WYDZIAŁ:	WYDZIAŁ STUDIÓW STRATEGICZNYCH I TECHNICZNYCH
KIERUNEK:	INFORMATYKA
SEMESTR:	III ROK, VI SEMESTR, TRYB NIESTACJONARNY

PRZEDMIOT

TECHNOLOGIE INTERNETOWE - LABORATORIUM

Prowadzący: doktor Dobosz Piotr

LABORATORIUM 1 I 2

TEMAT: STRONA WWW - PORTFOLIO

IMIĘ I NAZWISKO STUDENTA:	**DAGMARA OSZMIAŃCZUK**
NUMER INDEKSU:	17155
IMIĘ I NAZWISKO STUDENTA:	**KAROLINA ZIELIŃSKA**
NUMER INDEKSU:	16526

RADOM 2022

ZALICZENIE - OCENA	PODPIS PROWADZĄCEGO	UWAGI

CEL ĆWICZENIA

1. Cel zadania:

Celem niniejszego laboratorium było utworzenie strony WWW dla określonej organizacji oraz zapoznanie się z prototypowaniem aplikacji.

INFORMACJE WSTĘPNE

1. Informacje wstępne:

Wykonany projekt jest stroną przedstawiającą portfolio grafika komputerowego. Jest to temat pozwalający zaprezentować wiele problemów dotyczących projektowania stron internetowych. Głównym zagadnieniem poruszanym w stworzonym projekcie jest praca z CMS – Content Management System. Podejście takie pozwala w prosty sposób zarządzać treścią znajdującą się na zarządzanej stronie. Jest to szczególnie przydatne, kiedy tworzymy aplikację dla osoby nietechniczej, dla której modyfikacja kodu byłaby zbyt dużym wyzwaniem. W stworzonym projekcie wykorzystano Headless CMS pozwalający na zarządzanie jedynie treścią lub zawartością poszczególnych elementów aplikacji, co umożliwia własnoręczną implementację warstwy widoku aplikacji. Jest to wygodne w użyciu rozwiązanie ze względu na oferowaną swobodę przy projektowaniu interfejsu graficznego, a także dostępnych funkcjonalności. Dodatkowym atutem jest brak korzystania z technologii WYSIWYG – What You See Is What You Get, która z początku może wydawać się, że eliminuje potrzebę pisania kodu strony, jednak po przeanalizowaniu wygenerowanych schematów często okazuje się, że nie są one stworzone w poprawny sposób.

Dodatkowym problemem przedstawionym w projekcie była komunikacja użytkownika z serwerem. Tworząc strony internetowe zależy nam na daniu użytkownikowi możliwości dokonywania różnych operacji. Wtedy stworzona aplikacja jest nie tylko statyczną stroną, ale umożliwia również interakcję, która często pełni funkcję automatyzacji różnych procesów. Nadaje to tworzonej aplikacji cel, który jest czymś więcej niż możliwość odczytu statycznych dokumentów tekstowych zamieszczonych przez właściciela strony. W celu zapewnienia takiej interakcji zazwyczaj potrzebny jest serwer, który jest jedną z dwóch stron potrzebnych do komunikacji.

Obsługa takiej komunikacji opiera się na odpowiedniej obsłudze żądań wysyłanych przez użytkownika. W stworzonym projekcie została stworzona usługa w architekturze REST – Representational State Transfer. Polega to na pracy serwera, nastawionej na nasłuchiwanie odpowiedniego kanału komunikacji (oczekuje na żądania od użytkownika). Typy takich żądań mogą być spokrewnione z tym, czego oczekujemy od serwera po otrzymaniu takiego żądania. Żądania typu GET pozwalają na pobieranie informacji z serwera, POST lub PUT na przesyłanie nowych danych, a DELETE do usuwania wybranych danych. Są to najczęściej wykorzystywane typy żądań, które pozwalają na obsługę większości potencjalnych funkcjonalności.

WYKAZ UŻYTYCH NARZĘDZI

a) Środowisko uruchomieniowe Node.js w wersji 16.13.
b) Biblioteka React w wersji 17.0.1
c) Szkielet aplikacyjny Gatsby w wersji 4.10.3
d) Szkielet aplikacyjny ExpressJS w wersji 4.17.3
e) Contentful Headless CMS

DOKUMENTACJA TECHNICZNA

Dokumentacja zawiera funkcjonalne i techniczne wymagania do kodu aplikacji, przedstawiającej szablon portfolio przeznaczonego dla grafika komputerowego

Wersja: 1.0.2022

Spis treści

Dokumentacja została opracowana przez:
Dagmara Oszmiańczuk,
Karolina Zielińska,

Data publikacji: XX.XX.2022

1. Opis ogólny

1.1 Cel

Celem tego dokumentu jest dostarczenie informacji, które umożliwią stworzenie, rozwój, utrzymanie, hosting i obsługę aplikacji przestawiającej portfolio grafika komputerowego.

1.2 Zakres i możliwości

Budowa strony:

- Nagłówek u góry strony zawierający nawigację z odnośnikami do różnych sekcji strony.
- Stopka strony z informacją o prawach autorskich oraz bieżącym rokiem.
- Ekran powitalny przedstawiający imię i nazwisko osoby, której dotyczy portfolio wraz z krótkim motto.
- Opis grafika wraz ze zdjęciem oraz głównymi aspektami jego pracy.
- Sekcja oferty zawierające detale dotyczące oferowanych usług.
- Formularz kontaktowy pozwalający na wysłanie wiadomości email do właściciela strony.
- Sekcja portfolio zawierająca wszystkie dotychczasowe prace grafika.

