

**Politechnika Wrocławskaw
Wydział Informatyki i Telekomunikacji**

Kierunek: **Informatyka Techniczna (ITE)**
Specjalność: **Systemy informatyki w medycynie (IMT)**

**PRACA DYPLOMOWA
INŻYNIERSKA**

Strona internetowa stajni

Szymon Hutnik

Opiekun pracy
dr inż. Agata Kirjanów-Błażej

Słowa kluczowe: Aplikacja webowa, React, Express

WROCŁAW 2022

SKRÓTY

REST (ang. *Representational State Transfer*)

API (ang. *Application Programming Interface*)

JSON (ang. *JavaScript Object Notation*)

JWT (ang. *JSON Web Token*)

HTML (ang. *HyperText Markup Language*)

JIT (ang. *Just-in-Time (compilation)*)

UI (ang. *User Interface*)

GUI (ang. *Graphical User Interface*)

URL (ang. *Uniform Resource Locator*)

URI (ang. *Uniform Resource Identifier*)

SPIS TREŚCI

Skróty	1
Wstęp	4
Cel pracy	4
Struktura pracy	4
1. Wykorzystane technologie	5
1.1. JavaScript	5
1.1.1. NodeJs	5
1.1.2. Express	5
1.1.3. React	5
1.2. PostgreSQL	5
1.3. Deta Drive	6
1.4. Railway	6
2. Projekt aplikacji	7
2.1. Wymagania	7
2.1.1. Wymagania funkcjonalne	7
2.1.2. Wymagania niefunkcjonalne	7
2.1.3. Diagram przypadków użycia	8
2.2. Projekt bazy danych	9
2.3. Projekt serwera	10
2.4. Projekt strony internetowej	11
2.5. Wdrożenie	12
3. Implementacja	13
3.1. Baza danych	13
3.2. Serwer	15
3.2.1. Routing	15
3.2.2. REST API	16
3.2.3. Modyfikacja danych	17
3.2.4. Autentykacja i autoryzacja	17
3.3. Strona internetowa	19
3.3.1. Komponenty	19
3.3.2. Komunikacja z serwerem	19
3.3.3. Logowanie	21
3.3.4. Edytory treści	22
3.3.5. Widoki	24
3.4. Wdrożenie projektu	39
3.4.1. Uruchomienie lokalne	39
3.4.2. Uruchomienie na platformie Railway	40
Podsumowanie	41
Bibliografia	42
Spis rysunków	43

Spis listingów	44
-----------------------	----

WSTĘP

Tematem pracy inżynierskiej jest strona internetowa stajni, która może być edytowana przez administratora z poziomu przeglądarki internetowej. W dzisiejszych czasach dostęp do informacji jest bardzo ułatwiony, ponad 90% gospodarstw domowych w Polsce ma dostęp do internetu [1]. Przekłada się to na wagę informacji dostępnych w sieci. Jeżeli potencjalny klient nie może znaleźć informacji o firmie, zazwyczaj wybiera ofertę konkurencji. W internecie można już znaleźć informacje o większości stajni, jednak wciąż rzadkością są ich strony internetowe. To na nich można sprawdzić szczegółowe informacje o obiekcie, zapoznać się z końmi i trenerami, sprawdzić ofertę lub poznać ceny. Dzięki stronie internetowej stajnia ma szansę na przyciągnięcie większej liczby osób jeżdżących konno — klientów. Jako początkujący jeździec miałem trudności ze znalezieniem informacji o stajniach w mojej okolicy, dlatego postanowiłem stworzyć taką aplikację.

CEL PRACY

Celem pracy jest projekt i implementacja nowoczesnej strony internetowej, która zachęci potencjalnych klientów do odwiedzenia stajni. Dane będą przechowywane w bazie danych i będą wysyłane oraz modyfikowane przy użyciu REST API. Strona internetowa ma pozwolić klientom na sprawdzenie informacji o stajni: koniach, trenerach, ofercie, cenach i informacjach kontaktowych, a także na przeglądanie galerii zdjęć. Ma również umożliwić administratorowi, możliwie najprostszy sposób edycji danych.

STRUKTURA PRACY

Praca jest podzielona na: wstęp, 3 rozdziały i podsumowanie. Pierwszy rozdział opisuje wykorzystane technologie. W drugim przedstawiono projekt aplikacji, podano wymagania oraz załączono diagramy mające ułatwić pracę. W trzecim rozdziale można przeczytać, jak zaimplementowano projekt, znajdują się tam fragmenty kodu oraz zrzuty ekranów ze strony internetowej, a także opis jak uruchomić projekt. Na końcu pracy znajduje się podsumowanie.

1. WYKORZYSTANE TECHNOLOGIE

1.1. JAVASCRIPT

JavaScript jest lekkim, interpretowalnym, obiektowym, prototypowym językiem programowania z *first-class functions*. Stosowany do pisania skryptów, które najczęściej wykonują się jako część aplikacji webowej lub jako część serwera [2]. Menadżer paczek *npm* umożliwia dostęp do ogromnej ilości bibliotek i modułów, które można dodawać do swoich projektów [3]. Ilość gotowych rozwiązań znacznie przyspiesza czas tworzenia rozbudowanych aplikacji.

1.1.1. NodeJs

NodeJs jest asynchronicznym, sterowanym eventami środowiskiem Javascript, które pozwala na budowę skalujących się aplikacji webowych. Napędza je silnik V8 napisany w C++, którego używają również przeglądarki oparte na Chromium [4]. Stosuje nieblokującą architekturę oraz może przetwarzanie wiele procesów jednocześnie, co jest bardzo ważne, gdy projektujemy serwer [5].

1.1.2. Express

Express jest szybkim, minimalistycznym webowym frameworkiem w NodeJs przystosowanym do tworzenia aplikacji webowych. Ogromna ilość wbudowanych lub łatwych do dodania modułów umożliwia tworzenie dopracowanych API. Jego minimalizm dostarcza tylko potrzebnych komponentów niezaśmiecających środowiska oraz nie zmniejszając wydajności NodeJs'a. Jego jakość potwierdza fakt, że stanowi bazę, na której zbudowane są inne frameworki [6].

1.1.3. React

React jest biblioteką napisaną przez firmę Meta w języku Javascript, która pozwala na budowanie interaktywnego interfejsu użytkownika. Wyróżnia się użyciem niezależnych komponentów stworzonych przez dewelopera lub importowanych z innych bibliotek, każdy z nich zarządza własnym stanem oraz zwraca elementy napisane w html. React (re)renderuje komponenty, tylko gdy ich stany ulegną zmianie [7]. Duże zmiany nastąpiły w 2018 roku, kiedy wprowadzono *React Hooks*. Nowa funkcjonalność pozwoliła na tworzenie komponentów bez pisania klas, kod stał się całkowicie funkcyjny, bardziej przejrzysty i przyjazny deweloperowi [8].

1.2. POSTGRESQL

PostgreSQL jest potężnym, open source'owym, obiektowym systemem bazodanowym. Jest jedną z najpopularniejszych baz danych, przekłada się to na dobrą dokumentację oraz duże wsparcie społeczności. Postgres wspiera klasyczny język SQL oraz pozwala na definiowanie własnych funkcji i typów danych [9]. Do pracy z bazą można używać dostarczonego przez twórców narzędzia *psql*, które uruchomione w terminalu zapewnia minimalistyczny interfejs. Oprócz wykonywania zapytań można korzystać z dużej ilości meta-komend, które ułatwiają pracę [10].

1.3. DETA DRIVE

Deta jest na zawsze darmową platformą udostępniającą 3 serwisy: Deta Base, Deta Drive i Deta Micros. W tym projekcie wykorzystany zostanie Deta Drive, czyli zarządzany, bezpieczny i skalujący się do 10GB system przechowywania plików. Dysk można obsługiwać poprzez przeglądarkowy interfejs lub za pomocą modułów w Pythonie lub JavaScript [11].

1.4. RAILWAY

Railway jest platformą, na której można tworzyć i wdrażać infrastrukturę. Jest w tym podobny do platformy Heroku, jednak stosuje rozwiązania niewymagające od dewelopera dodatkowej wiedzy, aby móc uruchomić swój projekt. W kilku krokach można stworzyć bazę danych dostępną online, jak również bez dodatkowych konfiguracji uruchomić programy napisane w NodeJs. Dzięki zastosowaniu opłat na postawie zużycia oraz darmowego miesięcznego limitu utrzymanie infrastruktury może być całkowicie darmowe [12].

2. PROJEKT APLIKACJI

2.1. WYMAGANIA

Aplikację można podzielić na części:

- baza danych,
- REST API,
- strona internetowa.

Razem tworzą cały system, dla którego sporządzono zbiór wymagań oraz diagram przypadków użycia przedstawiony na rysunku 2.1.

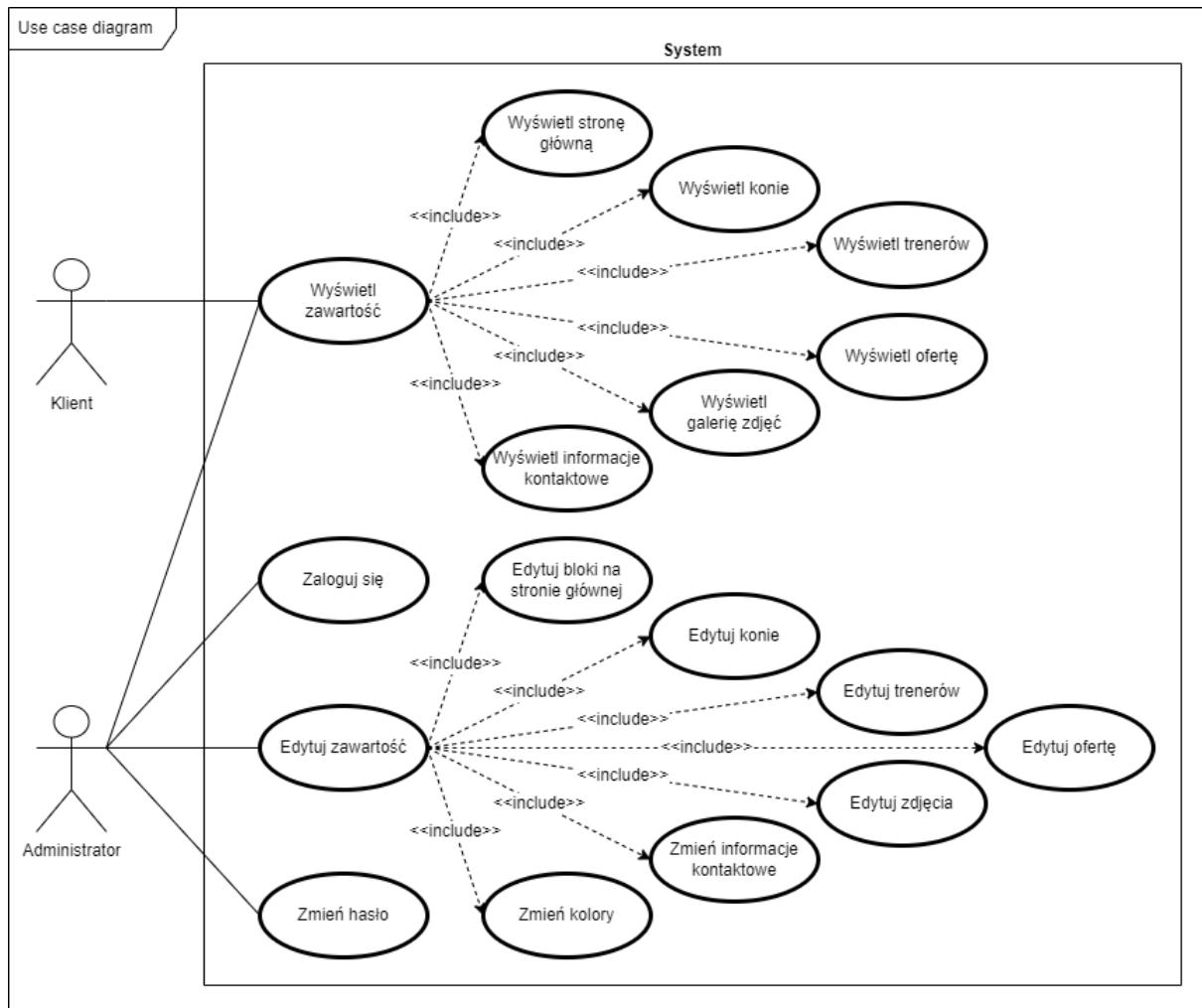
2.1.1. Wymagania funkcjonalne

1. Klient może wyświetlać stronę i jej podstrony.
2. Administrator może się zalogować hasłem.
3. Administrator może dodawać: bloki tekstu na stronie głównej, konie, trenerów, oferty, ceny w cenniku i zdjęcia.
4. Administrator może modyfikować: bloki tekstu na stronie głównej, konie, trenerów, oferty, ceny w cenniku, dane kontaktowe, kolory strony, hasło i które zdjęcia są wyświetlane w galerii.
5. Administrator może usuwać: bloki tekstu na stronie głównej, konie, trenerów, oferty, ceny w cenniku i zdjęcia.
6. Każdy koń i trener mają zdjęcie profilowe.
7. Konie, trenerzy i oferty wyświetlają galerię zdjęć z nimi powiązanych.

