`

5.2.1 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 konkatenacji jest ++, a dla typów logicznych dostępne są operatory logiczne „`&&`" (AND) oraz „`||`" (OR).

Na rysunku 5.3 pokazane został przykłady efektów takich wywołań.

(a) Liczba **(b)** Ciąg znaków **(c)** Typ logiczny

Rysunek 5.3: Przykłady wartości w Elmie

5.2.2 Funkcje

Funkcje w Elmie określają, w jaki sposób wartości mogą zostać przetworzone. Na rysunku 5.4 pokazana została przykładowa funkcja `hello`, która przyjmuje argument `name` i zwraca nowy ciąg znaków.

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

Rysunek 5.4: Przykładowa funkcja w Elmie

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

Przykład takiego komunikatu został przedstawiony na rysunku 5.5.

```
> 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 5.5: Komunikar kompilatora o błędzie

5.2.3 Listy

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

```
> 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 5.6: Przykładowe operacje na listach

5.2.4 Rekordy

5.3 Aplikacja frontendowa

W ramach większego projektu zbudujemy stronę internetową typu `startpage`, czyli strony startowej przeglądarki zawierającej najważniejsze i najczęściej używane elementy. W naszym przypadku będą to cztery moduły — zegar, 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.

5.3.1 Zegar

5.3.2 Pogoda

5.3.3 Wyszukiwarka

5.3.4 Zakładki

5.3.5 Dokument hipertekstowy

Listing 5.1: Zawartość pliku `index.html`

```
<head>
  <link rel="stylesheet" href="assets/styles.css" />
  <script src="assets/main.js"></script>
  <script src="assets/bookmarks.js"></script>
</head>
<body>
  <div id="myapp"></div>
  <script>
    var app = Elm.Main.init({
      node: document.getElementById("myapp"),
      flags: bookmarks,
    });
  </script>
</body>
```

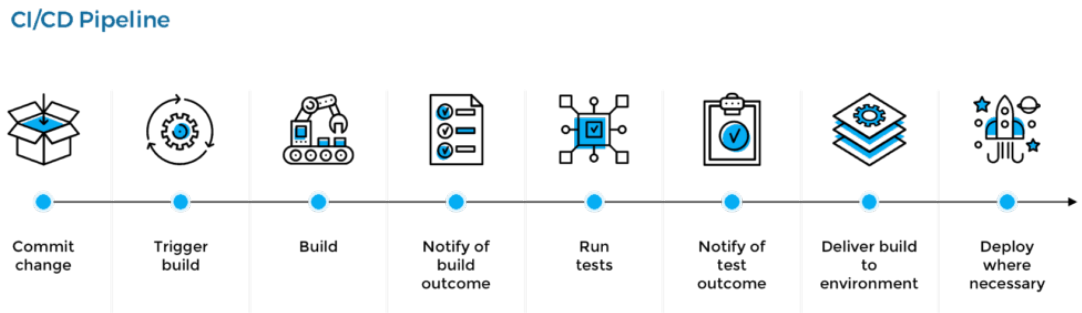
5.3.6 Style

Automatyzacja

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

6.1 CI/CD

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



Rysunek 6.1: Przykładowy potok CI/CD

Na rysunku 6.1 przedstawiony został przykład potoku CI/CD.

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.

Ciągłe wdrażanie jednakże skupia się dostarczeniu produktu do użytkownika końcowego, aby uzyskać informację zwrotną od klienta. W następnej sekcji postaram się opisać procesy CD w sposób bardziej szczegółowy.

6.1.2 Ciągłe wdrażanie

6.2 GitHub Actions

Od samego rozpoczęcia pracy nad aplikacją w Elmie, cały kod źródłowy przechowywany był z użyciem systemu kontroli wersji Git [14], a wybór serwisu hostującego zdalne repozytorium kodu padł na GitHub [15]. Ze względu na ten wybór jak i niewielki rozmiar stworzonej w Elmie aplikacji, zdecydowałem się wykorzystać GitHub Actions do zbudowania potoków CI/CD.

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

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

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 być zbudowana i 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 dodoawany jest otrzymany podczas przeprowadzonej wcześniej 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.

Podsumowanie

7.1 Wnioski

Spis rysunków

Spis listingów

Bibliografia

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

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

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

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

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

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

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

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

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

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

[11] Isaac Z. Schlueter. *npm documentation*. URL: https://docs.npmjs.com/ (term. wiz. 17. 05. 2022).

[12] Evan Czaplicki. *Elm installer*. 2019. URL: https://github.com/elm/compiler/releases.

[13] *Visual Studio Code documentation*. Microsoft. URL: https://code.visualstudio.com/docs (term. wiz. 17. 05. 2022).

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

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