Możliwe przypadki użycia:

- Uzyskanie podstawowych informacji o właścicielu strony.
- Wyświetlenie galerii wykonanych prac graficznych.
- Filtrowanie prac graficznych po kategoriach.
- Wyświetlenie pracy graficznej na pełnym ekranie.
- Przełączanie się między pracami graficznymi na powiększonym ekranie.
- Wysłanie wiadomości do właściciela strony poprzez formularz na stronie.

1.3 Definicje

Właściciel strony – osoba, która jest w posiadaniu stworzonej aplikacji internetowej oraz której ona dotyczy.

JS – JavaScript – interpretowany, skryptowy język programowania wykorzystany do zapewnienia interakcji od strony użytkownika poprzez ingerencję w struktury DOM strony internetowej, komunikację z serwerem CMS zawierającym dane zawarte na stronie internetowej oraz obsługę serwera pozwalającego na obsługę żądań związanych z wysyłaniem wiadomości mailowych.

DOM – Document Object Model – zapewnia reprezentację struktur dokumentów HTML jednocześnie łącząc ją z warstwą skryptów JS pozwalając na interakcję z poszczególnymi elementami.

NPM – manager pakietów środowiska node.js pozwalający za zarządzanie potrzebnymi bibliotekami oraz ich zależnościami. Wykorzystywane biblioteki składowane są w katalogu *node_modules*.

SCSS – język pozwalający na wygodniejsze i czytelniejsze tworzenie kodu CSS odpowiadającego za warstwę arkuszy stylów.

HTML – hipertekstowy język znaczników, wykorzystywany do tworzenia dokumentów hipertekstowych.

React – biblioteka JavaScript pozwalająca na tworzenie aplikacji internetowych w sposób komponentowy. Pozwala na tworzenie aplikacji w standardzie Single Page Application pozwalając na dynamiczne ładowanie tylko tej zawartości strony, która ulega zmianie, czego rezultatem jest ciągła operacja na jednym pliku HTML w obrębie stworzonej aplikacji.

CMS – Content Management System – system zarządzania treścią pozwalający na modyfikację treści strony internetowej bez potrzeby ingerencji w kod strony, a także opcję zarządzania warstwą prezentacji. Zastosowanym rozwiązaniem jest Headless CMS ograniczający się jedynie do pierwszej z dwóch podanych funkcjonalności.

Serwer REST / Backend – Warstwa aplikacji odpowiadająca za obsługę otrzymywanych od klienta żądań.

Aplikacja widoku klienta / Frontend – Warstwa aplikacji odpowiadająca za zapewnienie interakcji z użytkownikiem oraz szatę graficzną stworzonego projektu.

GraphQL – Język budowania zapytań dostępu do danych oraz ich manipulacji.

2. Projekt

2.1 Wstęp do struktury projektu

Stworzony projekt opiera się na dwóch podprojektach znajdujących się w bazowym katalogu projektu. W folderze *backend* znajdują się pliki odpowiadające za działanie *Serwera REST* obsługującego przychodzące żądania. W folderze *client* znajdują się z kolei pliki odpowiadające za działanie *aplikacji widoku klienta.* Każdy z podprojektów został stworzony przy pomocy środowiska uruchomieniowego *node.js*. Oznacza to, że podprojekty te ściśle powiązane są ze znajdującymi się w nich plikach *package.json* określającymi wymagane do ich działania zależności. Pliki te zawierają też skrypty przydatne w trakcie tworzenia aplikacji takie jak skrypt *start* wykorzystywany dalej w punkcie *2.2*.

Dodatkowo podprojekt odpowiadający za *Frontend* stworzony został przy wykorzystaniu podejścia modułowego dodanego w wersji *ES6* lub inaczej *ES2015* będącego jednym ze standardów opisujących język *JavaScript.*

W katalogu głównym znajduje się również plik *.prettierrc* zawierający konfigurację narzędzia *prettier* wykorzystanego do automatycznego formatowania pisanego kodu podczas zapisu plików, co wpłynęło na jakość i szybkość tworzonego kodu.

2.2 Uruchomienie projektu

Do działania projektu wymagane jest zainstalowanie środowiska uruchomieniowego *nodejs* wraz z managerem pakietów *npm*. Wymagane jest również zainstalowanie globalnie pakietu *gatsby-cli* pozwalającego na uruchamianie aplikacji widoku klienta i zarządzanie nią.

W celu uruchomienia projektu (w celu przetestowania działania) należy uruchomić kolejno oba podprojekty pozwalając na równoległe działanie zwrówno *backendu* jak i *frontendu.* W tym celu należy udać się do katalogu *backend* znajdującego się w katalogu głównym projektu, a następnie wykonać polecenie *npm install*, które pobierze wszystkie potrzebne zależności i umieści je w katalogu *node_modules.* Po pomyślnej instalacji pakietów należy wykonać polecenie *npm start* wykonujące skrypty potrzebne do uruchomienia aplikacji. Aplikacja zostanie uruchomiona na porcie 8800, co potwierdzi wyświetlona przez aplikację informacja.

```
\backend> npm install
(...)

\backend> npm start
> backend@1.0.0 start
> nodemon index.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
App running on port 8800
```

Następnie należy wykonać te same kroki w podprojekcie *client* pozostawiając proces odpowiadający za *backend* włączony. W tym celu należy udać się do katalogu *client* i wykonać następujące polecenia.

```
\client> npm install
(...)

\client> npm start
> portfolio-demo@1.0.0 start
> gatsby develop
(...)

You can now view portfolio-demo in the browser.

  http://localhost:8000/

View GraphiQL, an in-browser IDE, to explore your site's data and schema

  http://localhost:8000/___graphql

Note that the development build is not optimized.
To create a production build, use gatsby build

success Building development bundle - 14.678s
success Writing page-data.json files to public directory - 0.189s - 1/2 10.58/s
```

Aplikacja widoku klienta została uruchomiona na porcie 8000. Możemy również zauważyć adres przenoszący do konsoli pozwalającej w graficzny sposób budować zapytania *GraphQL* z poziomu przeglądarki na adresie *http://localhost:8000/__graphql*.