2.1.2. Wymagania niefunkcjonalne

1. Każdy będzie mógł odwiedzić stronę internetową.
2. Strona oraz REST API będą działać pod jednym adresem.
3. Hasło będzie hashowane.
4. Baza danych jest backupem, REST API nie powinno się z nią komunikować, jeżeli nie są wprowadzane modyfikacje.
5. Zdjęcia będą przechowywane w chmurze.

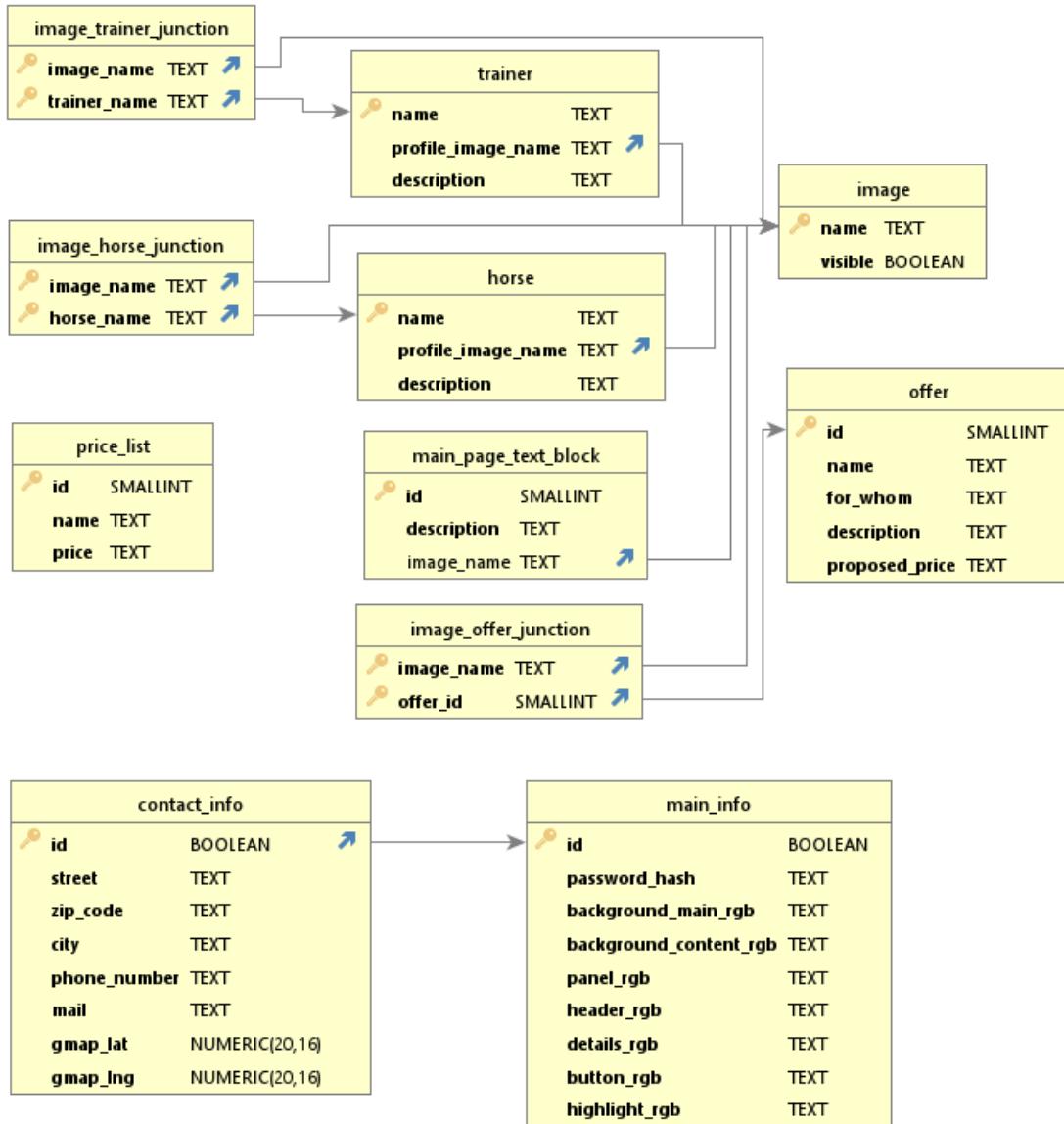
2.1.3. Diagram przypadków użycia



Rys. 2.1. Diagram przypadków użycia aplikacji

2.2. PROJEKT BAZY DANYCH

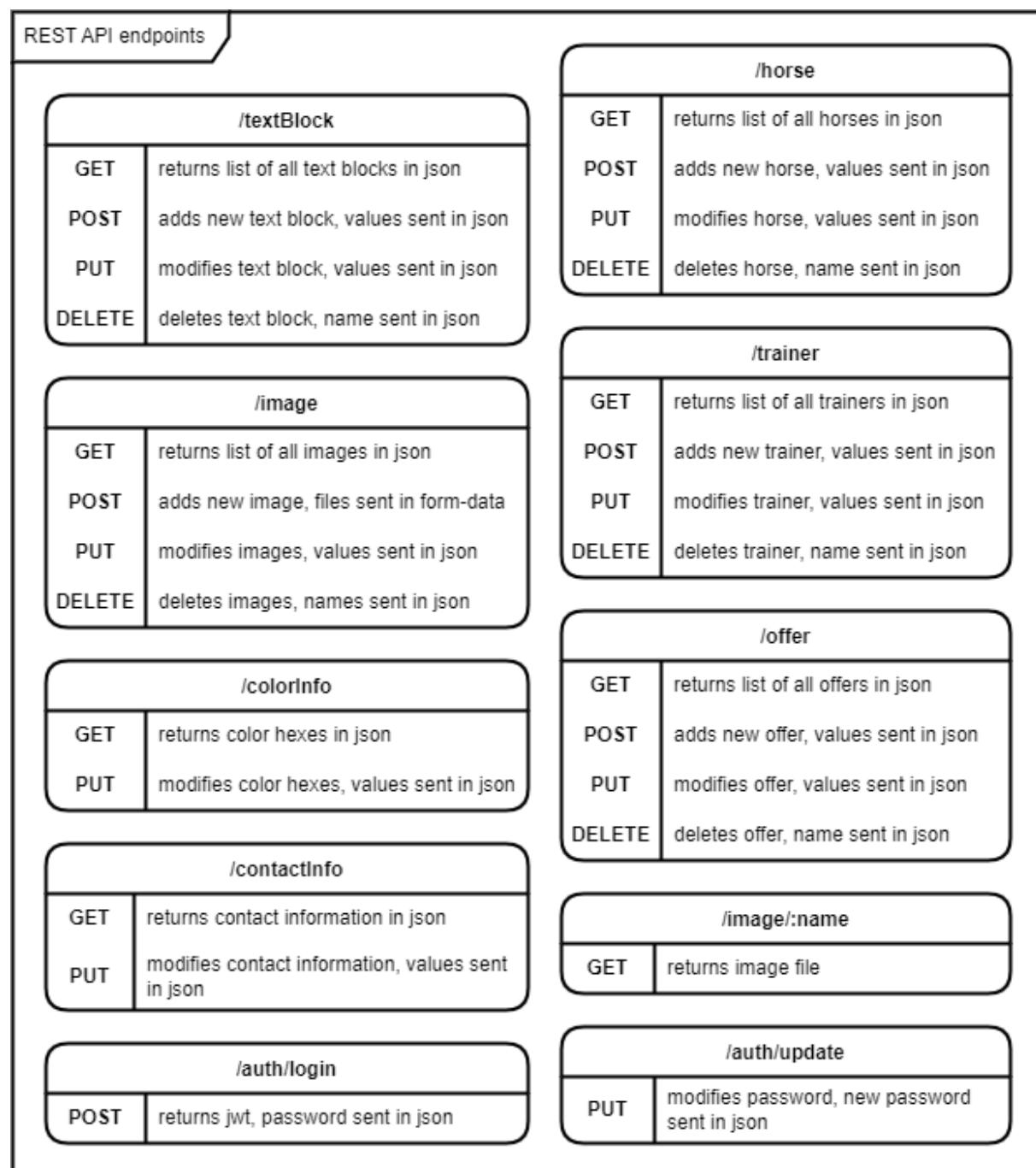
Baza danych będzie używana do przechowywania obiektów wyświetlanego na stronie internetowej. Obrazki będą przechowywane w Deta Drive, a ich dane trzymane w bazie danych. Jako system do zarządzania bazą danych wybrano PostgreSQL. Na rysunku 2.2 przedstawiono model bazy danych w formie diagramu klas. Dodatkowo zostaną przygotowane widoki, które będą umożliwiały łatwe pobranie wartości i zapisanie do formatu JSON.



Rys. 2.2. Diagram klas bazy danych

2.3. PROJEKT SERWERA

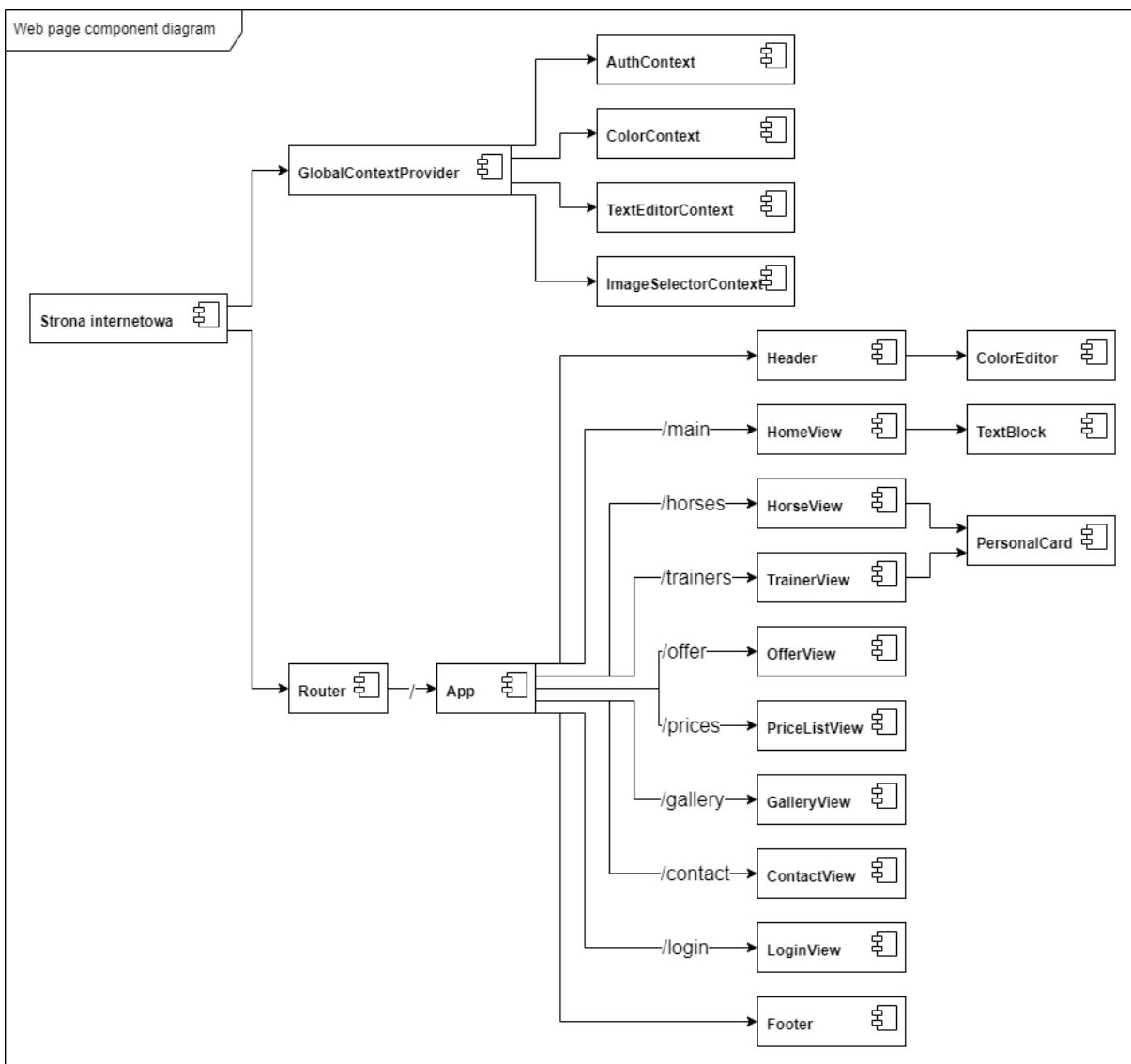
Serwer będzie wykonywać dwie czynności: serwować REST API oraz stronę internetową. REST API będzie wysyłać dane stronie internetowej oraz na żądanie modyfikować bazę danych. W trakcie działania serwer będzie wykorzystywał wartości, które pobrał w trakcie uruchamiania. Interakcje z bazą danych będą się odbywały, tylko gdy administrator zmodyfikuje informacje wyświetlane na stronie. Po zapisaniu zmian serwer zaktualizuje przechowywane wartości, aby nie musieć odpytywać bazy danych przy każdym żądaniu. Do jego stworzenia zostanie wykorzystany NodeJs oraz framework Express. Na rysunku 2.3 przedstawiono mapę endpointów REST API, znajdujących się pod ścieżką /api.