Po przejściu na adres *http://localhost:8000/* zostaniemy przeniesieni na stronę główną stworzonej aplikacji.

2.3 Analiza struktury podprojektu *backend*

Głównym korzeniem aplikacji jest plik *backend/index.js* zawierający główną konfigurację serwera REST stworzonego za pomocą szkieletu aplikacyjnego *expressJS.*

Został również wydzielony katalog *routes* przeznaczony na podzielenie podprojektu na moduły w celu zachowania porządku w stworzonym kodzie. W katalogu *routes* znajduje się katalog *email* zawierające punkty końcowe obsługujące żądania związane z wysyłaniem wiadomości email.

Pomimo, że w stworzonym projekcie znajduje się tylko jeden moduł – *email*, zastosowany podział na moduły nie tylko wprowadza porządek w stworzonym kodzie, ale również zapewnia możliwość dalszego rozwoju aplikacji.

W katalogu *backend* znajduje się również plik *.env* zawierający zmienne środowiskowe tworzone dla projektu na czas jego działania. Został on utworzony w celu przechowywania danych dostępowych do konta poczty potrzebnych do obsługi modułu *email*. Ze względu na fakt, że zawiera dane wrażliwe nie został umieszczony na repozytorium.

Struktura pliku *.env:*

```
EMAIL_USERNAME = adres_email_konta
EMAIL_PASSWORD = hasło_dostępu
```

Poniżej zaprezentowano strukturę podprojektu */backend*.

```
\backend> tree /F
├───node_modules
│   └─── (...)
│   .env
│   index.js
│   package-lock.json
│   package.json
└───routes
    └───email
            emailRouter.js
template.html
```

2.4 Analiza struktury podprojektu *client*

Podprojekt *client* zawiera w sobie katalogi *src* – zawierające kod źródłowy aplikacji oraz *public* – zawierający pliki wygenerowane przez szkielet aplikacyjny Gatsby. W katalogu głównym znajdziemy również pliki konfiguracyjne takie jak *gatsby-config.js,* czy *package.json,* ale także pliki *style.scss* oraz *variables.scss* będące podstawą zastosowanej struktury plików SCSS zawierające kolejno globalne style dla całej aplikacji i zmienne wykorzystywane w innych plikach zawierających arkusze stylów.

Wykorzystany szkielet aplikacyjny Gatsby poprzez oferowane przez siebie funkcje wymusza pewien schemat architektury projektów. W przeciwieństwie do projektów pisanych jedynie w oparciu o bibliotekę React, nie mamy tutaj do czynienia z plikiem *index.js* znajdującym się w katalogu głównym podprojektu, będącym korzeniem aplikacji. Zamiast tego framework Gatsby narzuca konwencję tworzenia plików zawierających zawartość poszczególnych stron w katalogu client/src/*pages,* jednocześnie mapując utworzone tam pliki z podstronami naszej aplikacji. Rozwiązanie takie zastępuje konieczność ręcznej konfiguracji routingu za pomocą biblioteki *react-router-dom*.

W katalogu *src/components* znajdziemy komponenty wykorzystywane na poszczególnych stronach. Każdy komponent posiada swój dedykowany katalog, który oprócz pliku JavaScript z kodem źródłowym, zawiera także plik zawierający arkusz stylów powiązany z danym komponentem. Podział taki pozwala na lepszą organizację i przyspiesza poszukiwanie odpowiednich reguł CSS, które zostały już zastosowane dla poszczególnych komponentów.

Katalog *src/hooks* zawiera niestandardowe, stworzone na potrzeby aplikacji pliki *hooków* będących integralną częścią stosowanej biblioteki *React* oraz pozwalających na izolację logiki pewnych aspektów modelu do osobnych plików. Wykorzystanie takich *hooków* odbywa się przy pomocy wykorzystania destrukturyzacji krotek.

Stworzona aplikacja wykorzystuje do swojego działania również statyczne pliki obrazów umieszczanych w różnych miejscach na stronie. Są to obrazy takie jak tło strony czy też ikona aplikacji, tzw. *favicon.* W tym celu został przeznaczony katalog *src/images*.

Dodatkowo został stworzony również plik *src/util.js* zawierający funkcje pomocnicze, wykorzystywane przy tworzeniu aplikacji, jednak niepowiązane ściśle z jednym komponentem.

Wspomniany wcześniej katalog *public* zawiera pliki wygenerowane przez Gatsby na etapie uruchamiania aplikacji. Przechowuje on dane dotyczące ikony aplikacji *favicon* generując jednocześnie wiele potrzebnych dla różnych platform rozmiarów, a także dane dotyczące poszczególnych stron powiązanych z katalogiem */src/pages* z naciskiem na powiązane z nimi ścieżki, oraz wykorzystywane w nich zapytania *GraphQL*.

Poniżej zaprezentowano strukturę podprojektu */client* z zastosowaniem pewnego uproszczenia przy katalogach *public* ze względu na dużą ilość wygenerowanych plików oraz katalogu *src/components*, gdzie w każdym z jego podkatalogów znajduje się plik JavaScript o tej samej nazwie, a także plik *SCSS* zawierający arkusz stylów danego komponentu.

```
\client> tree /F
├───node_modules
│   └─── (...)
│   gatsby-config.js
│   package-lock.json
│   package.json
│   README.md
│   style.scss
│   variables.scss
├───public
│   │   favicon-32x32.png
│   │   manifest.webmanifest
│   │
│   ├───icons
│   ├───page-data
│   └───static
└───src
    │   util.js
    │
    ├───components
    │   ├───about
    │   ├───contact
    │   ├───footer
    │   ├───gallery
    │   ├───galleryFullScreen
    │   ├───layout
    │   ├───main
    │   ├───navbar
    │   └───offer
    ├───hooks
    │       useTags.js
    │
    ├───images
    │   │   icon.png
    │   │
    │   └───bg
    │           doodles.png
```

```
    │           main.png
    │
    └───pages
            404.js
            index.js
            photos.js
```

3. Dokumentacja kodu

3.1 Podprojekt *backend*

Stworzona aplikacja wymaga serwera obsługującego żądania przesyłane od klienta. W tym celu została zastosowana architektura *REST* z użyciem szkieletu aplikacyjnego *ExpressJS*.

W prezentowanym przykładzie stworzony został jeden punkt końcowy obsługujący żądania typu POST na adres *http://localhost:8800/api/email* pozwalający na wysłanie wiadomości mailowej do właściciela strony, przekazując w sekcji *body* dane takie jak *from, title* oraz *content.* Poniżej zaprezentowano ich znaczenie:

from	*adres email nadawcy wiadomości*
title	*tytuł przesyłanej wiadomości (przedział 4-32 znaki)*
content	*zawartość przesyłanej wiadomości (przedział 16-8000 znaków)*

Obsługa żądania odbywa się za pomocą metody *post* oferowanej przez obiekt *emailRouter* klasy *Router* oferowanej przez *ExpressJS.* Obiekt ten musi zostać zmapowany do odpowiedniego adresu URL, który będzie prefiksem do wszystkich adresów żądań tego routera. Mapowanie to odbywa się w pliku *index.js* przy pomocy metody *use* na obiekcie *app* będący instancją aplikacji zgodnie z biblioteką *ExpressJS.*

```
app.use("/api/email", emailRouter)
```

Metoda *post* obiektu *emailRouter* przyjmuje dwa argumenty – adres URL żądania oraz funkcję anonimową, służącą do obsługi danego żądania.

```
emailRouter.post("/", (req, res)=> {
  (...)
})
```

W funkcji tej pobierane są wymienione wcześniej pola sekcji *body*, które następnie wykorzystywane są przy obsłudze funkcji *sendMail* oferowanej przez bibliotekę *nodemailer* pozwalającej na wysyłanie wiadomości mailowych.

```
transporter.sendMail(generateMailOptions(from, title, html), (error)=> {
  (...)
})
```

Biblioteka *nodemailer* wymaga jednak podania konfiguracji zawierającej dane potrzebne do uwierzytelnienia konta, z którego będą wysyłane wiadomości. W tym celu zastosowano mechanizm zmiennych środowiskowych w celu uniknięcia wprowadzenia wrażliwych danych do kodu aplikacji.

```
const transporter = nodemailer.createTransport({
   service:"gmail",
   auth: {
      user: process.env.EMAIL_USERNAME,
      pass: process.env.EMAIL_PASSWORD,
   },
})
```

Wysyłana wiadomość mailowa zawiera zawartość w formacie HTML. W celu zapewnienia takiej funkcjonalności stworzono plik z szablonem treści wiadomości *routes/email/template.html*, który jeszcze przed procesem wysłania wiadomości zostaje poddany modyfikacji, umieszczając w nim dane przesłane w żądaniu w miejscach im poświęconym. W celu stworzenia takiego szablonu została zastosowana biblioteka *handlebars.*

Fragment pliku szablonu wiadomości *template.html:*

```
<h3>Nadawca: {{from}}</h3>
<h3>Temat: {{title}}</h3>

<hr />
<p>{{content}}</p>
```

Podczas obsługi żądań kierowanych do naszej aplikacji należy pamiętać jednak o odpowiedniej walidacji przesyłanych danych. W tym celu w ciele funkcji obsługującej żądanie zostały stworzone dwie zmienne do obsługi błędów – *errors* będące tablicą obiektów z informacjami o błędach oraz *errorCode* zawierający odpowiedni, zwracany kod błędu. W analizowanym przykładzie walidacji poddawane są wszystkie przesyłane dane sprawdzając, czy pole *from* zawiera poprawną składnię adresu email, a także czy pola *title* i *content* mieszczą się w dopuszczalnej długości liczby znaków.

W przypadku, gdy w trakcie trwania funkcji zostanie zgłoszony przynajmniej jeden błąd, zostanie zwrócona odpowiedź o odpowiednim kodzie błędu z informacjami o błędzie w formacie *JSON*. W przeciwnym wypadku zostanie zwrócona odpowiedź o statusie 200 sygnalizując pomyślne wykonanie żądania.

```
if (errors.length >0) {
   res.status(errorCode).json(errors)
}

res.status(200).json({
   msg:"Wiadomość została pomyślnie wysłana",
})
```

3.2 Aplikacja *Single Page Application*

Zastosowana w podprojekcie *client* biblioteka *React* pozwala na tworzenie aplikacji internetowych w standardzie *Single Page Application*. Oznacza to dla użytkownika to, że podczas przełączania się między podstronami naszej aplikacji nie jest pobierany nowy plik HTML, a jedynie ta zawartość strony, która ulega zmianie. Przyspiesza to działanie samej aplikacji, a jedną z tego konsekwencji jest komponentowe podejście do tworzenia aplikacji.