Rys. 2.3. Mapa endpointów REST API

2.4. PROJEKT STRONY INTERNETOWEJ

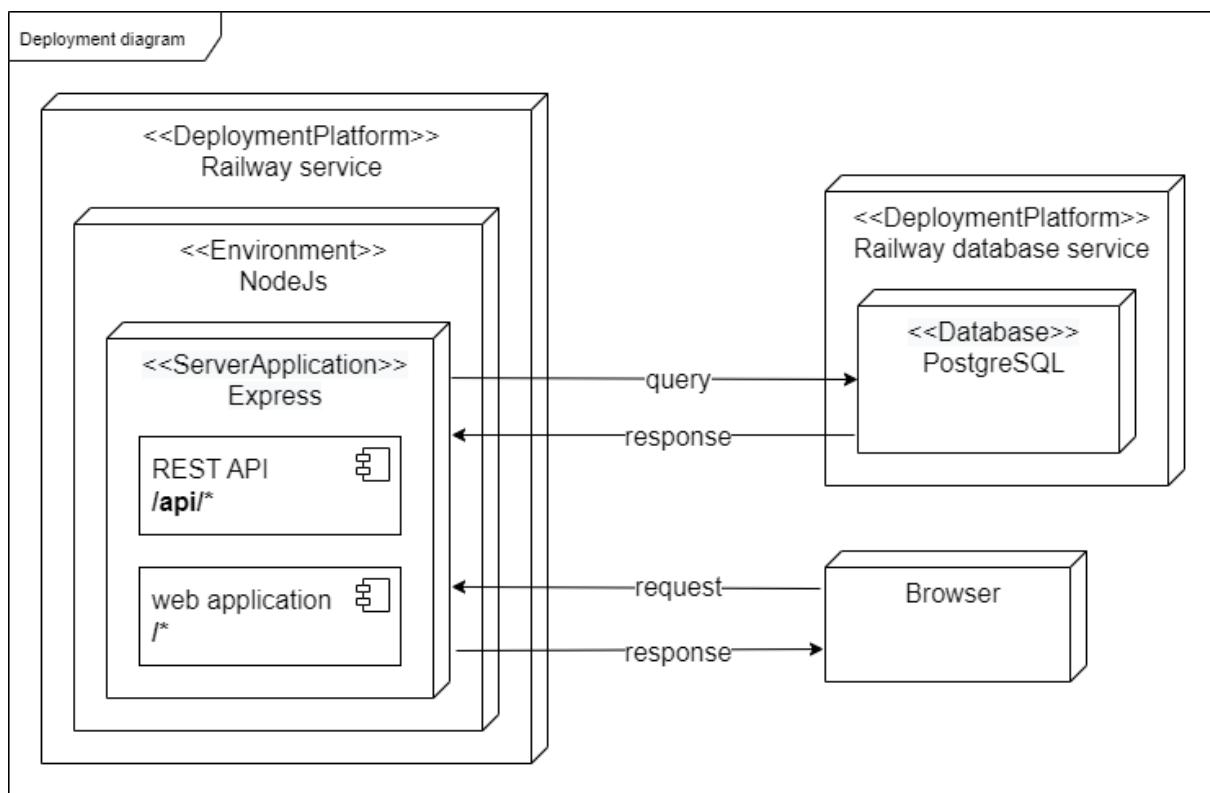
Strona internetowa jest główną częścią projektu; umożliwia ona klientowi przeglądanie informacji o stajni, a administratorowi zarządzanie jej zawartością. Dzięki implementacji interaktywnego interfejsu do zarządzania zasobami administrator strony nie będzie musiał posiadać żadnej wiedzy technicznej, aby wprowadzać zmiany. Strona zostanie napisana we framework'u React. Na rysunku 2.4 znajduje się diagram komponentów, który przedstawia podział strony na logiczne części. Każdy z nich odpowiada za inną funkcjonalność, posiada własną logikę i stany.



Rys. 2.4. Diagram komponentów strony internetowej

2.5. WDROŻENIE

Do wdrożenia aplikacji będzie potrzebna platforma, która pozwoli na hostowanie serwera oraz bazy danych. W tym projekcie zdecydowałem się na platformę Railway, ponieważ jest nowoczesnym, nieskomplikowanym i w pewnym zakresie darmowym rozwiązaniem. Na platformie zostaną stworzone dwa kontenery, w pierwszym będzie postawiona baza danych, a w drugim serwer. REST API znajdzie się pod adresem `/api/*`, a strona internetowa będzie wyświetlana dla pozostałych ścieżek. Plan wdrożenia przedstawia diagram wdrożenia na rysunku 2.5.



Rys. 2.5. Diagram wdrożenia aplikacji

3. IMPLEMENTACJA

3.1. BAZA DANYCH

Baza danych zgodna z zaprojektowanym modelem przedstawionym na rysunku 2.2 została zapisana w formie skryptu w języku SQL. Będzie on służyć do zainicjalizowania bazy, stworzy tabele, widoki i doda po rekordzie do tabel *main_info*, *contact_info* oraz *image*. Tabele *main_info* i *contact_info* są ograniczone do jednego rekordu i zawierają podstawowe informacje o kolorach strony oraz stajni. Pierwszy rekord w tabeli *image* zawiera dane o obrazku, który ma zostać wyświetlony w razie gdy inny obrazek nie zostanie znaleziony, nie można go zmienić. Poniżej znajdują się fragmenty skryptu inicjalizującego: listing 3.1 pokazuje tworzenie tabel, listing 3.2 pokazuje dodanie kluczy obcych do tabel, listing 3.3 pokazuje tworzenie widoków.

```
CREATE TABLE image (
    name          text NOT NULL,
    visible       boolean NOT NULL,
    CONSTRAINT pk_image PRIMARY KEY ( name )
);

CREATE TABLE horse (
    name          text NOT NULL,
    profile_image_name text DEFAULT 'dummyImage.jpg' NOT NULL,
    description    text NOT NULL,
    CONSTRAINT pk_horse PRIMARY KEY ( name )
);

CREATE TABLE image_horse_junction (
    image_name     text NOT NULL,
    horse_name     text NOT NULL,
    CONSTRAINT pk_image_horse_junction PRIMARY KEY ( image_name, horse_name
)
);
```

Listing 3.1: Tworzenie wybranych tabel: *image*, *horse*, *image_horse_junction*, fragment dbSchema.sql

```
ALTER TABLE horse ADD CONSTRAINT fk_horse_image FOREIGN KEY ( profile_image_name
↪ ) REFERENCES image( name ) ON DELETE SET DEFAULT ON UPDATE SET DEFAULT;

ALTER TABLE image_horse_junction ADD CONSTRAINT fk_image_horse_junction_1
↪ FOREIGN KEY ( image_name ) REFERENCES image( name ) ON DELETE CASCADE ON
↪ UPDATE CASCADE;

ALTER TABLE image_horse_junction ADD CONSTRAINT fk_image_horse_junction_2
↪ FOREIGN KEY ( horse_name ) REFERENCES horse( name ) ON DELETE CASCADE ON
↪ UPDATE CASCADE;
```

Listing 3.2: Dodawanie kluczy obcych do wybranych tabelach: *horse*, *image_horse_junction*, fragment dbSchema.sql

```
CREATE VIEW image_list_view AS
SELECT name as image,
       visible
FROM image
WHERE name != 'dummyImage.jpg';

CREATE VIEW horse_list_view AS
SELECT horse.name,
       horse.description,
       array_remove(array_prepend(horse.profile_image_name,
↪ array_agg(image_horse_junction.image_name)), NULL) as images
FROM horse
LEFT JOIN image_horse_junction ON horse.name = image_horse_junction.horse_name
GROUP BY name;
```

Listing 3.3: Tworzenie wybranych widoków: *image_list_view*, *horse_list_view*, fragment dbSchema.sql

3.2. SERWER

Serwer został napisany we frameworku Express w środowisku NodeJs. Oprócz bibliotek pobieranych przez **express-generator** podczas tworzenia podstawowego środowiska w projekcie znalazły się również:

- bcryptjs – biblioteka służąca do generowania i testowania hashy [13], wykorzystana do weryfikacji i aktualizacji hasła,
- deta – klasa służąca do komunikacji z serwisami Deta [14], wykorzystana do zapisu i odczytu obrazków z Deta Drive,
- express-fileupload – paczka dostarczająca middleware umożliwiający pobieranie plików [15],
- jsonwebtoken – paczka dostarczająca implementację JWT, służy do tworzenia i porównywania tokenów [16], wykorzystana w procesie logowania i autentykacji administratora,
- pg – kolekcja modułów umożliwiająca komunikację z bazą danych PostgreSQL [17],
- eslint – narzędzie do statycznej analizy kodu [18], pomaga utrzymać czysty kod i znaleźć potencjalne błędy.

3.2.1. Routing

Poprawnie działający serwer musi obsługiwać wszystkie endpointy zamodelowane na rysunku 2.3, a także serwować stronę internetową i zdjęcia. Aby spełnić dwa ostatnie wymagania zastosowano funkcje **static()** [19], listing 3.4. Pozwala ona na serwowanie plików statycznych znajdujących się w podanym katalogu oraz jego podfolderach. Serwowanie dzieje się automatycznie, jeśli plik na ścieżce podanej w url znajduje się w jednym z katalogów. Jeżeli nie zostanie on znaleziony, można przechwycić takie żądanie, aby przetworzyć je w dowolny sposób, na przykład przekierować je do właściwego miejsca, listing 3.5, lub wysłać odpowiedni plik bezpośrednio, listing 3.6.

```
app.use(express.static(path.join(__dirname, "public")));
app.use(express.static(path.join(__dirname, "/frontend/build")));
```

Listing 3.4: Express static routing, fragment src/app.js

```
router.get("/image/:name", (req, res) => {
  res.redirect(databaseConnector.DUMMY_IMAGE_PATH);
});
```

Listing 3.5: Express redirect, fragment src/routes/api.js

```
router.get("/*", (req, res) => {
  res.sendFile(path.join(__dirname, "/frontend/build/index.html"));
});
```

Listing 3.6: Express sendFile, fragment src/routes/index.js

Dzięki zastosowaniu Routerów można rozdzielić zarządzanie ścieżkami na pliki, każdy router będzie zarządzał jedną z nich, podział zastosowany w projekcie widać na listingu 3.7.

```
const indexRouter = require("./routes/index");
const apiRouter = require("./routes/api");

app.use("/api", apiRouter);
app.use("/", indexRouter);
```

Listing 3.7: Express routing, fragment src/app.js

3.2.2. REST API

REST API będzie służyło do wysyłania danych stajni stronie internetowej. Jest to most pomiędzy bazą danych, a GUI. Składać się będzie z routera zarządzającego routingiem, który serwuje endpointy opisane na rysunku 2.3, oraz modułu odpowiadającego za łączenie się z bazami danych i przechowywanie informacji o stajni. Po otrzymaniu żądania na jeden z endpointów w przypadku metody GET zwracany jest obrazek lub plik json z odpowiednimi wartościami. Jeżeli metoda była inna, weryfikuje się, czy użytkownik jest zalogowany, następnie sprawdza się poprawność przesłanych danych. Dopiero wtedy wywoływana jest odpowiednia funkcja z modułu komunikującego się z bazami danych. Przykładowo wysłanie żądania z metodą PUT na adres **/api/horse** wywoła funkcję pokazaną na listingu 3.8.

```
router.put("/horse", (req, res) => {
  const updatedHorse = req.body;
  if (
    !updatedHorse ||
    !updatedHorse.name ||
    !updatedHorse.description ||
    !updatedHorse.images ||
    updatedHorse.images.length === 0
  )
    return res.status(406).json("Mandatory fields not set");

  if ((err = checkName(updatedHorse.name, databaseConnector.getHorseList(),
    "Horse"))) return res.status(406).json(err);
  if ((err = checkImages(updatedHorse.images))) return
    res.status(406).json(err);

  databaseConnector.updateHorse(updatedHorse).then((err) => {
    return err ? res.status(500).json(err) : res.sendStatus(200);
  });
});
```