Wykorzystanie komponentów pozwala programiście na definiowanie powtarzających się elementów strony tylko w jednym miejscu oraz możliwość wywoływania tych komponentów w różnych miejscach na tworzonej stronie. Jest to szczególnie przydatne w przypadku komponentów zawierających np. nawigację, która jest w większości przypadków niezmienna dla większości stron. W przypadku potrzeby modyfikacji elementu takiej nawigacji nie jesteśmy zmuszeni pamiętać o wszystkich miejscach, w których jest ona zaimplementowana, a jedynie zmienić zawartość pliku z komponentem wspomnianej nawigacji.

Do tworzenia komponentów wykorzystywanych na tworzonych stronach wykorzystano komponenty funkcyjne, będące zgodne z nowymi standardami biblioteki *ReactJS*. Podejście takie wymusza korzystanie z tzw. *hooków* będących istotną częścią biblioteki, które ściśle związane są ze sposobem w jaki komponenty są rerenderowane w przypadku zmiany wartości poszczególnych zmiennych.

```
const [galleryIsOpen, setGalleryIsOpen] =useState(false)
const [currentPhoto, setCurrentPhoto] =useState(0)
```

W prezentowanym wyżej przykładzie pokazano deklarację *hooków useState* z wskazaniem zmiennych przechowujących wartości – *galleryIsOpen* oraz *currentPhoto,* ich setterów – *setGalleryIsOpen* oraz *setCurrentPhoto*, oraz wartości początkowych będącymi argumentami wykorzystanymi przy wywołaniu funkcji *useState.*

Biblioteka *ReactJS* pozwala nam na wykorzystywanie składni *JSX* do tworzenia komponentów. Składnia ta pozwala nam na umieszczanie kodu HTML w komponentach oraz dynamiczne umieszczanie w nich wartości wybranych zmiennych. W celu umieszczenia fragmentu kodu *JavaScript,* w takim fragmencie *JSX* wykorzystywane są nawiasy klamrowe. Poniżej zaprezentowano fragment komponentu *footer* zawierającego stopkę strony. Warto wspomnieć o konieczności zamiany atrybutów takich jak *class* na *classname* ze względu na konflikt składni języków HTML i JS.

```
return (
      <footer>
         <divclassName="copyright">
            <p>
               Copyright © by <spanclassName="person">{person}</span>{currentYear}
            </p>
         </div>
      </footer>
   )
```

3.3 Komunikacja z serwerem CMS

Głównym aspektem tworzonej aplikacji jest komunikacja z serwerem *Contentful* będący jednym z systemów *Headless CMS*. Pozwoliło to na przechowywanie danych umieszczanych na stronie takich jak tekstowe opisy w poszczególnych sekcjach lub zdjęcia wyświetlane w portfolio wraz z przypisanymi im kategoriami wykorzystywanymi przy filtrowaniu (rozwiązanie to zastąpiło bazę danych).

Struktury danych na serwerze *Contentful* przypominają stosowanie klas i obiektów. Odpowiednikami klas jest w tym przypadku model znajdujący się w zakładce *Content Model* panelu administracyjnego, a z kolei odpowiednikiem obiektów są dane znajdujące się w zakładce *Content*.

W ten sposób zostały stworzone poszczególne struktury:

About	*Elementy sekcji „o mnie”*
General	*Statyczne opisy na stronie*
Images	*Zdjęcia do sekcji portfolio*
ImagesTags	*Tagi do filtrowania zdjęć w portfolio*
Offer	*Zawartość bloków ofert na stronie głównej*
SingleImages	*Pojedyncze zdjęcia powiązane z danymi sekcjami*

Każda z utworzonych struktur posiada specjalne dla siebie pola, które muszą zostać wypełnione podczas dodawania nowej zawartości. Dzięki tej mechanice możemy sprawić, że zawartość należąca do modelu Images będzie musiała zawierać po jednym zdjęciu, krótkim tytule, opisie oraz liście tagów potrzebnych do filtrowania.

W celu załączenia wybranej zawartości na stronie zostały wykorzystane zapytania GraphQL, na których w dużej mierze opiera się szkielet aplikacyjny Gatsby. Zapytania te można generować w graficzny sposób w panelu administracyjnym pod adresem *http://localhost:8000/__graphql.* Składnia tych zapytań pozwala w intuicyjny sposób zauważyć struktury danych. Pozwala też na filtrowanie otrzymywanych wyników przy pomocy składni takich jak *in* zwracających obiekty o polu znajdującej się we wskazanej liście lub *eq* zwracający obiekty o polu z wartością taką jak podana. Przykładowe wykorzystanie zapytania zaprezentowano na listingu poniżej:

```
const data = useStaticQuery(graphql`
   queryContactQuery{
      allContentfulGeneral(filter: { title: { in: ["Email", "ContactDescription"] }}){
            nodes {
               childContentfulGeneralValueTextNode {
                  value
               }
               title
            }
         }
      }
   `)
```

Przy wykorzystaniu hooka oferowanego przez Gatsby – *useStaticQuery* możemy pobierać zawartości struktur CMS. W tym przypadku zostanie zwrócony obiekt zawierający pola ściśle powiązane z zastosowaną składnią. Na przykład w celu odniesienia się do wartości *value* obiektu z tytułem „Email" należy użyć poniższego zapisu

```
data.allContentfulGeneral.nodes[0].childContentfulGeneralValueTextNode.value
```