Listing 3.8: Funkcja obsługująca żądanie z metodą PUT wysłane na adres **/api/horse**, fragment src/routes/api.js

3.2.3. Modyfikacja danych

Połączenie z bazą danych jest zrealizowane za pomocą instancji klasy `Pool()` z kolekcji pg. Operacje na bazie danych są wykonywane poprzez wywołanie na niej funkcji `query()`, której podaje się zapytanie oraz listę zmienionych wartości [17]. Po wprowadzeniu zmian wywoływana jest funkcja pobierająca stan z bazy danych na serwer. Przykład funkcji dodającej rekord do tabeli `horse` w bazie danych można zobaczyć na listingu 3.9. Pobieranie i usuwanie zdjęć w Deta Drive odbywa się za pomocą modułu Deta, który łączy się z projektem za pomocą klucza projektu, znajdziemy go w panelu projektu na stronie <https://web.deta.sh/>. Następnie subklasa reprezentującą połączenie z wybranym dyskiem jest inicjalizowana [14], proces łączenia widać na listingu 3.10. Na zainicjalizowanej subklasie można wywoływać funkcje pozwalające na modyfikację plików zapisanych na dysku.

```
const createHorse = async (newHorse) => {
  try {
    await pool.query("INSERT INTO horse VALUES ($1, $2, $3)", [
      newHorse.name,
      newHorse.image,
      newHorse.description
    ]);
    return updateFromDatabase();
  } catch (err) {
    console.error(err);
    return INTERNAL_SERVER_ERROR_MESSAGE;
  }
};
```

Listing 3.9: Dodanie rekordu do tabeli `horse` w Javascript, fragment src/databaseConnector.js

```
const { Deta } = require("deta");
const detaInstance = Deta(process.env.DETA_KEY);
const detaImageDrive = detaInstance.Drive("images");
```

Listing 3.10: Inicjalizacja połączenia z Deta Drive `images`, fragment src/databaseConnector.js

3.2.4. Autentykacja i autoryzacja

Administrator strony musi mieć możliwość zalogowania się i modyfikacji treści, opcje te są zablokowane dla zwykłego użytkownika. Wymaga to zaimplementowania modułu autentykującego oraz autoryzującego użytkownika. Logowanie polega na wysłaniu na adres `/api/auth/login` żądania POST z hasłem, jeśli jest ono zgodne z hasłem przechowywanym w bazie danych serwer tworzy nowy jwt i wysyła go w odpowiedzi [20]; token jest ważny przez godzinę. Następnie, gdy użytkownik chce zmodyfikować dane, musi przejść weryfikację. W tym celu żądanie najpierw przechodzi przez autoryzator, jeżeli token jest poprawny, żądanie jest przekazywane do następnej funkcji. Autentykację przedstawiono na listingu 3.11, a autoryzację na listingu 3.12.

```
router.post("/login", async (req, res) => {
  try {
    if (!req.body || !req.body.password) return res.status(406).json("Password
      ↵ not sent");
    const password = req.body.password;

    const passwordIsValid = bcrypt.compareSync(password, await
      ↵ databaseConnector.getPassword());
    if (!passwordIsValid) return res.status(401).json("Invalid Password!");

    const token = jwt.sign({}, process.env.PRIVATE_KEY, {
      expiresIn: EXPIRATION_TIME,
    });
    return res.status(200).json({
      accessToken: token,
      expiresIn: EXPIRATION_TIME,
    });
  } catch (err) {
    console.log(err);
    return res.status(500).json("Internal server error, contact maintainer");
  }
});
```

Listing 3.11: Autentykacja, fragment src/routes/authenticator.js

```
const verifyToken = (req, res, next) => {
  const token = req.headers["x-access-token"];

  if (!token) {
    return res.status(403).json("No token provided!");
  }

  jwt.verify(token, process.env.PRIVATE_KEY, (err) => {
    if (err) {
      return res.status(401).json("Unauthorized!");
    }
    next();
  });
};
```

Listing 3.12: Autoryzacja, fragment src/middleware/authorizator.js

3.3. STRONA INTERNETOWA

Strona internetowa została napisana we framework'u React. Oprócz bibliotek pobieranych przez **create-react-app** podczas tworzenia podstawowego środowiska w projekcie znalazły się również: primeflex, primeicons, primereact – ta kolekcja pozwala na użycie gotowych komponentów z biblioteki PrimeReact [21].

3.3.1. Komponenty

Aplikacja została podzielona na zbiory komponentów, każdy z nich pełni inne funkcje:

- komponenty bez zbioru – w projekcie są dwa komponenty, które nie należą do żadnego zbioru: *src/index.js*, *src/App.js*. Stanowią one podstawę aplikacji, nie wykonują konkretnych funkcji, a jedynie renderują szkielet strony, który jest uzupełniany innymi komponentami.
- *views* (widoki) – każdy z nich jest innym widokiem, zakładką. Może być kompletnym elementem lub używać komponentów ze zbiorów layouts oraz components.
- *layouts* (układy) – są to fragmenty układu strony, które się powtarzają między widokami jak nagłówek.
- *contextProviders* – są to specjalne komponenty, które są dodawane i renderowane przed resztą elementów. Zapewniają kontekst, który umożliwia wykorzystanie wartości i funkcji z tych komponentów. W tym projekcie do tego zbioru będą należeć komponenty obsługujące logowanie, kolory, edycję tekstu i obrazków.
- *components* (komponenty) – do tego zbioru trafiają komponenty, które nie mają specjalnego zastosowania.

3.3.2. Komunikacja z serwerem

Aby strona mogła działać prawidłowo, musi pobierać dane z serwera. Do tego celu wykorzystano prostą funkcję **fetch()**, domyślną metodą jest GET. Po otrzymaniu odpowiedzi jest ona przetwarzana i jeśli status jest "ok" zmieniony zostaje stan komponentu, przykład użycia na listingu 3.13.

```
const fetchHorseList = () => {
  fetch(API_URL + "horse")
    .then((response) => checkResponseOk(response))
    .then((response) => setHorseList(response))
    .catch((err) => {
      console.error(`Server response: ${err}`);
    });
};
```

Listing 3.13: Przykład funkcji **fetch()**, fragment *src/views/HorseView/HorseView.js*

Aktualizowanie danych przebiega podobnie, jednak użytkownik musi być wcześniej zalogowany, aby uzyskać uprawnienia do edycji. Dlatego też funkcja aktualizująca została umieszczona w *contextProvider* odpowiedzialnym za logowanie. Przyjmuje w parametrach: adres url, metodę, obiekt, który ma zostać wysłany, callback, który ma zostać wywołany po udanej aktualizacji. Do żądania zostaje dołączony token, który umożliwia autoryzację. Funkcja wyświetla również komunikaty informujące użytkownika o jej wyniku, listing 3.14.

```
const performDataUpdate = (url, method, body, callback) => {
  const authToken = localStorage.getItem("authToken");
  if (!authToken) {
    toast.current.show({ severity: "error", summary: "Token lost", detail: "Nie
      ↵ znaleziono tokena, wylogowano!", life: 10000, });
    logoutUser();
    return;
  }
  fetch( API_URL + url, url === "image" && method === "POST"
    ? { method: "POST", headers: { "x-access-token": JSON.parse(authToken) },
      ↵ body: body, }
    : { method: method, headers: { "x-access-token": JSON.parse(authToken),
      ↵ "Content-Type": "application/json" }, body: JSON.stringify(body), }
  )
  .then((response) => checkResponseOk(response))
  .then(() => {
    callback();
    toast.current.show({ severity: "success", summary: "Sukces", detail:
      ↵ "Zmiany zostały zapisane", life: 2000 });
  })
  .catch((err) => {
    console.error(`Server response: ${err}`);
    if (err === "No token provided!") {
      toast.current.show({ severity: "error", summary: "Błąd", detail:
        ↵ `${err}`, token zgubiony, zostałeś wylogowany!`, life: 6000, });
      logoutUser();
      return;
    }
    if (err === "Unauthorized!") {
      toast.current.show({ severity: "warn", summary: "Brak uprawnień",
        ↵ detail: "Nie masz uprawnień, aby modyfikować ten zasób!", life:
        ↵ 6000, });
      return;
    }
    toast.current.show({ severity: "error", summary: "Błąd", detail: `Błąd
      ↵ serwera: ${err}`, zmiany nie zostały zapisane`, life: 6000, });
  });
};
```

Listing 3.14: Funkcja wysyłająca żądanie aktualizacji danych, fragment
src/contextProviders/AuthContextProvider/AuthContextProvider.js

3.3.3. Logowanie

W celu zalogowania administrator musi wpisać hasło, następnie jest ono wysyłane do REST API w celu autentykacji. Jeżeli jest ono poprawne, w odpowiedzi otrzymuje token służący do potwierdzania tożsamości, gdy wprowadzane są zmiany danych. Token i czas jego ważności jest zapisywany w **localStorage**, które pozwala przechowywać wartości, nawet gdy strona jest odświeżana, dzięki temu nie trzeba się niepotrzebnie logować. Administrator może się wylogować, kiedy zechce lub zostanie wylogowany automatycznie, gdy czas ważności tokena upłynie. Na listingu 3.15 znajduje się funkcja logująca, a na listingu 3.16 wylogowująca.

```
const loginUser = (password) => {
  fetch(API_URL + "auth/login", { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify({ password: password }) })
    .then((response) => checkResponseOk(response))
    .then((response) => {
      localStorage.setItem("authToken", JSON.stringify(response.accessToken));
      localStorage.setItem("expirationTime", Date.now() + response.expiresIn - BUFFER_EXPIRATION_TIME);

      setAuthContext((prevState) => {
        prevState.isLoggedIn = true;
        return { ...prevState };
      });
      navigate("/");
    })
    .catch((err) => {
      console.error(`Server response: ${err}`);
      toast.current.show({ severity: "error", summary: "Błąd", detail: "Próba logowania nie powiodła się", life: 10000, });
    });
};
```

Listing 3.15: Funkcja logująca, fragment
src/contextProviders/AuthContextProvider/AuthContextProvider.js

```

() => {
  console.warn("logout");
  localStorage.removeItem("authToken");
  localStorage.removeItem("expirationTime");

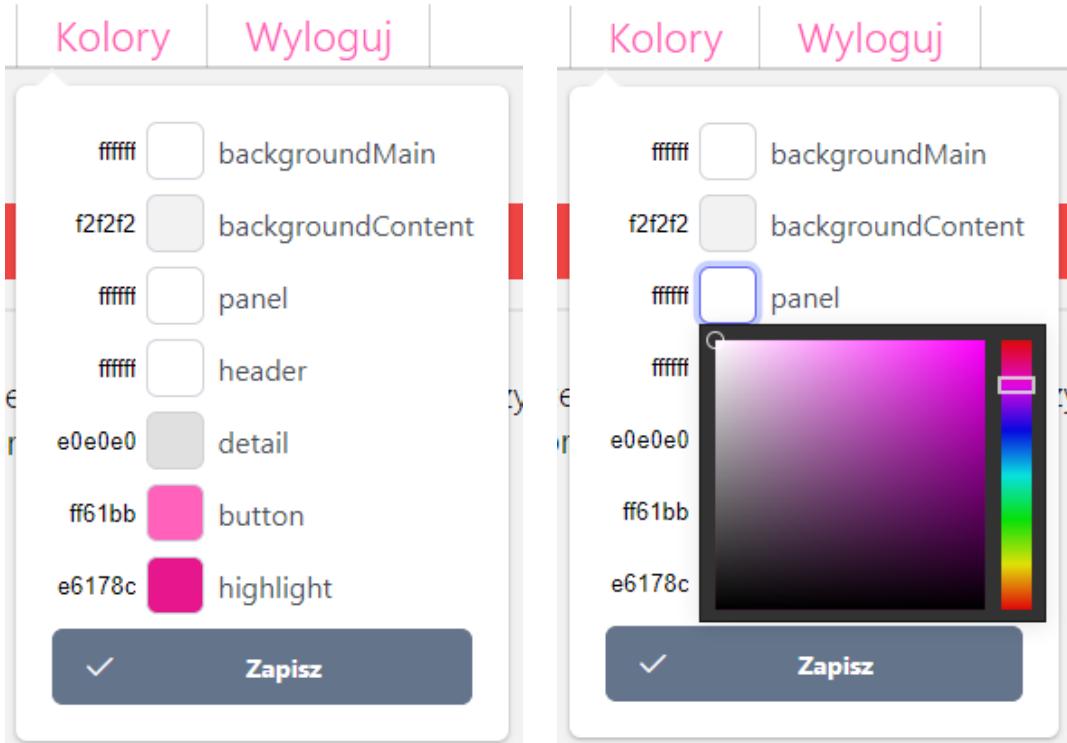
  setAuthContext((prevState) => {
    prevState.isLoggedIn = false;
    return { ...prevState };
  });
  navigate("/");
}

```

Listing 3.16: Funkcja wylogowująca, fragment
src/contextProviders/AuthContextProvider/AuthContextProvider.js

3.3.4. Edytory treści

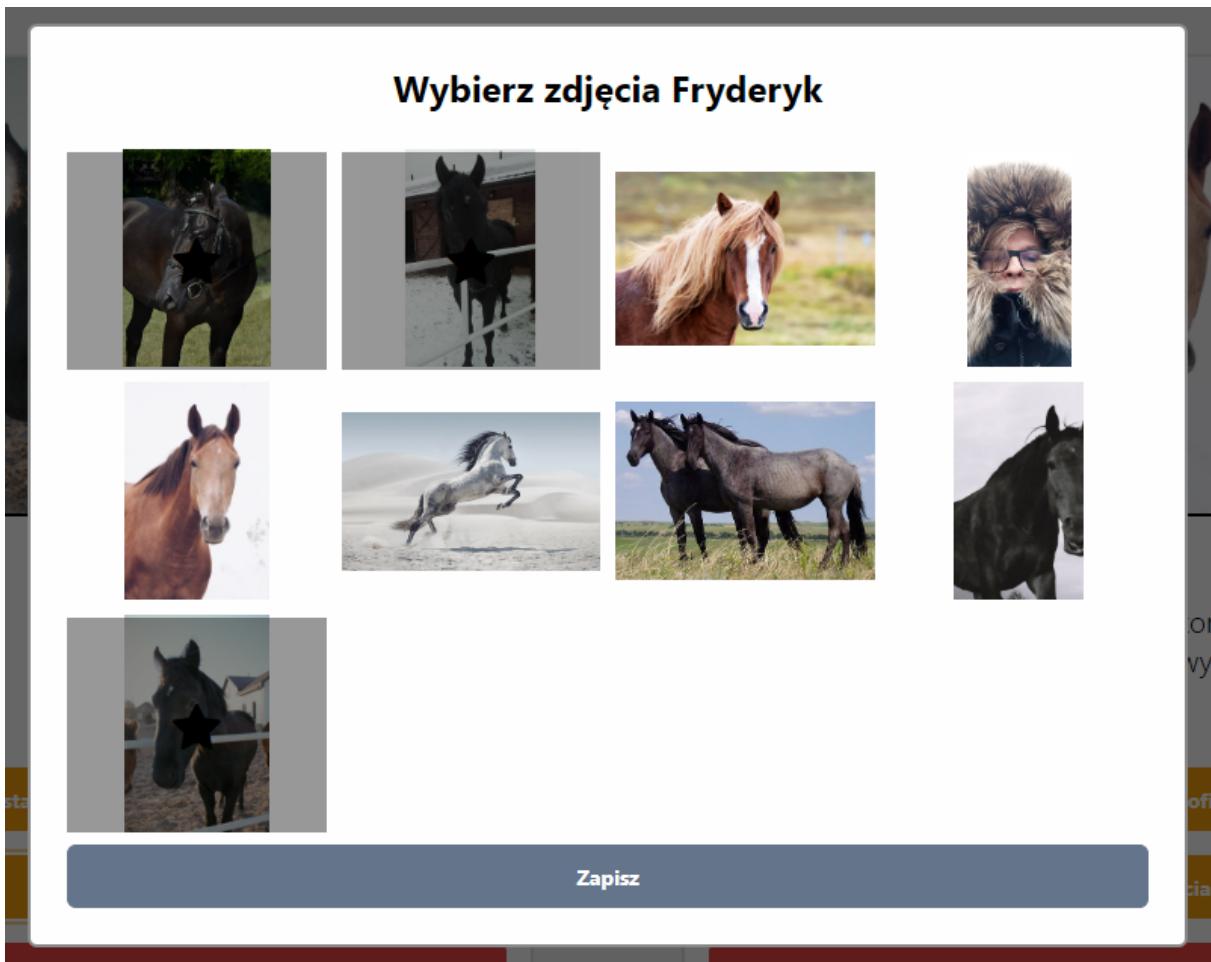
Administrator ma możliwość edycji treści wyświetlanej na stronie. Dokonuje tego za pomocą trzech edytorów: koloru, zdjęć i tekstu. Pierwszy z nich jest panelem, który pojawia się po kliknięciu przycisku w nagłówku, wyświetla listę kolorów, które mogą zostać zmienione, rysunek 3.1. Przy każdym z nich jest wyświetlony kod hex oraz podgląd nowego koloru. Edycji można dokonać poprzez ręczne wpisanie nowej wartości hex lub panel wybierania koloru, który otwiera się po kliknięciu kwadratowej ikonki z podglądem, rysunek 3.2. Zapisanie zmian następuje dopiero po kliknięciu przycisku "Zapisz", zamknięcie panelu powoduje utratę wprowadzonych zmian.



Rys. 3.1. Edytor kolorów

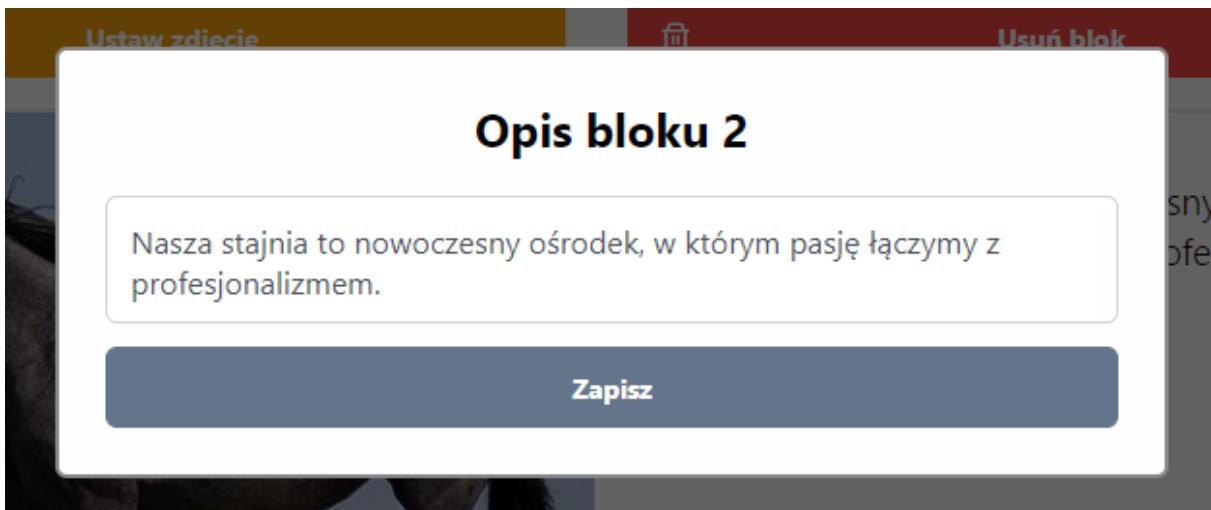
Rys. 3.2. Edytor kolorów, panel wyboru koloru

Komponent pozwalający na wybór obrazków również jest modelem. Pozwalać na wybranie kilku lub jednego zdjęcia, można podać maksymalnie jeden obrazek, który nie może zostać odznaczony. Zaznaczenia dokonuje się poprzez kliknięcie obrazka, zaznaczone zdjęcia mają szary filtr z gwiazdką. Wybrane obrazki należy zatwierdzić klikając przycisk "Zapisz", wywoła to funkcję, która zapisze wprowadzone zmiany, naciśnięcie szarego obszaru powoduje odrzucenie zmian.



Rys. 3.3. Wybieranie obrazków

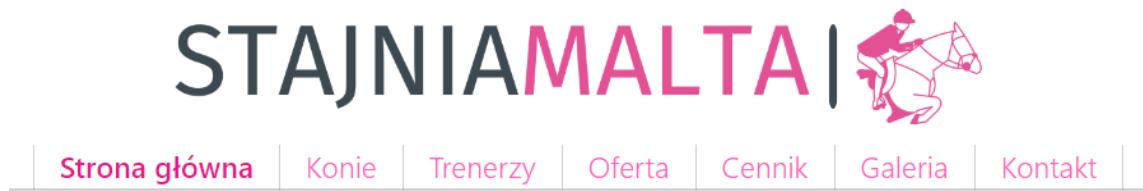
Edytor tekstu jest modelem, który zostaje wywołany, gdy administrator modyfikuje lub wprowadza nowy tekst, rysunek 3.4. Nową wartość należy zatwierdzić kliknięciem przycisku, wywoła to funkcję, która zapisze wprowadzone zmiany, naciśnięcie szarego obszaru powoduje odrzucenie zmian.



Rys. 3.4. Edytor tekstu

3.3.5. Widoki

Strona internetowa będzie podzielona na 8 widoków, które będą różnić się w zależności, czy użytkownikiem jest klient, czy zalogowanym administratorem. Widoki klienta będą pokazywać stronę w formacie "read-only". Administratorowi będą wyświetlane dodatkowe przyciski służące do wprowadzania zmian w treści za pomocą edytorów omówionych w sekcji 3.3.4. Podstawowy układ strony składa się z trzech części: nagłówka, treści oraz stopki. Nagłówek zawiera logo oraz przyciski służące do nawigacji po stronie; po zalogowaniu powiększa się o dwa przyciski: do zmiany kolorów i do wylogowania. Nagłówek przed zalogowaniem pokazano na rysunku 3.5, a po zalogowaniu na rysunku 3.6. Stopka zawiera informację o prawach autorskich. Strona jest dostosowana urządzeń mobilnych, jednak zaleca się korzystanie z ekranów o standardowych rozdzielczościach w szczególności po zalogowaniu.



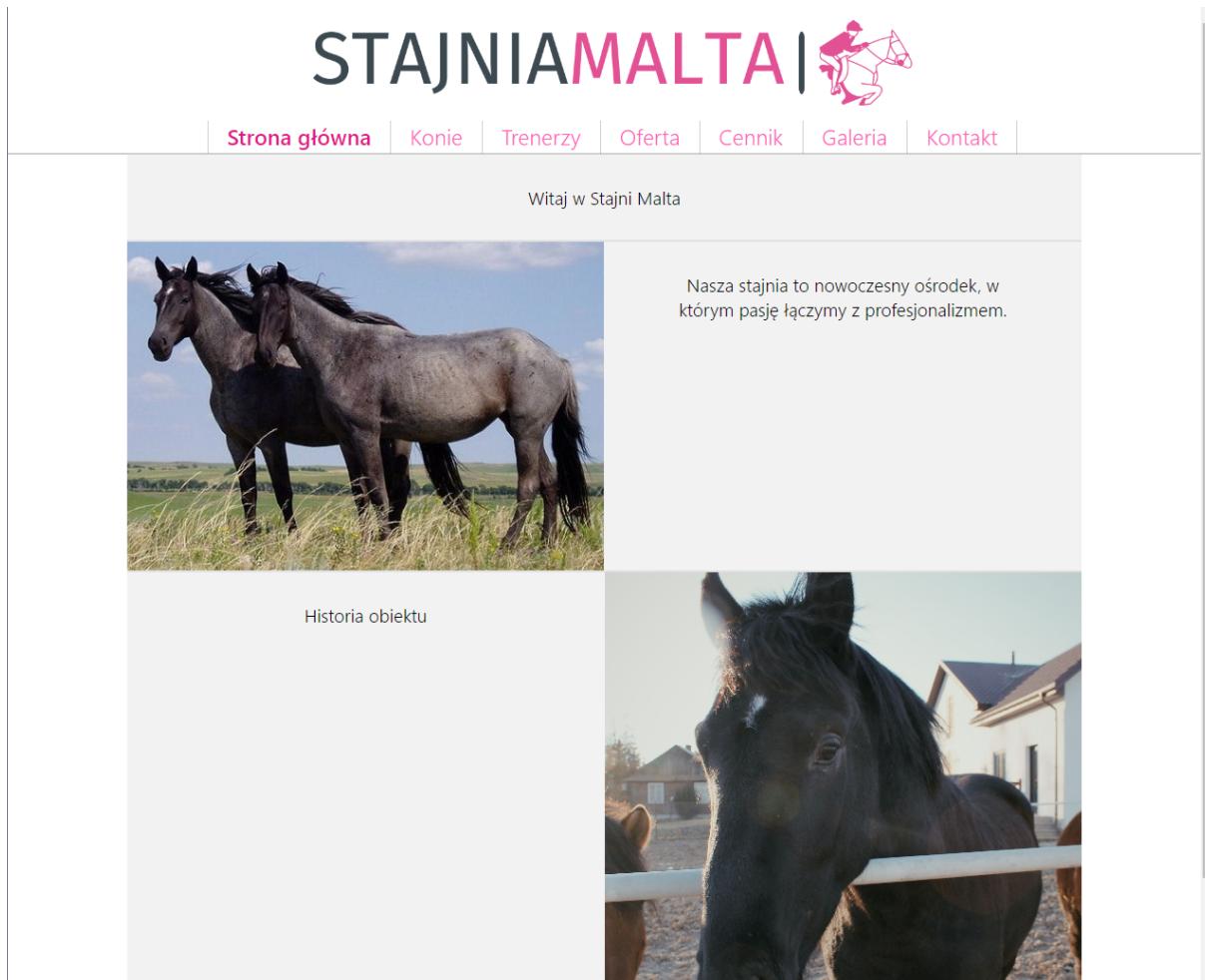
Rys. 3.5. Nagłówek klienta



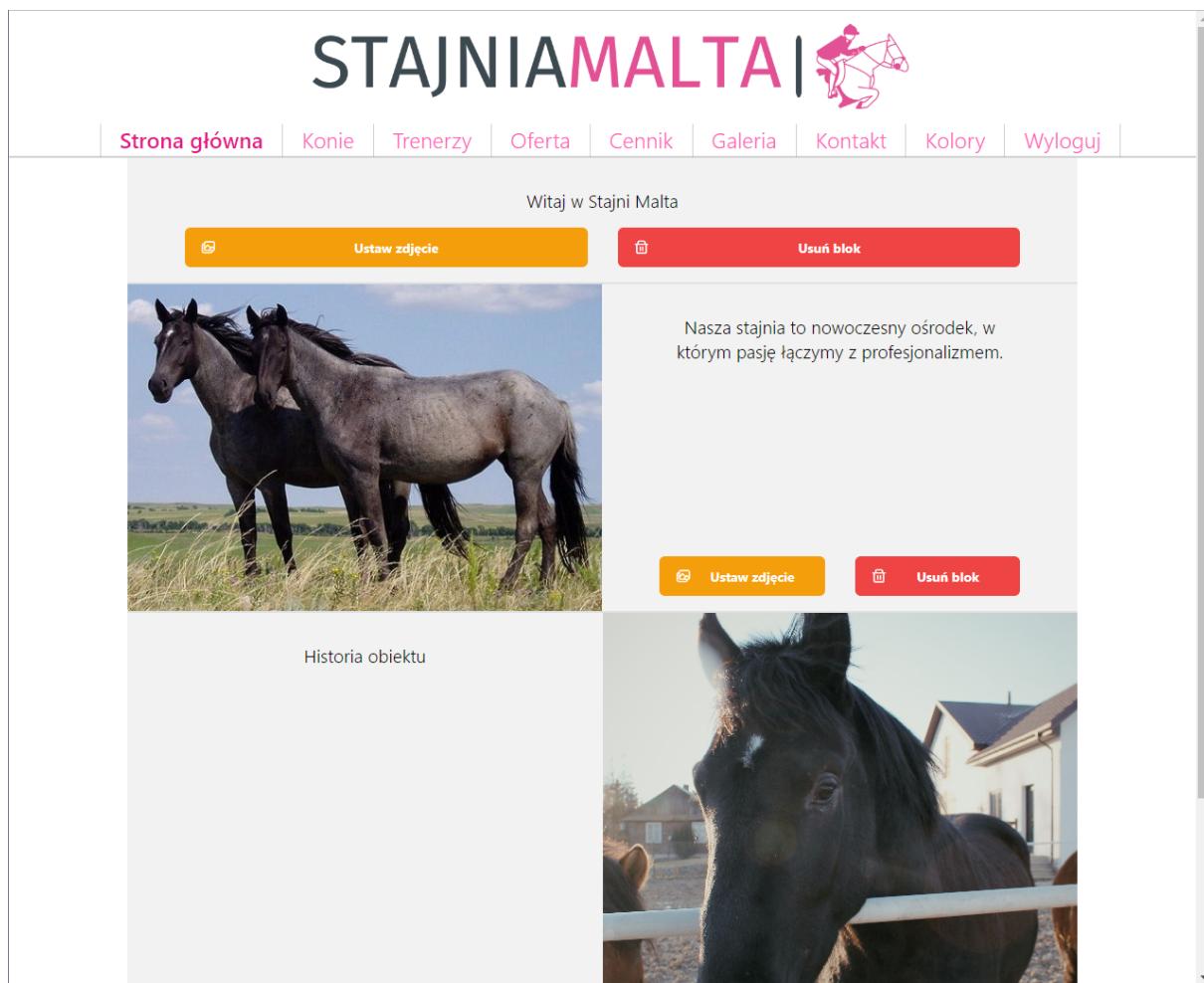
Rys. 3.6. Nagłówek administratora

Widok główny

Widok główny (domowy) jest zlokalizowany pod adresem /main; wszystkie adresy, które nie prowadzą do innych widoków, będą do niego przekierowywanie. Składa się z bloków z tekstem lub z tekstem i obrazem. Jego celem jest autoprezentacja i przekazanie informacji o stajni, opowiadzenie historii lub kilku ciekawostek o obiekcie. Administrator może kliknąć tekst, aby otworzyć edytor tekstu. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.7, oraz z perspektywy administratora, rysunek 3.8.



Rys. 3.7. Przykładowy wygląd widoku głównego z perspektywy klienta



Rys. 3.8. Przykładowy wygląd widoku głównego z perspektywy administratora

Widok koni

Widok koni jest zlokalizowany pod adresem **/horses**. Składa się z paneli z galerią obrazków oraz imieniem i opisem konia. Tutaj znajdują się informacje o koniach w stajni, a także galeria zdjęć związanych z każdym z nich. Administrator może edytować opis, zdjęcie profilowe oraz zdjęcia powiązane z koniem. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.9, oraz z perspektywy administratora, rysunek 3.10.

The screenshot shows a website for 'STAJNIA MALTA'. The header features the logo 'STAJNIA MALTA' with a silhouette of a person riding a horse, followed by a menu bar with links: Strona główna, Konie, Trenerzy, Oferta, Cennik, Galeria, and Kontakt. Below the menu, there are two cards for horses:

- Fryderyk**: A dark brown horse standing in a fenced enclosure. Below the image is a link labeled 'Opis'.
- Malta**: A light brown horse. Below the image is a detailed description: 'Malta jest 10 letnim doświadczonym koniem. Jako oczko w głowie właścicielki jest bardzo zadbane oraz wytrenowana. Doskonaly koń dla każdego jeźdza'.

At the bottom right of the page, there is a small copyright notice: '© 2022 Stajnia Malta. All rights reserved'.

Rys. 3.9. Przykładowy wygląd widoku koni z perspektywy klienta

STAJNIA MALTA | 

Strona główna Konie Trenerzy Oferta Cennik Galeria Kontakt Kolory Wyloguj



Fryderyk

Opis

 Ustaw zdjęcie profilowe

 Ustaw zdjęcia

 Usuń



Malta

Malta jest 10 letnim doświadczonym koniem. Jako oczko w głowie właścicielki jest bardzo zadbane oraz wytrenowana. Doskonały koń dla każdego jeźdźca

 Ustaw zdjęcie profilowe

 Ustaw zdjęcia

 Usuń

Dodaj nowego konia

© 2022 Stajnia Malta. All rights reserved

Rys. 3.10. Przykładowy wygląd widoku koni z perspektywy administratora

Widok trenerów

Widok trenerów jest zlokalizowany pod adresem /trainers. Jest on analogiczny do widoku koni, ponieważ stosuje ten sam komponent do wyświetlania informacji. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.11, oraz z perspektywy administratora, rysunek 3.12.

The screenshot shows a web page with a header containing the logo 'STAJNIA MALTA' and a small horse icon. Below the header is a navigation bar with links: Strona główna, Konie, Trenerzy, Oferta, Cennik, Galeria, and Kontakt. The main content area features a large image of a person wearing a fur-trimmed hood and glasses. Below the image is a box containing the name 'Szymon' and a short bio: 'Szymon jest naszym najbardziej doświadczonym trenerem. Jego umiejętności potwierdzają liczne nagrody zdobyte w zawodach.' At the bottom right of the page, there is a small copyright notice: '© 2022 Stajnia Malta. All rights reserved.'

Rys. 3.11. Przykładowy wygląd widoku trenerów z perspektywy klienta

STAJNIA MALTA |

Strona główna | Konie | Trenerzy | Oferta | Cennik | Galeria | Kontakt | Kolory | Wyloguj



Szymon

Szymon jest naszym najbardziej doświadczonym trenerem. Jego umiejętności potwierdzają liczne nagrody zdobyte w zawodach.

[Dodaj nowego trenera](#)



[Ustaw zdjęcie profilowe](#)



[Ustaw zdjęcia](#)



[Usuń](#)

© 2022 Stajnia Malta. All rights reserved

Rys. 3.12. Przykładowy wygląd widoku trenerów z perspektywy administratora

Widok ofert

Widok ofert jest zlokalizowany pod adresem **/offer**. Jego celem jest zachęcenie potencjalnego klienta do skorzystania z jednej z usług oferowanych przez stajnię. Widok jest złożony z harmoniki, można rozwinać maksymalnie jedną ofertę na raz. Każda z nich ma swoją nazwę, opis, informację do kogo jest skierowana oraz proponowaną cenę; jeśli powiązano z nią obrazki to zostaną one wyświetlane pod tekstem jako galeria. Administrator może edytować każdą z tych wartości. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.13, oraz z perspektywy administratora, rysunek 3.14.

The screenshot shows a website for 'STAJNIA MALTA'. The header features the logo 'STAJNIA MALTA' with a small horse and jockey icon, and a navigation menu with links: Strona główna, Konie, Trenerzy, Oferta, Cennik, Galeria, and Kontakt. Below the header, a section titled 'Jazda Indywidualna' is expanded, showing a description: 'Doskonaly sposób na szlifowanie swoich umiejętności jeździeckich pod okiem doświadczonych trenerów.' (A perfect way to refine your equestrian skills under the guidance of experienced trainers.) It also lists 'Dla kogo: dla każdego' (For whom: for everyone) and 'Proponowana cena: 100' (Proposed price: 100). A central image shows a dark horse from the side, looking towards the camera. At the bottom of the expanded section, there is a link 'Jazda terenowa'.

Rys. 3.13. Przykładowy wygląd widoku ofert z perspektywy klienta

Strona główna Konie Trenerzy Oferta Cennik Galeria Kontakt Kolory Wyloguj

▼ Jazda indywidualna

Doskonały sposób na szlifowanie swoich umiejętności jeździeckich pod okiem doświadczonych trenerów.

Dla kogo: dla każdego
Proponowana cena: 100



Zmień nazwę Ustaw zdjęcie Usuń

➤ Jazda terenowa

Dodaj nową ofertę

Rys. 3.14. Przykładowy wygląd widoku ofert z perspektywy administratora

Widok cennika

Widok cennika jest zlokalizowany pod adresem **/prices**. Wyświetla tabelę z nazwą oferty i ceną. Z założenia ma być uproszczonym widokiem ofert, tutaj klient uzyskuje konkretną informację dotyczącą ceny. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.15, oraz z perspektywy administratora, rysunek 3.16.

The screenshot shows a website header with the logo 'STAJNIA MALTA' and a horse icon. Below the header is a navigation menu with links: Strona główna, Konie, Trenerzy, Oferta, Cennik, Galeria, Kontakt. The main content area displays a table with one row:

Usługa	Cena
Jazda indywidualna - godzina	100

Rys. 3.15. Przykładowy wygląd widoku cennika z perspektywy klienta

The screenshot shows a website header with the logo 'STAJNIA MALTA' and a horse icon. Below the header is a navigation menu with links: Strona główna, Konie, Trenerzy, Oferta, Cennik, Galeria, Kontakt, Kolory, Wyloguj. The main content area displays a table with one row and a 'Usuń' (Delete) button, followed by a blue button labeled 'Dodaj nowy przedmiot' (Add new item). The table structure is identical to the one in Rys. 3.15.

Usługa	Cena	Usuń
Jazda indywidualna - godzina	100	

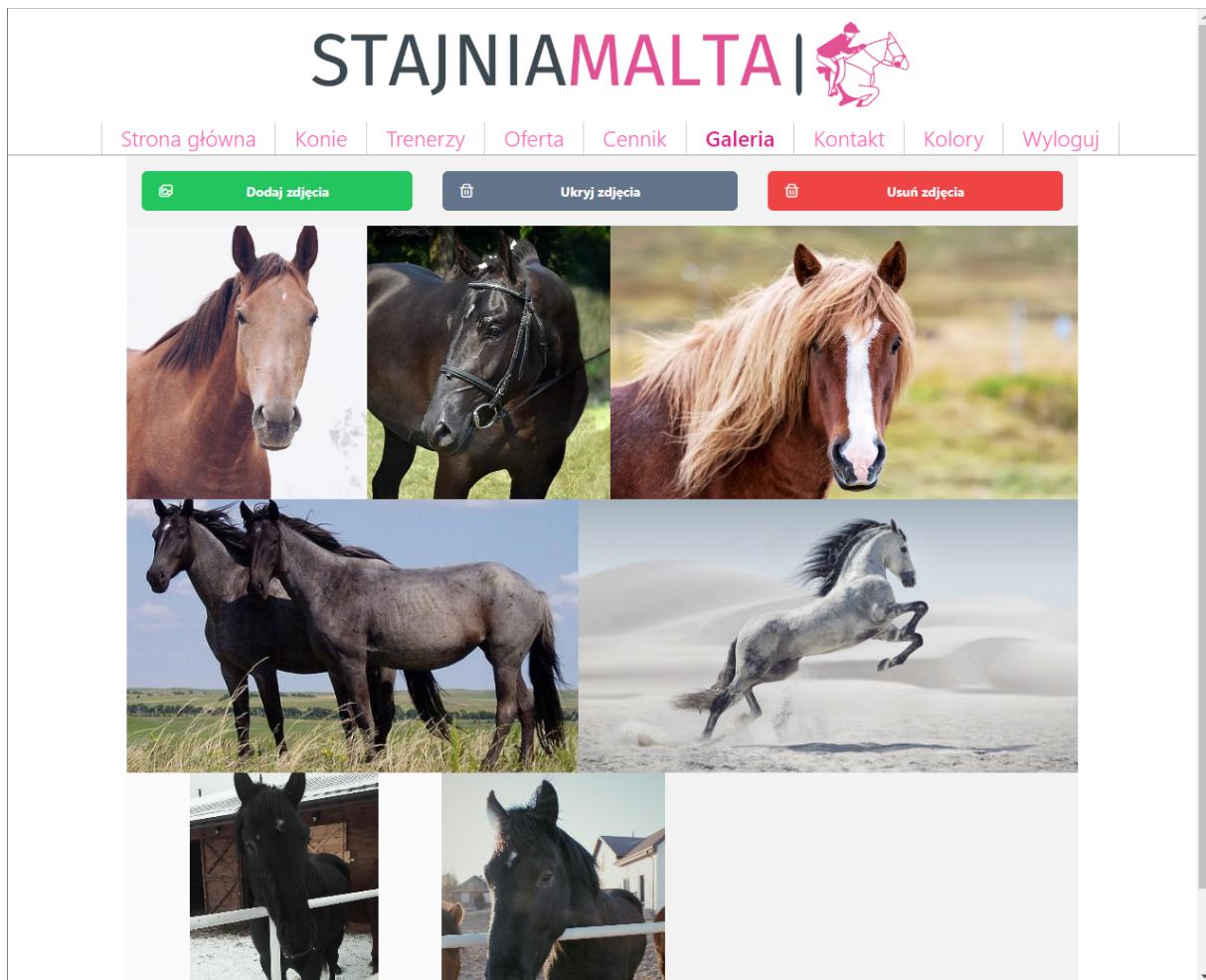
Rys. 3.16. Przykładowy wygląd widoku cennika z perspektywy administratora

Widok galerii

Widok galerii jest zlokalizowany pod adresem **/gallery**. Wyświetla galerię zdjęć, które są wgrane na stronę. W tym widoku administrator może dodawać i usuwać zdjęcia z bazy oraz zaznaczać, które z nich mają nie być wyświetlane w galerii. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.17, oraz z perspektywy administratora, rysunek 3.18.



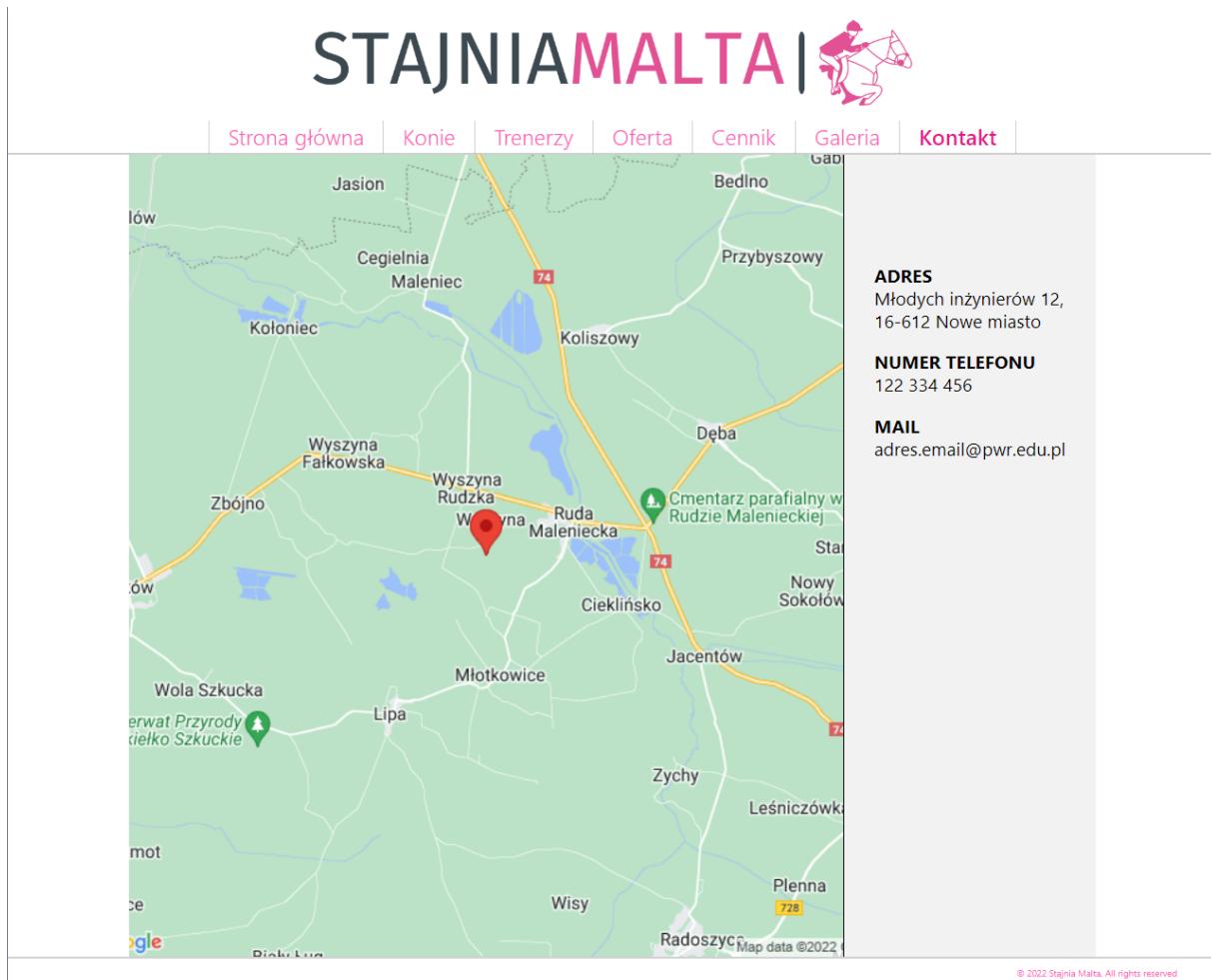
Rys. 3.17. Przykładowy wygląd widoku galerii z perspektywy klienta



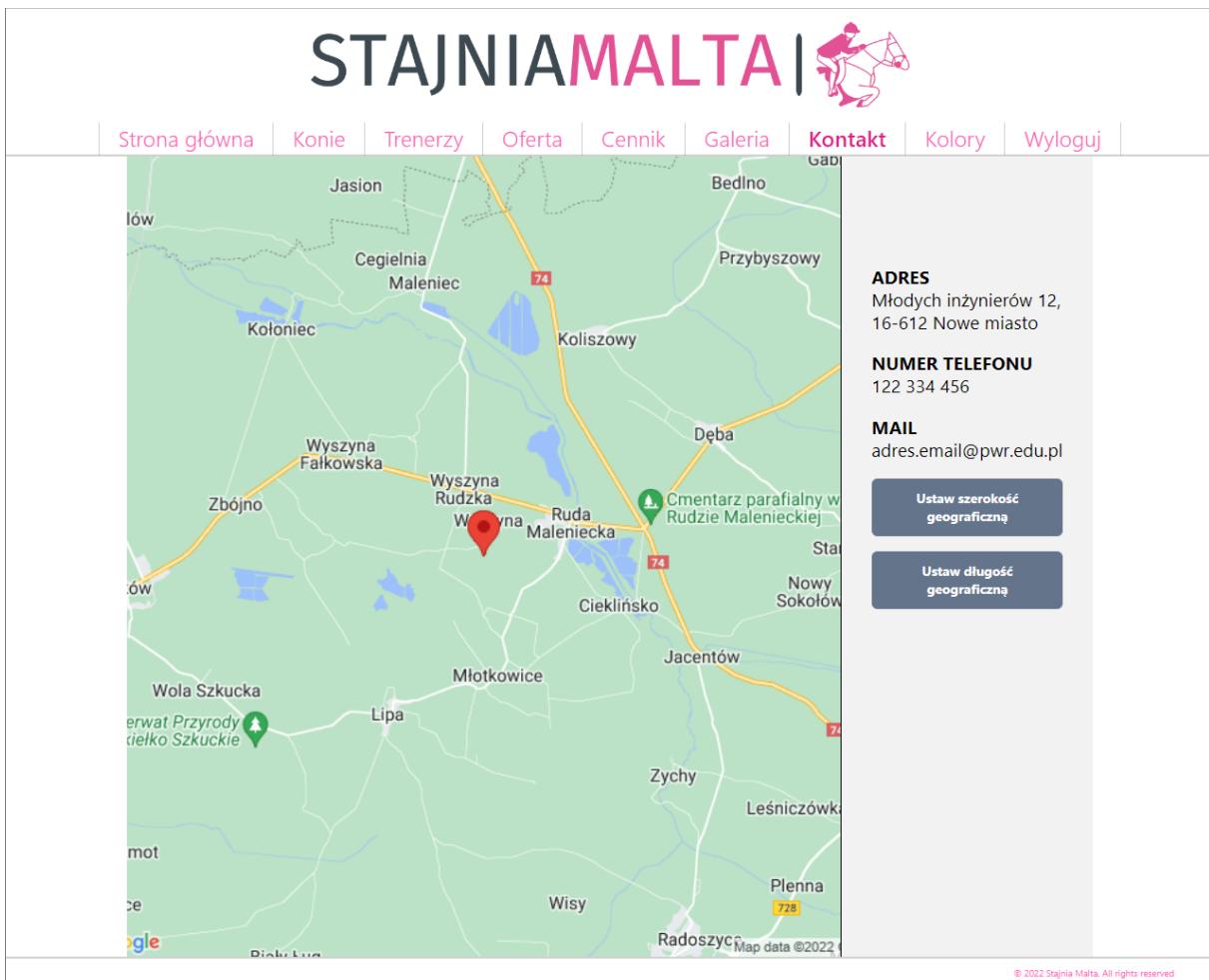
Rys. 3.18. Przykładowy wygląd widoku galerii z perspektywy administratora

Widok kontaktowy

Widok kontaktowy jest zlokalizowany pod adresem [/contact](#). To w nim można znaleźć mapę, wykorzystującą Google Maps API [22], oraz informacje kontaktowe stajni. Administrator może edytować te informacje i zmieniać pozycję mapy podając nowe wartości długości i szerokości geograficznych. Przykładowy wygląd widoku z perspektywy klienta, rysunek 3.19, oraz z perspektywy administratora, rysunek 3.20.



Rys. 3.19. Przykładowy wygląd widoku kontaktowego z perspektywy klienta



Rys. 3.20. Przykładowy wygląd widoku kontaktowego z perspektywy administratora

Widok logowania i zmiany hasła

Widok logowania i zmiany hasła jest zlokalizowany pod adresem **/login**. Jest to jedyny widok, do którego nie można przejść przyciskiem, należy wpisać adres bezpośrednio w przeglądarkę. Wynika to z faktu, że ta funkcjonalność jest niedostępna dla zwykłego użytkownika. Strona logowania składa się z elementu do wpisania hasła i przycisku, rysunek 3.21. Strona zmiany hasła jest podobna, można do niej przejść tylko poprzez ręczne wpisanie adresu w przeglądarkę po uprzednim zalogowaniu. Również składa się z elementu do wpisania hasła i przycisku, rysunek 3.22. Na obu stronach, zamiast przycisku można posłużyć się klawiszem ENTER.

The screenshot shows the login page for the website 'STAJNIA MALTA'. At the top, there is a navigation bar with links: 'Strona główna', 'Konie', 'Trenerzy', 'Oferta', 'Cennik', 'Galeria', and 'Kontakt'. Below the navigation bar is a logo consisting of the text 'STAJNIA MALTA' in a large, bold, black font, followed by a small icon of a person riding a horse. The main form area contains a text input field labeled 'Podaj hasło' (Enter password) with a placeholder icon of an eye. Below the input field is a green button labeled 'Zaloguj' (Log in). The background of the form area is light gray.

Rys. 3.21. Widok logowania

The screenshot shows the password change page for the website 'STAJNIA MALTA'. At the top, there is a navigation bar with links: 'Strona główna', 'Konie', 'Trenerzy', 'Oferta', 'Cennik', 'Galeria', 'Kontakt', 'Kolory', and 'Wyloguj'. Below the navigation bar is a logo consisting of the text 'STAJNIA MALTA' in a large, bold, black font, followed by a small icon of a person riding a horse. The main form area contains a text input field labeled 'Nowe hasło' (New password) with a placeholder icon of an eye. Below the input field is a dark blue button labeled 'Zmień hasło' (Change password). The background of the form area is light gray.

Rys. 3.22. Widok zmiany hasła

3.4. WDROŻENIE PROJEKTU

Projekt można uruchomić lokalnie lub z użyciem infrastruktury opisanej na diagramie wdrożenia, rysunek 2.5. W obu przypadkach należy sprawdzić, czy dostępne będą poniższe programy:

- NodeJs, zalecana wersja 18.3.0 lub nowsza,
- npm, zalecana wersja 18.3.0 lub nowsza,
- PostgreSQL zalecana wersja 13.

Oprócz powyższych wymagań należy założyć konto na platformie Deta oraz Google Cloud Platform i uruchomić serwis umożliwiający korzystanie z Google Maps Static API [22]. Przed uruchomieniem projektu należy zgromadzić następujące dane:

- wartości służące do połączenia z bazą danych: PGDATABASE, PGHOST, PGPASSWORD, PGPORT, PGUSER,
- klucz prywatny służący do podpisywania jwt – PRIVATE_KEY,
- klucz do projektu na platformie Deta – DETA_KEY,
- klucz do Google Cloud Platform – REACT_APP_GMAP_API_KEY,
- adres REST API – REACT_APP_REST_API_URL.

Wartości, które będą wykorzystywane we frameworku React muszą mieć przedrostek REACT_APP. Można również podać PORT, na którym ma zostać uruchomiony serwer, domyślnie jest to 3001.

3.4.1. Uruchomienie lokalne

Przed przystąpieniem do uruchomienia serwera należy wgrać skrypt z pliku dbSchema.sql do bazy danych.

Zalecanym systemem operacyjnym jest Linux. W celu uruchomienia lokalnie należy przejść do katalogu, w którym znajduje się projekt i wprowadzić do lokalnego środowiska wartości wymienione w poprzednim punkcie, można to zrobić komendą **export**, przykład takiej operacji na listingu 3.17.

```
export PGDATABASE=database
export PGHOST=host
export PGPASSWORD=password
export PGPORT=port
export PGUSER=postgres
export PRIVATE_KEY=private_key
export DETA_KEY=deta_key

export REACT_APP_GMAP_API_KEY=gmap_key
export REACT_APP_REST_API_URL="http://localhost:3001/api/"
```

Listing 3.17: Przykład użycia komendy export

Następnie należy wpisać standardowe komendy, listing 3.18. Uruchomią one skrypty, które zainstalują wymagane paczki, zbudują stronę internetową i uruchomią serwer.

```
npm install  
npm run build  
npm start
```

Listing 3.18: Komendy uruchamiające serwer

Stronę internetową można uruchomić niezależnie od serwera w celach dewelopowania.

3.4.2. Uruchomienie na platformie Railway

Platforma Railway jest docelowym miejscem, z którego aplikacja ma funkcjonować. Pierwszym krokiem powinno być sklonowanie projektu do repozytorium na platformie GitHub [23]. Następnie należy założyć konto na platformie Railway, może być ono całkowicie darmowe, stworzyć nowy projekt i w nim dodać bazę danych PostgreSQL. Udostępniony zostanie przejrzysty interfejs, którego możemy użyć do wgrania skryptu z pliku dbSchema.sql, jednak zaleca się skorzystanie z programu psql [10]. W zakładce "Variables" widoczne są wartości służące do komunikacji z bazą danych, nie należy ich kopiować, jeśli aplikacja serwera będzie uruchomiona w ramach tego samego projektu [12].

Kolejnym krokiem jest dodanie kontenera "GitHub Repo" i połączenie go z wcześniej utworzonym repozytorium. W zakładce "Variables" należy dodać: klucz prywatny, klucz do projektu na platformie Deta, klucz do Google Cloud Platform i adres REST API. Adres strony ustawiany jest w zakładce "Settings", można podać własną domenę lub podać część nazwy, która zostanie zakończona ".up.railway.app". W zakładce "Deployments" widoczny jest aktualnie działająca wersja aplikacji, domyślnie każda zmiana dodana do projektu na GitHubie powoduje przebudowanie aplikacji do najnowszej wersji. Dzięki skryptom znajdującym się w projekcie instalacja, budowanie i uruchomienie są automatyczne.

PODSUMOWANIE

Celem pracy było zaprojektowanie i zaimplementowanie strony internetowej stajni. Założenia projektu, opisane w rozdziale 2, zostały spełnione. Zaimplementowano bazę danych i serwer serwujący REST API oraz stronę internetową. Klient może odwiedzać stronę internetową, a administrator ma możliwość edycji informacji na niej wyświetlanych. Aplikacja będzie stanowić doskonały dodatek do każdej stajni, zapewni dostęp do informacji potencjalnym klientom i umożliwi właścielowi edycję wyświetlonej treści.

Mimo że aplikacja jest kompletna i nie wymaga dodatkowej pracy, aby spełniać zadane jej wymagania, można wprowadzić w niej kilka zmian. Jednak przed edycją projektu należałoby napisać do niego testy, pozwoli to wychwycić błędy, które mogą powstać podczas wprowadzania nowych funkcjonalności. Następnie w widoku kontaktu, można użyć interaktywnej mapy zamiast stałego obrazu. Na życzenie właściciela stajni można również wprowadzać zmiany w wyglądzie strony.

BIBLIOGRAFIA

- [1] Urząd Statystyczny w Szczecinie, *Społeczeństwo informacyjne w Polsce w 2022 r.*, <https://stat.gov.pl/obszary-tematyczne/nauka-i-technika-spoleczenstwo-informacyjne/spoleczenstwo-informacyjne/spoleczenstwo-informacyjne-w-polsce-w-2022-roku,2,12.html>.
- [2] Mozilla Foundation, *About javascript*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. Stan na dzień 20.11.2022.
- [3] npm, Inc., *npm*, <https://docs.npmjs.com/cli/v8/commands/npm?v=true>. Stan na dzień 20.11.2022.
- [4] OpenJS Foundation, *What is v8?*, <https://nodejs.org/en/about/>. Stan na dzień 22.11.2022.
- [5] OpenJS Foundation, *About node.js*, <https://nodejs.org/en/about/>. Stan na dzień 22.11.2022.
- [6] OpenJS Foundation, *Express*, <https://expressjs.com/>. Stan na dzień 22.11.2022.
- [7] Meta Platforms, Inc., *React 18 documentation*, <https://reactjs.org/docs/getting-started.html>. Stan na dzień 08.10.2022.
- [8] Meta Platforms, Inc., *Introducing hooks*, <https://reactjs.org/docs/hooks-intro.html>. Stan na dzień 20.11.2022.
- [9] The PostgreSQL Global Development Group, *About postgresql*, <https://www.postgresql.org/about/>. Stan na dzień 22.11.2022.
- [10] The PostgreSQL Global Development Group, *Psql documentation*, <https://www.postgresql.org/docs/current/app-psql.html>. Stan na dzień 11.11.2022.
- [11] Abstract Computing UG (haftungsbeschränkt), *Deta*, <https://docs.deta.sh/docs/home>. Stan na dzień 22.11.2022.
- [12] Railway Corp., *Railway documentation*, <https://docs.railway.app/>. Stan na dzień 08.10.2022.
- [13] *bcryptjs*, <https://www.npmjs.com/package/bcryptjs>. Stan na dzień 25.10.2022.
- [14] Abstract Computing UG (haftungsbeschränkt), *Deta drive documentation*, <https://docs.deta.sh/docs/drive/about>. Stan na dzień 08.10.2022.
- [15] *express-fileupload*, <https://www.npmjs.com/package/express-fileupload>. Stan na dzień 04.11.2022.
- [16] *jsonwebtoken*, <https://www.npmjs.com/package/jsonwebtoken>. Stan na dzień 25.10.2022.
- [17] Carlson, B., *Pg documentation*, <https://node-postgres.com/>. Stan na dzień 10.11.2022.
- [18] *Eslint*, <https://www.npmjs.com/package/eslint>. Stan na dzień 25.10.2022.
- [19] OpenJs Foundation, *Express 4.x documentation*, <https://expressjs.com/en/api.html>. Stan na dzień 08.10.2022.
- [20] bezkoder, *React + node.js express: User authentication with jwt example*, <https://www.bezkoder.com/react-express-authentication-jwt/>. Stan na dzień 01.11.2022.
- [21] PrimeTek, *Primereact components - documentation*, <https://www.primefaces.org/primereact/setup/>. Stan na dzień 08.10.2022.
- [22] Alphabet Inc., *Google map api documentation*, <https://developers.google.com/maps/documentation/maps-static/start>. Stan na dzień 18.10.2022.
- [23] GitHub Inc., *Github documentation*, <https://docs.github.com/en>. Stan na dzień 15.11.2022.

SPIS RYSUNKÓW

2.1.	Diagram przypadków użycia aplikacji	8
2.2.	Diagram klas bazy danych	9
2.3.	Mapa endpointów REST API	10
2.4.	Diagram komponentów strony internetowej	11
2.5.	Diagram wdrożenia aplikacji	12
3.1.	Edytor kolorów	22
3.2.	Edytor kolorów, panel wyboru koloru	22
3.3.	Wybieranie obrazków	23
3.4.	Edytor tekstu	24
3.5.	Nagłówek klienta	24
3.6.	Nagłówek administratora	24
3.7.	Przykładowy wygląd widoku głównego z perspektywy klienta	25
3.8.	Przykładowy wygląd widoku głównego z perspektywy administratora	26
3.9.	Przykładowy wygląd widoku koni z perspektywy klienta	27
3.10.	Przykładowy wygląd widoku koni z perspektywy administratora	28
3.11.	Przykładowy wygląd widoku trenerów z perspektywy klienta	29
3.12.	Przykładowy wygląd widoku trenerów z perspektywy administratora	30
3.13.	Przykładowy wygląd widoku ofert z perspektywy klienta	31
3.14.	Przykładowy wygląd widoku ofert z perspektywy administratora	32
3.15.	Przykładowy wygląd widoku cennika z perspektywy klienta	33
3.16.	Przykładowy wygląd widoku cennika z perspektywy administratora	33
3.17.	Przykładowy wygląd widoku galerii z perspektywy klienta	34
3.18.	Przykładowy wygląd widoku galerii z perspektywy administratora	35
3.19.	Przykładowy wygląd widoku kontaktowego z perspektywy klienta	36
3.20.	Przykładowy wygląd widoku kontaktowego z perspektywy administratora	37
3.21.	Widok logowania	38
3.22.	Widok zmiany hasła	38

SPIS LISTINGÓW

3.1	Tworzenie wybranych tabel: <i>image</i> , <i>horse</i> , <i>image_horse_junction</i> , fragment dbSchema.sql	13
3.2	Dodawanie kluczów obcych do wybranych tabelach: <i>horse</i> , <i>image_horse_junction</i> , fragment dbSchema.sql	14
3.3	Tworzenie wybranych widoków: <i>image_list_view</i> , <i>horse_list_view</i> , fragment dbSchema.sql	14
3.4	Express static routing, fragment src/app.js	15
3.5	Express redirect, fragment src/routes/api.js	15
3.6	Express sendFile, fragment src/routes/index.js	15
3.7	Express routing, fragment src/app.js	16
3.8	Funkcja obsługująca żądanie z metodą PUT wysłane na adres /api/horse , fragment src/routes/api.js	16
3.9	Dodanie rekordu do tabeli <i>horse</i> w Javascript, fragment src/databaseConnector.js	17
3.10	Inicjalizacja połączenia z Deta Drive <i>images</i> , fragment src/databaseConnector.js	17
3.11	Autentykacja, fragment src/routes/authenticator.js	18
3.12	Autoryzacja, fragment src/middleware/authorizator.js	18
3.13	Przykład funkcji <code>fetch()</code> , fragment src/views/HorseView/HorseView.js	19
3.14	Funkcja wysyłająca żądanie aktualizacji danych, fragment src/contextProviders/AuthContextProvider/AuthContextProvider.js	20
3.15	Funkcja logująca, fragment src/contextProviders/AuthContextProvider/AuthContextProvider.js	21
3.16	Funkcja wylogowująca, fragment src/contextProviders/AuthContextProvider/AuthContextProvider.js	22
3.17	Przykład użycia komendy <code>export</code>	39
3.18	Komendy uruchamiające serwer	40