Może to być stosunkowo niewygodne przy ciągłym wykorzystaniu dlatego została stworzona funkcja *getValueFromGeneral* zapisana w pliku *src/util.js* przyjmujący dwa parametry – dane otrzymywane w wyniku zapytania oraz poszukiwana wartość *title*.

```
exportconstgetValueFromGeneral=(data, name)=> {
   return data.allContentfulGeneral.nodes.filter(
      (node)=> node.title === name
   )[0].childContentfulGeneralValueTextNode.value
}
```

Czasami konieczna jest praca na tablicy np. w przypadku, gdy operujemy na zdjęciach wyświetlanych w portfolio. Chcąc otrzymać wszystkie obrazy do wyświetlenia otrzymujemy listę, po której należy następnie przeiterować za pomocą metody *map* obiektu *Array* oferowanego przez JavaScript, który zwraca nową tablicę. W sposób zaprezentowany poniżej uzyskano tablicę znaczników JSX przedstawiającą kod HTML zawierający pojedyncze obrazy.

```
   {data.allContentfulImages.nodes.map((node, index)=> {
      if (tags[0].active ||getActiveTags().includes(...node.tags)) {
         return (
            <div
               className="img-wrapper"
               role="button"
               tabIndex={index}
               onClick={()=> {
                  setGalleryIsOpen(true)
                  setCurrentPhoto(index)
               }}
               key={node.id}
            >
               <img
                  src={node.image.gatsbyImageData.images.fallback.src}
                  alt={node.title}
               />
               <divclassName="img-description">
                  <h1>{node.title}</h1>
                  <p>{node.description}</p>
               </div>
            </div>
         )
      }
      returnnull
   })}
```

3.4 Responsywność

Podczas projektowania stron internetowych należy mieć na uwadze fakt, że większość użytkowników będzie korzystało z naszej aplikacji z urządzenia mobilnego. Z tego względu należy zapewnić obsługę responsywności dla tworzonej strony. Oznacza to nie tylko odpowiednie skalowanie zawartości poszczególnych elementów, ale także zmianę sposobu ich wyświetlania w zależności od szerokości ekranu. Dobrym tego przykładem jest wyświetlanie nawigacji. Poniżej szerokości ekranu równej 668px lista z łączami do innych podstron naszej aplikacji znika, a na jej miejsce pojawia się przycisk pozwalający rozwinąć już nie poziomą, a pionową listę łączy.

Implementacja poszczególnych stylów związanych z responsywnością została zapewniona przy pomocy mechanik *mixin* i *include* oferowanych przez SCSS. Dzięki temu możemy w intuicyjny sposób definiować zachowanie elementów przy poszczególnych zakresach szerokości zdefiniowanych w pliku *variables.scss*. Należy przy tym jednak pamiętać o działaniu kaskadowości, więc stylowanie definiowane jest najpierw dla najmniejszych szerokości, a dopiero potem reguły CSS nadpisywane są dla większych rozdzielczości. Poniżej zaprezentowano fragment arkusza stylów dla komponentu *navbar*.

```
button{
   padding:1rem2rem;
   height:100%;
   align-items: center;
   cursor: pointer;
}

@includesm{
   button{
      display: flex;
   }
}

@includelg{
   button{
      display: none;
   }
}
```

Oprócz zastosowania przedziałów szerokości zastosowano również jednostki *rem* zastępujące w tym przypadku jednostkę *px*. Jednostki *rem* odnoszą się do rozmiaru czcionki dokumentu (domyślnie 16px), dzięki czemu deklarujemy sposób skalowania, a zmiana głównego rozmiaru czcionki dokumentu pozwoli nam zmienić dokładne wartości wszystkich wartości, gdzie została użyta jednostka *rem.* Przyspiesza to proces tworzenia responsywnej strony internetowej ograniczając jednocześnie ilość potrzebnych linii kodu wykorzystujących mechanikę przedziałów szerokości.

3.5 Obsługa żądań po stronie aplikacji widoku klienta

W celu zapewnienia poprawnego działania funkcjonalności wykorzystujących komunikację z serwerem REST należy zapewnić także wysyłanie żądań po stronie aplikacji widoku klienta. W tym celu wykorzystuje się funkcję *fetch* oferowaną przez silniki przeglądarek internetowych. Jest to funkcja asynchroniczna, co oznacza, że czas jej wykonania zajmuje więcej czasu niż synchronicznych funkcji. W związku z tym należy wykorzystać mechanizm *async/await* podczas wywołania, informując jednocześnie, w którym momencie interpreter powinien zaczekać z dalszym analizowaniem wywoływanej funkcji, aż wskazana przez słowo kluczowe *await* obietnica *promise* w postaci funkcji *fetch* nie zostanie zakończona.

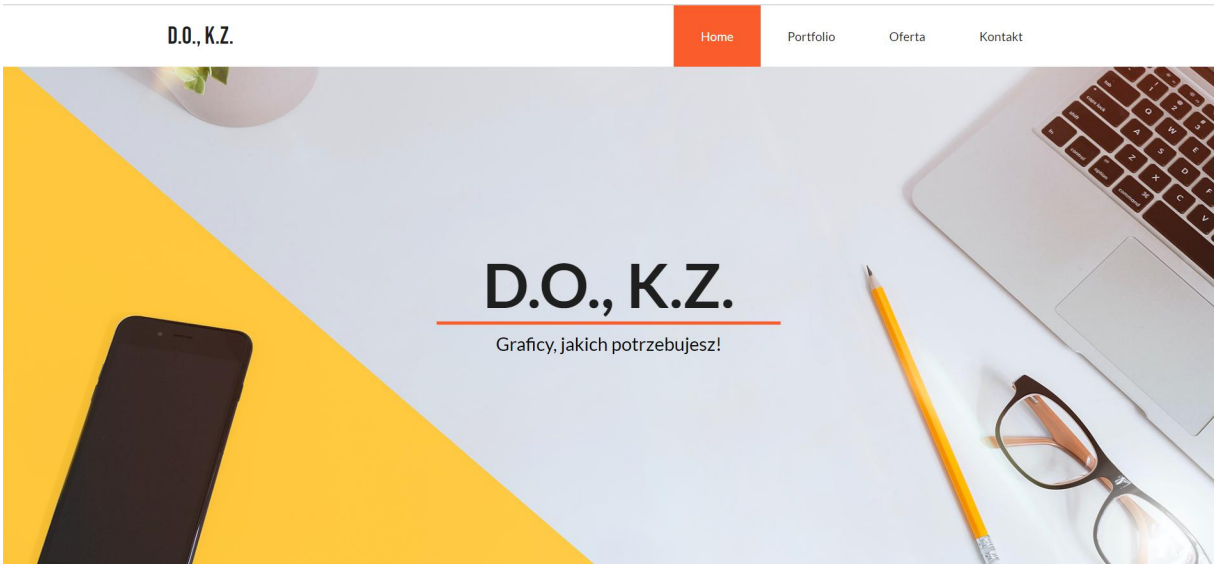
```
exportconstsendEmail=async(bodyData)=> {
   const response =awaitfetch("http://localhost:8800/api/email", {
      method:"POST",
      headers: {
         "Content-Type":"application/json; charset=utf-8",
      },
      body: JSON.stringify(bodyData),
   })

   const resJSON =await response.json()

   return [response.ok, resJSON]
}
```

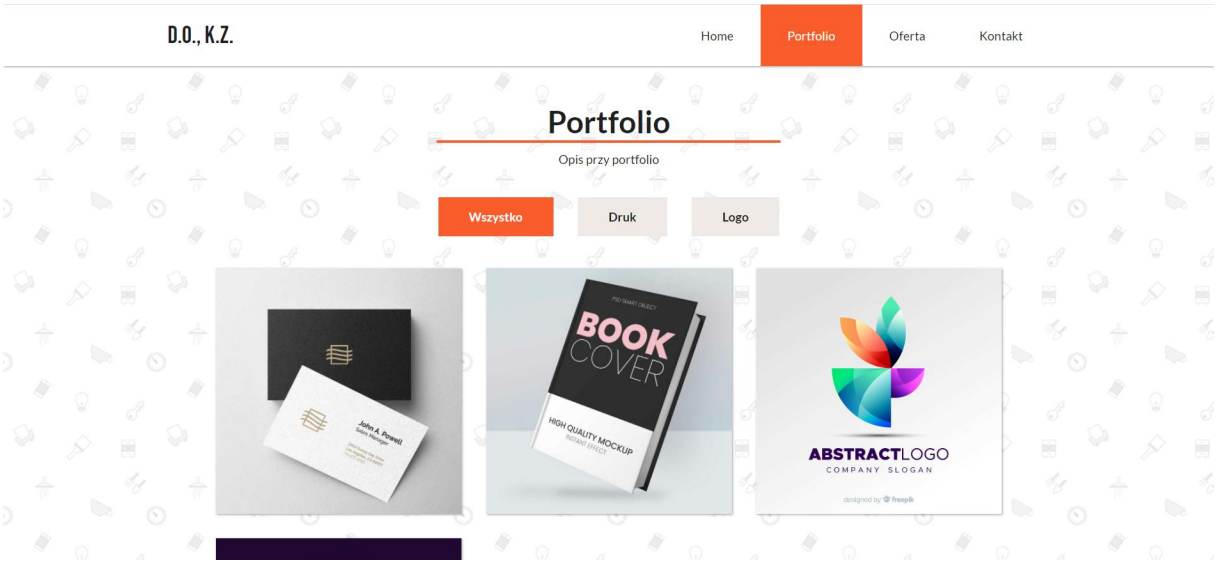
Powyższa funkcja zwraca tablicę dwóch elementów (krotkę) zawierającą wartość *boolean* z informacją o pomyślnym zakończeniu funkcji oraz obiekt przechowujący zwrócone przez odpowiedź żądania dane. Dzięki temu możemy zapewnić mechanizm obsługi zwracanych błędów, a także w przypadku powodzenia wyświetlić informację o sukcesie.

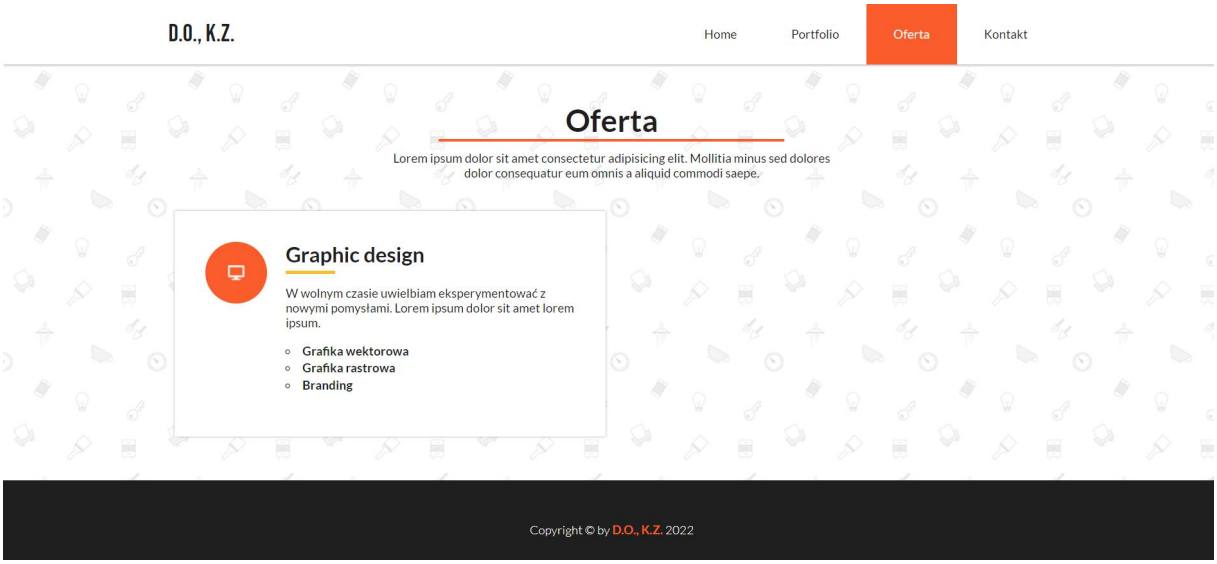
Funkcja ta wywoływana jest w funkcji *handleSubmit*, która podpięta jest pod zdarzenie *onSubmit* formularza wysyłania wiadomości mailowej w komponencie *contact.* Zwrócone przez funkcję *sendEmail* dane są odpowiednio analizowane, a następnie na ich podstawie aktualizowane są zmienne *isSuccess* oraz *errors*, które odpowiadają za wyświetlenie komunikatu o pomyślnym wysłaniu wiadomości oraz potencjalnych informacji o występujących błędach.

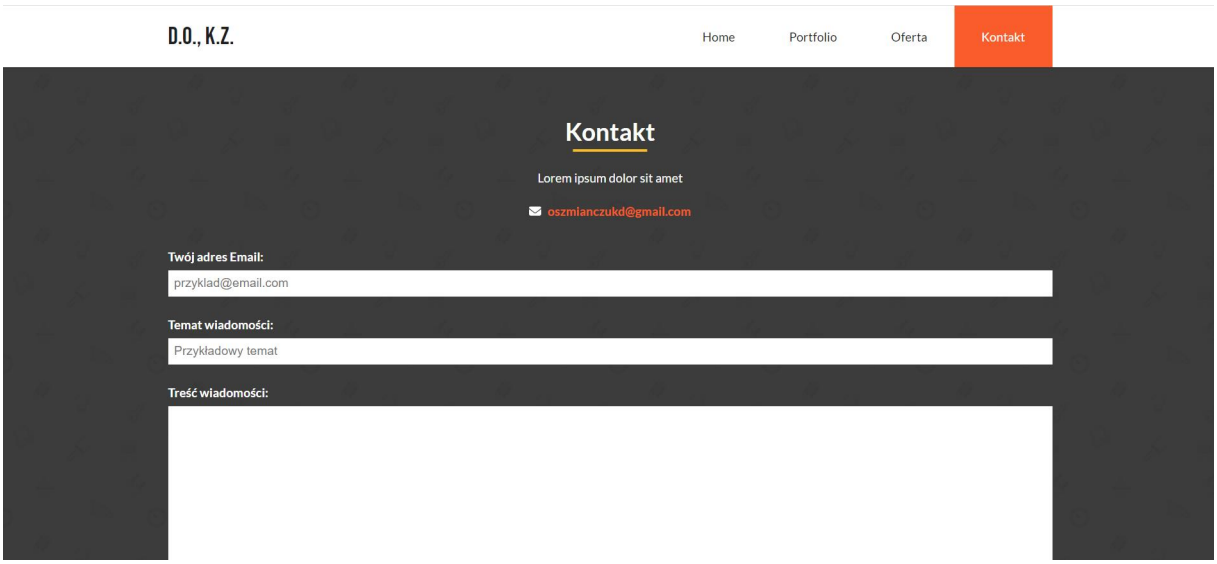
```
consthandleSubmit=async(e)=> {
   e.preventDefault()
   const [isSuccess, errorList] =awaitsendEmail({ from: emailFrom, title, content })

   setErrors([])
   if (!isSuccess) {
      setErrors(errorList)
   }
   setSuccess(isSuccess)
}
```

Strona internetowa: https://lively-duckanoo-738840.netlify.app/






D.O., K.Z.
Home
Portfolio
Oferta
Kontakt
Oferta
Lorem ipsum dolor sit amet consectetur adipisicing elit. Mollitia minus sed dolores dolor consequatur eum omnis a aliquid commodi saepe.
Graphic design
W wolnym czasie uwielbiam eksperymentować z nowymi pomysłami. Lorem ipsum dolor sit amet lorem ipsum.
Grafika wektorowa
Grafika rastrowa
Branding
Copyright © by D.O., K.Z. 2022


D.O., K.Z.
Home
Portfolio
Oferta
Kontakt
Kontakt
Lorem ipsum dolor sit amet
oszmianczukd@gmail.com
Twój adres Email:
przyklad@email.com
Temat wiadomości:
Przykładowy temat
Treść wiadomości:

Widok mobilny: