System zamówień

JavaScript + React

Jakub Starmach

listopad 2025

Spis treści

I. Ogólny opis projektu

Temat

Aplikacja System Zamówień ma na celu umożliwienie użytkownikowi dodawanie klientów, tworzenie zamówień i ich przeglądanie w kalendarzu.

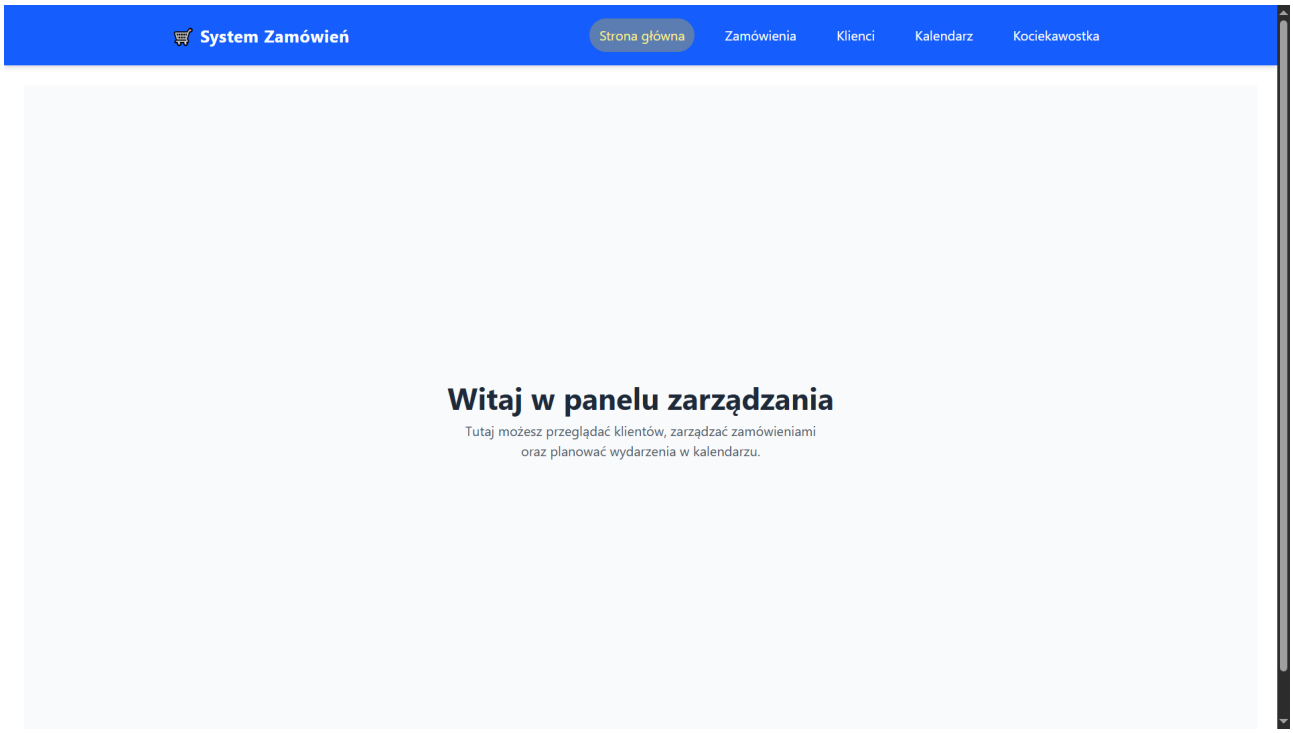
Przeznaczenie

Aplikacja jest przeznaczona dla wszystkich, od małych firm po samotnych działaczy. Nawet studenci czy uczniowie znajdą w niej zastosowanie.

Podstrony

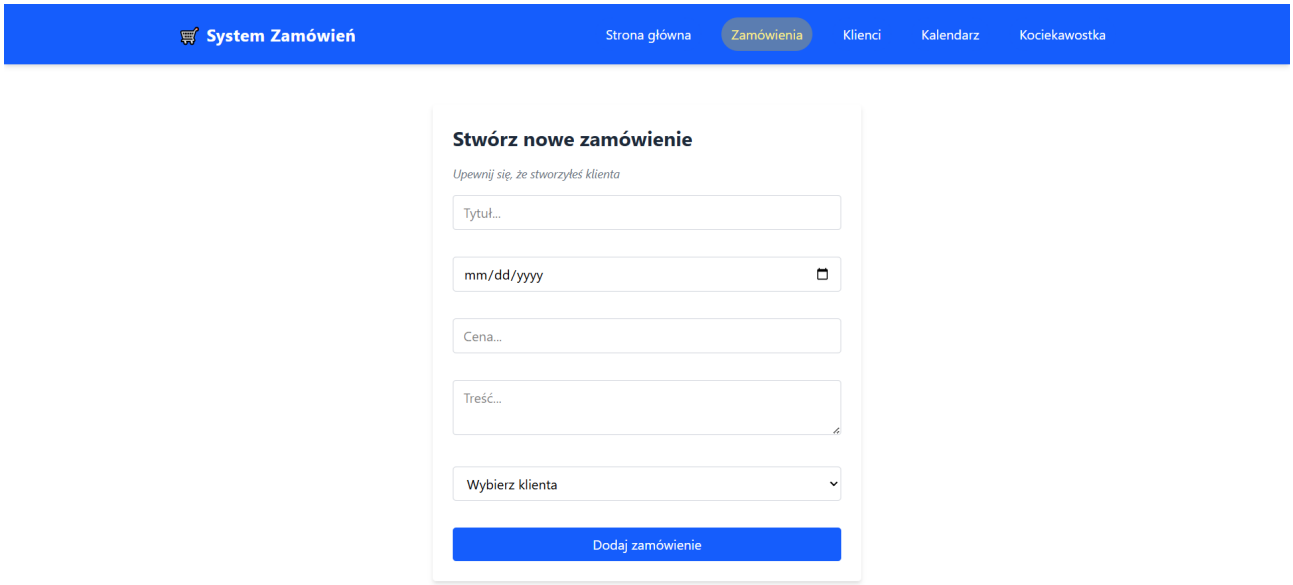
Aplikacja dzieli się na kilka podstron:

Strona główna

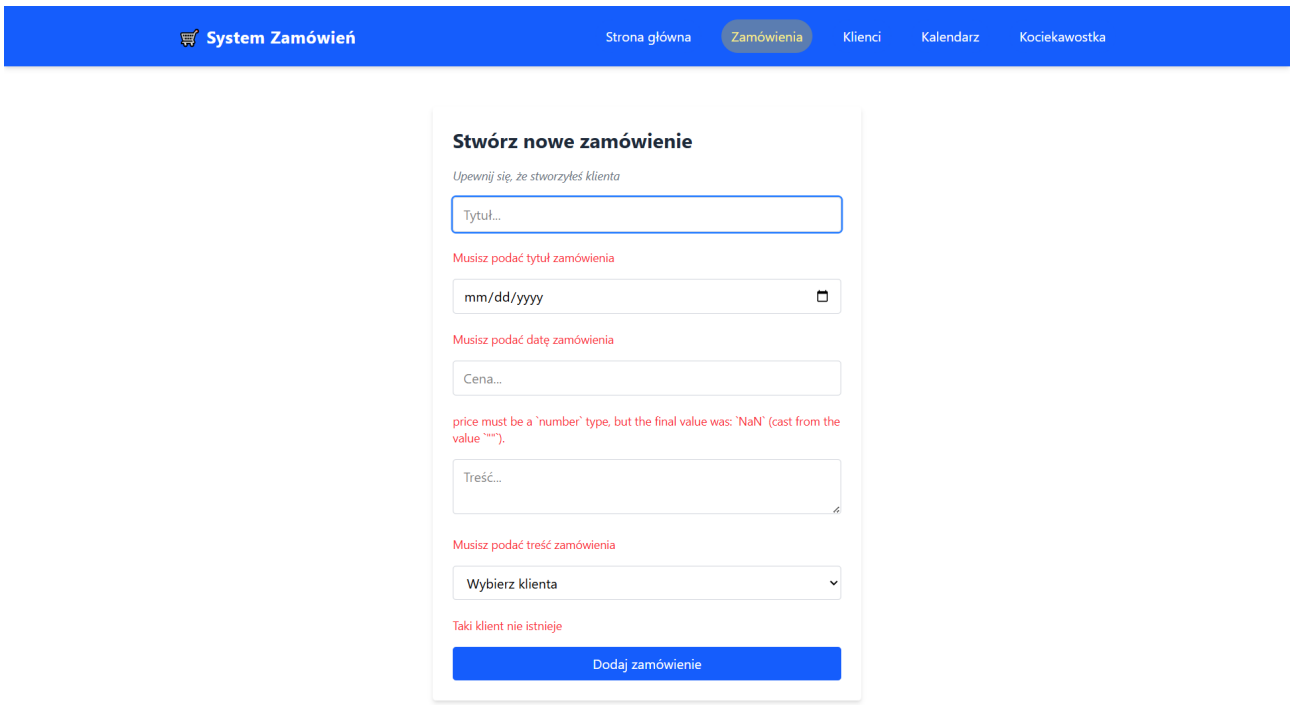


Na stronie głównej nic się nie dzieje

Zamówienia

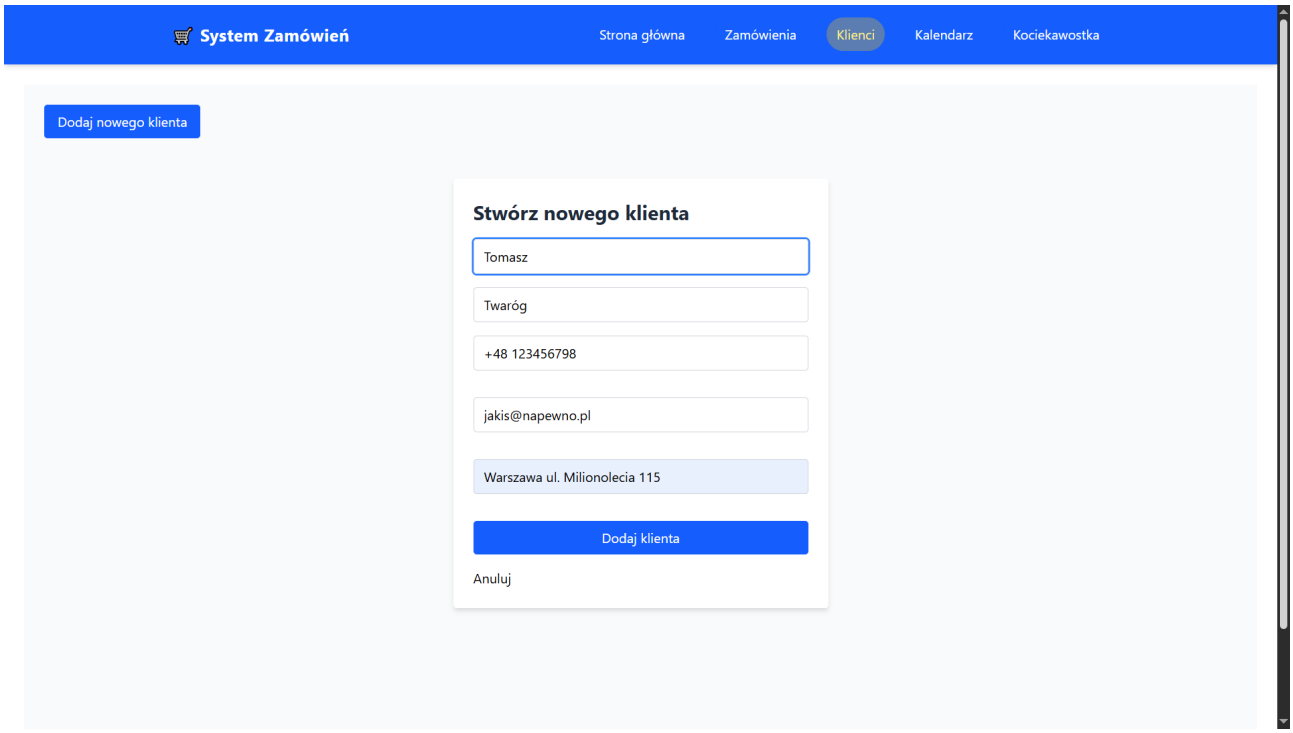


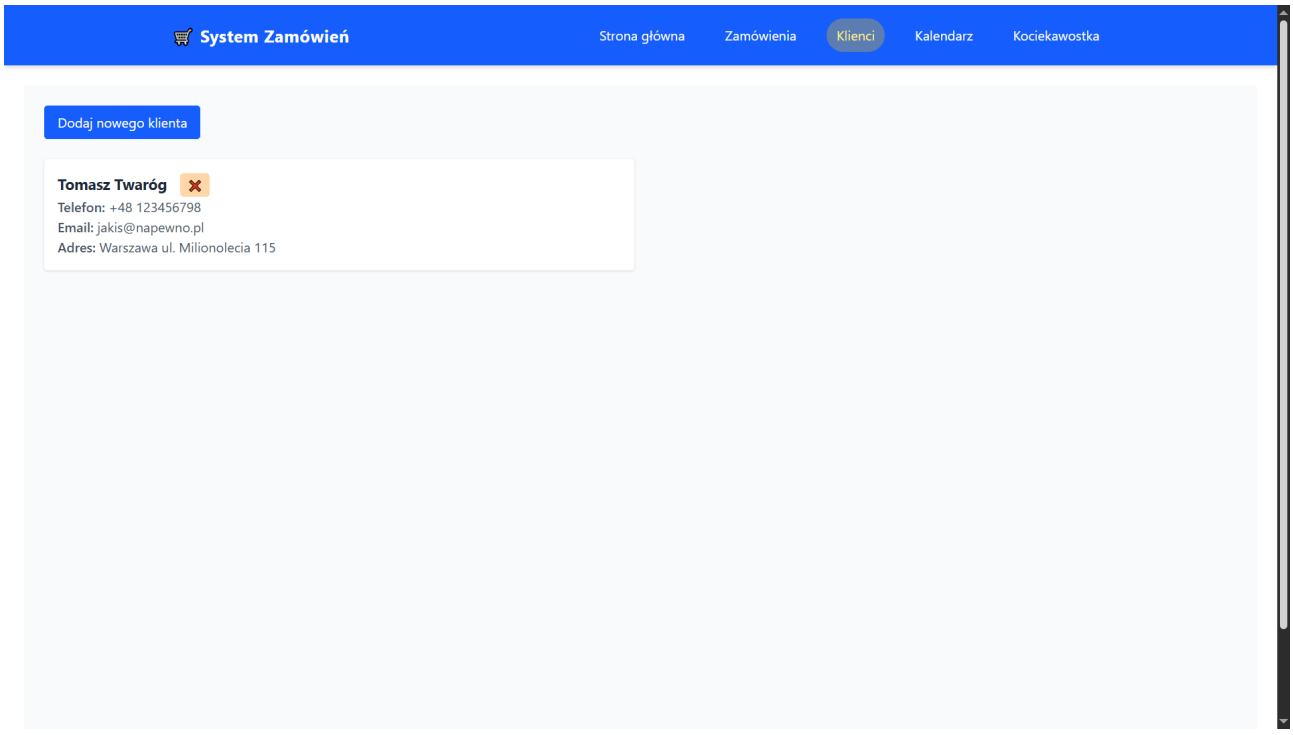
Tutaj użytkownik może dodawać zamówienia. Każde pole jest obowiązkowe i poinformuje użytkownika o ewentualnych błędach. Po prawidłowym utworzeniu zostanie przekierowany od razu do kalendarza



Klienci

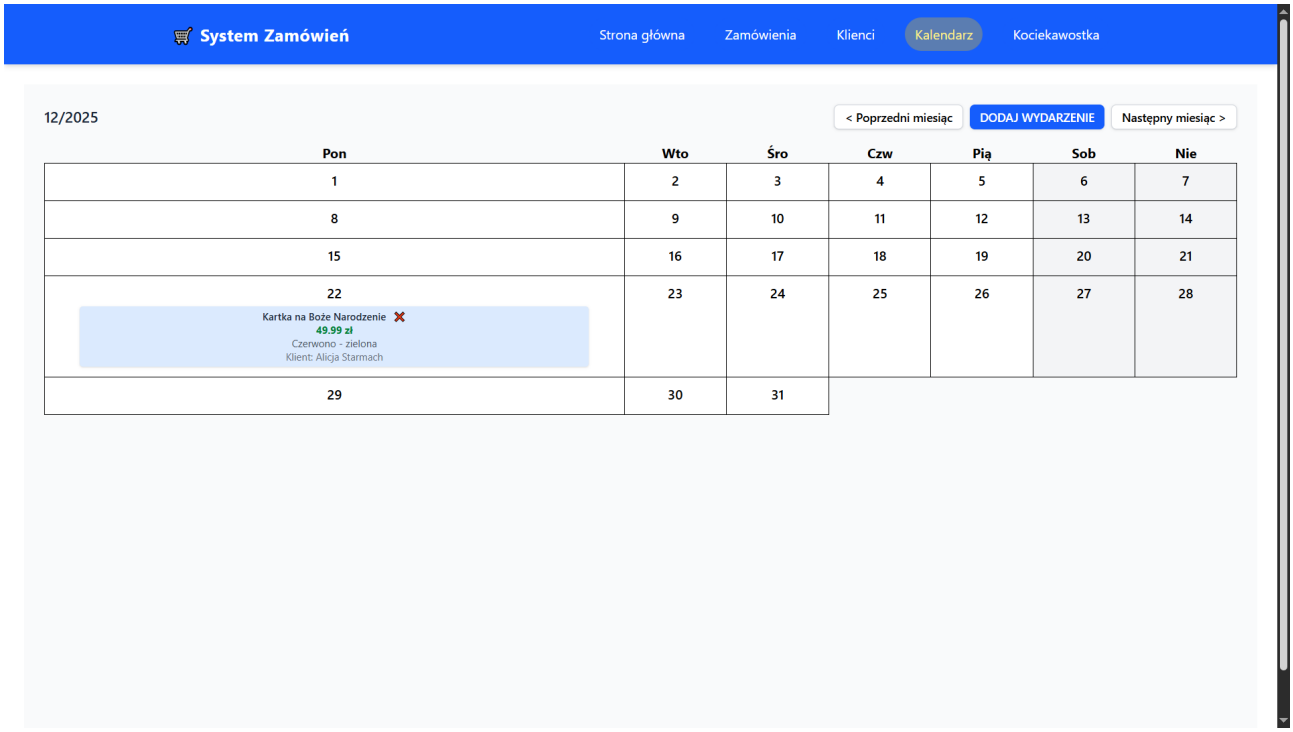
Tutaj użytkownik może dodawać klientów i ich przeglądać / usuwać





Kalendarz

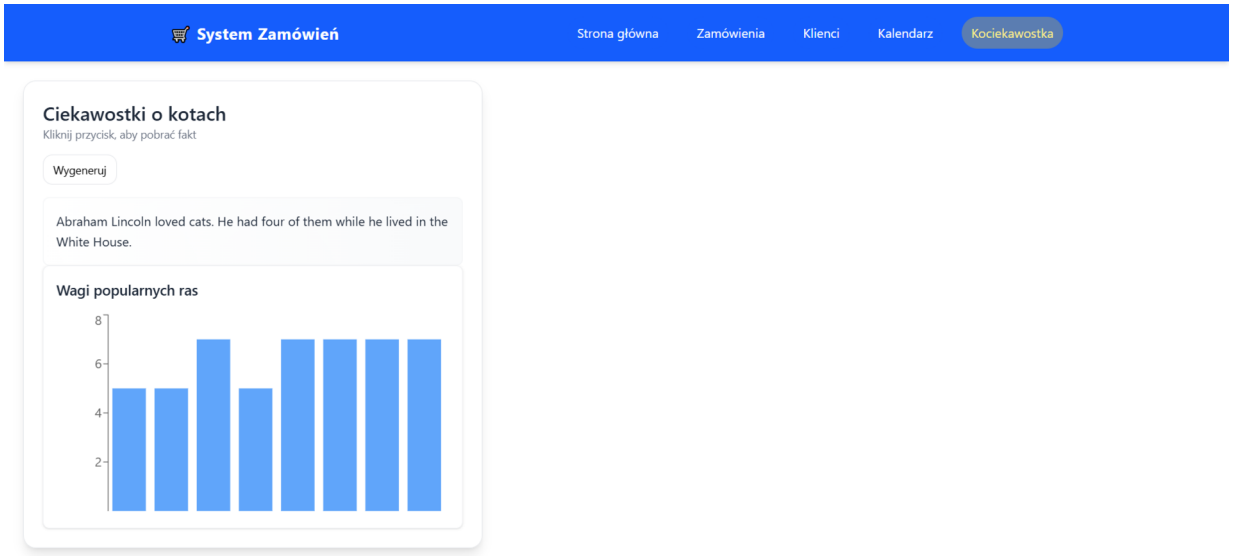
Tutaj użytkownik może przeglądać kalendarz, przełączać się między miesiącami i podglądać dodane wydarzenia / zamówienia.



Przycisk „Dodaj wydarzenie” dodaje przykładowe wydarzenie dnia 29 listopada 2025

Kociekawostka

Ta strona po kliknięciu przycisku pobiera ze strony catfact.ninja losową ciekawostkę o kotach. Poniżej wyświetlany jest wykres – średnia waga kota w zależności od rasy – dane pobierane są ze strony api.thecatapi.com



II. Mechanika projektu

Komponent useCalendar

Import funkcji

Komponent **useCalendar** jest najważniejszym w całym projekcie, ponieważ jest on **hookiem**, który umożliwia nam tworzenie **terminarza** i dodawanie do niego wydarzeń. Aby z niego skorzystać należy zaimportować „useCalendar” oraz pobrać:

- calendar – obiekt z zapisanymi wydarzeniami, pobieranymi z lokalnej pamięci
- setCalendar – funkcja służąca do ustawienia wartości całego kalendarza. Używa ona do tego useState z Reacta

```
const [calendar, setCalendar] = useState(() => {
  const savedCalendar = localStorage.getItem("calendar");
  return savedCalendar ? JSON.parse(savedCalendar) : {};
});
```

oraz useEffect z Reacta do automatycznego zapisywania

```
useEffect(() => {
  localStorage.setItem("calendar", JSON.stringify(calendar));
}, [calendar]);
```

- addEvent – funkcja służąca do dodawania nowych wydarzeń w kalendarzu
- removeEvent – funkcja służąca do usuwania wydarzeń z kalendarza na podstawie dnia i tytułu
- getDay i getMonth – funkcje zwracające tablice wydarzeń z danego dnia i tablicę dni z danego miesiąca

Całość powinna wyglądać tak:

```
import { useCalendar } from "./pages/useCalendar";
const { calendar, addEvent, getDay, getMonth, removeEvent } = useCalendar();
```

addEvent

Funkcja ta umożliwia dopisanie do kalendarza nowego wydarzenia

Argumenty

Użytkownik przekazuje do funkcji:

a) obiekt daty, czyli { year, month, day } (np. { year:2025, month:11, day:1 })

// warto zauważyć, że przekazujemy numery dni, miesięcy i lat odpowiadające ich numerom w kalendarzu, nie zaczynamy liczenia od 0. Jeżeli podamy nieprawidłowe wartości, to funkcja zwróci do konsoli błąd i zakończy wykonanie

```
if (day < 1 || month < 1 || month > 12 || month > 12) {
    console.error("Tried to create Event in non existing day!");
    return;
}
```

b) dane o wydarzeniu, czyli obiekt. Może to być sam tytuł, ale również opis, cena, lista uczestników itp.

Mechanika

Po walidacji danych funkcja korzysta z setCalendar, czyli useState.

```
setCalendar((prev) => {
  const updated = { ...prev };
  if (!updated[year]) updated[year] = {};
  if (!updated[year][month]) updated[year][month] = {};
  if (!updated[year][month][day]) updated[year][month][day] = [];
  const exists = updated[year][month][day].some(
    (event) => event.title === data.title
  );
  if (!exists) updated[year][month][day].push(data);
  return updated;
});
```

a) Używa ona wewnętrznie stworzonej funkcji, która pobiera **prev**, czyli kalendarz przed zmianą

b) Następnie przypisuje do stałej **updated** obiekt prev z dostępnymi do tej pory danymi

c) Sprawdza, czy istnieją obiekty danego dnia. Jeśli nie istnieją, to tworzy je i przypisuje pusty obiekt, a w przypadku dnia pustą listę

- Kalendarz to lista obiektów **year**, rok to lista obiektów **month**, miesiąc to lista obiektów **day**, a dzień to lista obiektów **event**

d) Gdy dany dzień w kalendarzu będzie dostępny, to sprawdza, czy dane wydarzenie tego dnia o tej samej nazwie już nie istnieje (**title** to element unikalny dla wydarzeń danego dnia)

   - Używa do tego funkcji **some**, która przeszukuje listę (iteruje po niej biorąc pod uwagę pojedynczy element, czyli **event** i porównuje jego tytuł z tytułem nowego wydarzenia, które funkcja chce dodać)

e) Jeżeli istnieje, to przypisuje do kalendarza obiekt przed zmianą

f) Jeżeli nie istnieje, to używa funkcji **push** żeby dodać nowy element do listy na końcu tablicy

g) Na koniec zwraca nadpisany kalendarz

removeEvent

Funkcja ta umożliwia usunięcie z kalendarza danego wydarzenia

Argumenty

Użytkownik przekazuje do funkcji:

a) obiekt daty, czyli { year, month, day } (np. {year:2025, month:11, day:1})

b) tytuł wydarzenia, który jest unikalny dla danego dnia

Mechanika

Funkcja korzysta z setCalendar, czyli useState.

a) Funkcja najpierw wywołuje okno potwierdzenia, żeby użytkownik przez przypadek nie usunął wydarzenia

b) Jeżeli potwierdzi usunięcie, to przekazuje do setCalendar wewnętrznie stworzonej funkcji, która:

   a) Sprawdza, czy dany dzień w ogóle istnieje

      - Jeżeli nie, to zwraca błąd i kończy działanie funkcji poprzez zwrócenie niezmienionego poprzedniego kalendarza

   b) Jeżeli istnieje, to używa funkcji **filter**, która iteruje po całej tablicy biorąc pod uwagę pojedynczy element, czyli **event** i sprawdza, czy tytuł tego elementu jest równy tytułowi, który podaliśmy do funkcji. Jeżeli jest równy, to go usuwa

   c) Na koniec zwraca nadpisany kalendarz

getDay

Funkcja ta umożliwia pobranie wybranego dnia (tablicy obiektów wydarzeń) z kalendarza

Argumenty

Użytkownik przekazuje do funkcji: obiekt daty, czyli **{ year, month, day }** (np. { year:2025, month:11, day:1 })

Mechanika

Funkcja zwraca dany dzień sprawdzając, czy owy w ogóle istnieje (sprawdza rok, miesiąc, a dopiero potem dzień). W przypadku gdy dany dzień nie istnieje to zwraca obiekt pusty **null**.

```
[ { title: "Christmas Party", description: "At my place!" } ] | null
```

getMonth

Funkcja ta działa bardzo podobnie do getDay z tą różnicą, że nie zwraca tablicy dnia, a miesiąca

Argumenty

Użytkownik przekazuje do funkcji: obiekt daty, czyli **{ year, month }** (np. { year:2025, month:11 })

Mechanika

Funkcja zwraca dany miesiąc sprawdzając, czy owy w ogóle istnieje. W przypadku gdy dany miesiąc nie istnieje to zwraca **pustą tablicę.**

```
[ { day: 24, events: [
{ title: "Christmas Party", description: "At my place!" }
] } ]
```

Dostęp do wydarzeń

Aby uzyskać dostęp do danego wydarzenia należy:

a) uzyskać miesiąc: **month**

b) określić element listy: **[i]**

- jest to lista, więc dni miesiąca nie będą numerowane po kolei, tylko tak jak zwykłe elementy listy poczynając od **zera**

c) teraz mamy dostęp do:

- numeru dnia w kalendarzu: **1-31**
- listy wydarzeń danego dnia: **.events**

i. nie jesteśmy pewni, czy dany dzień zawiera jakiekolwiek wydarzenia, więc najlepiej użyć **?.events**

ii. każde wydarzenie to obiekt, więc mamy dostęp do:

a) tytuł wydarzenia: .**title** (należy go podać przy dodawaniu wydarzenia, ponieważ służy od do identyfikowania wydarzeń)

b) innych elementów, które podał użytkownik (np. .description)

Iteracja po wszystkich dniach miesiąca: (mapowanie kopii)

```
month((element) => element.day) // numer dnia
```

Iteracja po wszystkich wydarzeniach któregoś dnia miesiąca: (mapowaniekopii)

```
month[0]?.events.map((element) => element.title) // tytuł wydarzeń
z pierwszego dnia dodanego do tego miesiąca
```

1. Strona aplikacji

To serce całego projektu, które zapewnia podstronom dostęp do kalendarza, klientów, buduje strony i tworzy ścieżki do podstron.

Context Providery

Strona obsługuje dwa konteksty:

- kalendarza

```
export const MainCalendarContext = createContext();

const { calendar, addEvent, getDay, getMonth, removeEvent } = useCalendar();

<MainCalendarContext.Provider
  value={{ calendar, addEvent, getDay, getMonth, removeEvent }}
>
```

- klientów (useState – lista klientów zapisywana lokalnie)

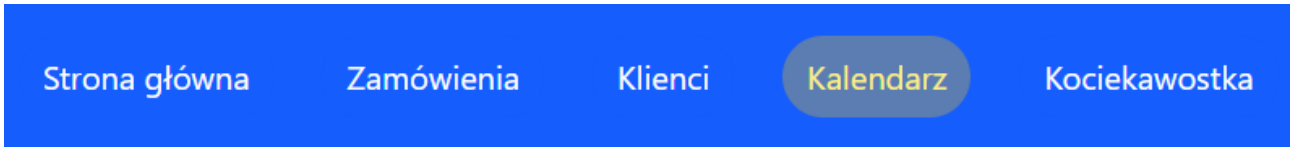
```
export const ClientsContext = createContext();

const [clients, setClients] = useState(() => {
    const savedClients = localStorage.getItem("clients");
    return savedClients ? JSON.parse(savedClients) : [];
  });

useEffect(() => {
    localStorage.setItem("clients", JSON.stringify(clients));
  }, [clients]);

<ClientsContext.Provider value={{ clients, setClients }}>
```

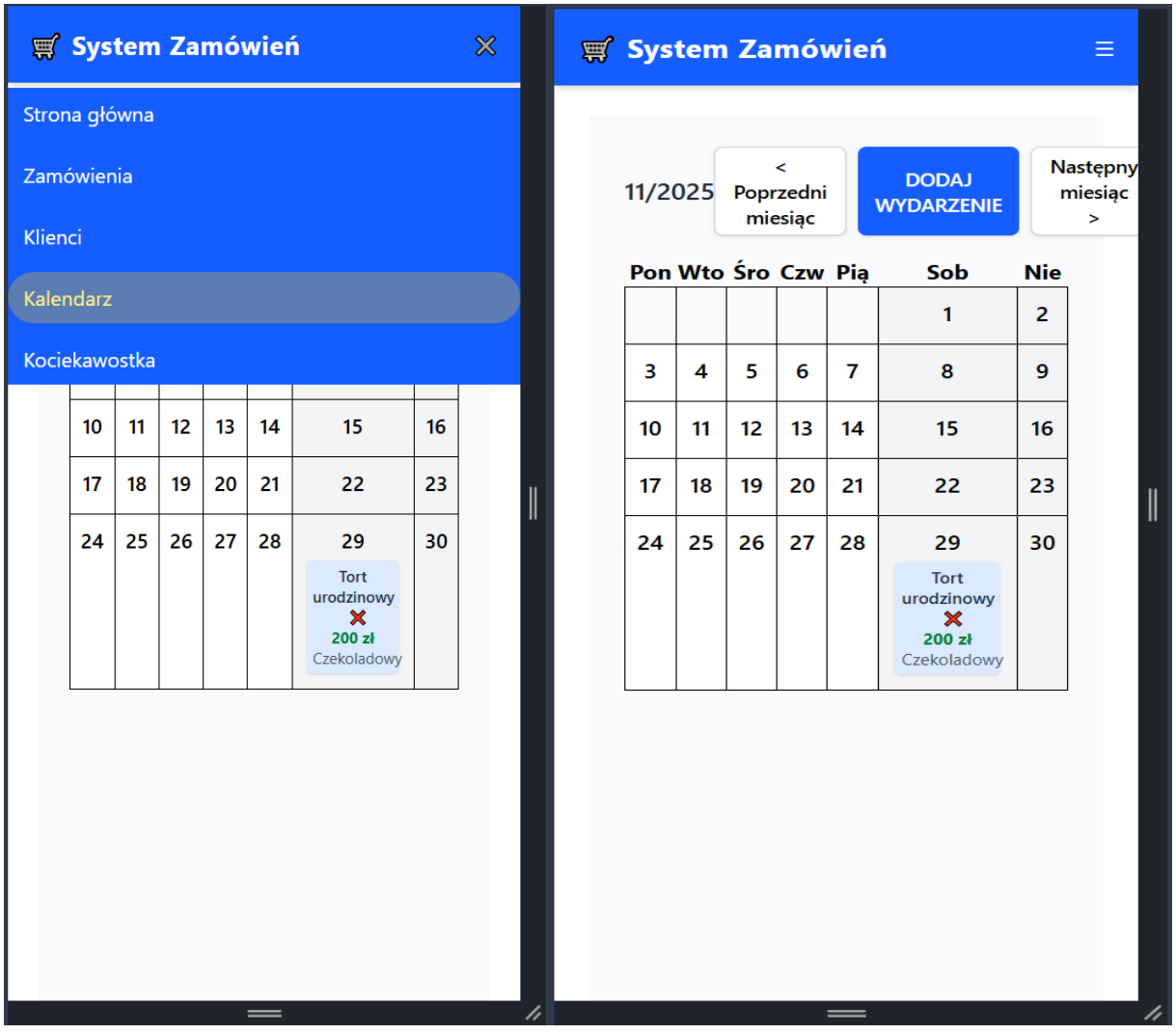
Następnie strona tworzy ścieżki do podstron oraz renderuje linki w nagłówku za pomocą własnego komponentu **NavLink**, który zmienia wygląd w zależności od tego na której stronie się jest i czy najeżdża się kursorem na link.



Ma ona również mały element responsywności – na telefonach można schować lub rozwinąć nawigację dzięki fladze **menuOpen** i **setMenuOpen** (**useState**)

```
const [menuOpen, setMenuOpen] = useState(false);
```

```
<div className="md:hidden">
    <button
    onClick={() => setMenuOpen(!menuOpen)}
    className="focus:outline-none"
    >
        {menuOpen ? "✖" : "☰"}
    </button>
</div>
```



2. Kalendarz

To podstrona, na której użytkownik może przeglądać kalendarz – miesiące, dni i wydarzenia

Stałe

- const { calendar, addEvent, getMonth, removeEvent }
  = useContext(MainCalendarContext); - funkcje z hooka useCalendar
- **daysOfWeek** – lista skrótów nazw dni tygnodna; używana przy renderowaniu
- **currentMonth** – useState; obecny przeglądany miesiąc; obiekt { year, month }; domyślnie listopad 2025
- **firstDay** – useState; liczba mówiąca, który dzień w obecnie przeglądanym miesiącu jest pierwszym dniem miesiąca (6 – sobota); domyślnie
  1 – poniedziałek
- **eventsInMonth** – useState; lista wydarzeń danego dnia; domyślnie pusta
- **daysInMonth** – lista dni w każdym miesiącu, np. 1:31 – styczeń 31 dni; używana przy renderowaniu

Metody

useEffect

Na tej podstronie jest wywoływana dwa razy:

- do pierwszej przekazywana funkcja getFirstDay. Uruchamia się przy zmianie obecnego miesiąca currentMonth
- do drugiej przekazywana jest funkcja setEventsInMonth (czyli useState), do której jest przekazywana funkcja getMonth z hooka useCalendar, do której jest przekazywany obecny miesiąc currentMonth. Uruchamia się przy zmianie obecnego miesiąca currentMonth i zawartości kalendarza calendar

handleButton

Odpowiada za funkcję przyciska „Dodaj wydarzenie” w kalendarzu. W tym przypadku dodaje przykładowe wydarzenie do kalendarza, czyli dnia 29.11.2025 „Tort urodzinowy” o treści „Czekoladowy”. Jest to również przykład zastosowania funkcji addEvent w kodzie

changeMonth

Odpowiada za zmianę wyświetlanego miesiąca. Przekazuje się do niej kierunek (**„next", „prev"**). Na podstawie niego zmienia currentMonth tak, żeby odpowiednio zmieniało nie tylko miesiąc, ale również rok. Używa funkcji setCurrentMonth (useState).

getFirstDay

Ustawia firstDay na podstawie daty. Używa klasy **Date**, odczytuje odpowiedni pierwszy dzień danego miesiąca i przekształca go na numer dnia tygodnia za pomocą funkcji **getDay**.

Renderowanie miesiąca

Kalendarz jest renderowany przy pomocy tablicy, Domyślnie jest nagłówek, w którym jest mapowana kopia tablicy skrótów nazw dni tygodnia, a następnie w ciele tabeli jest renderowany własny komponent **RenderMonth**, do którego za pomocą propsów są przekazywane: firstDay, eventsInMonth, datsInMonths[currentMonth] (czyli liczba dni w danym miesiącu), removeEvent oraz currentMonth.

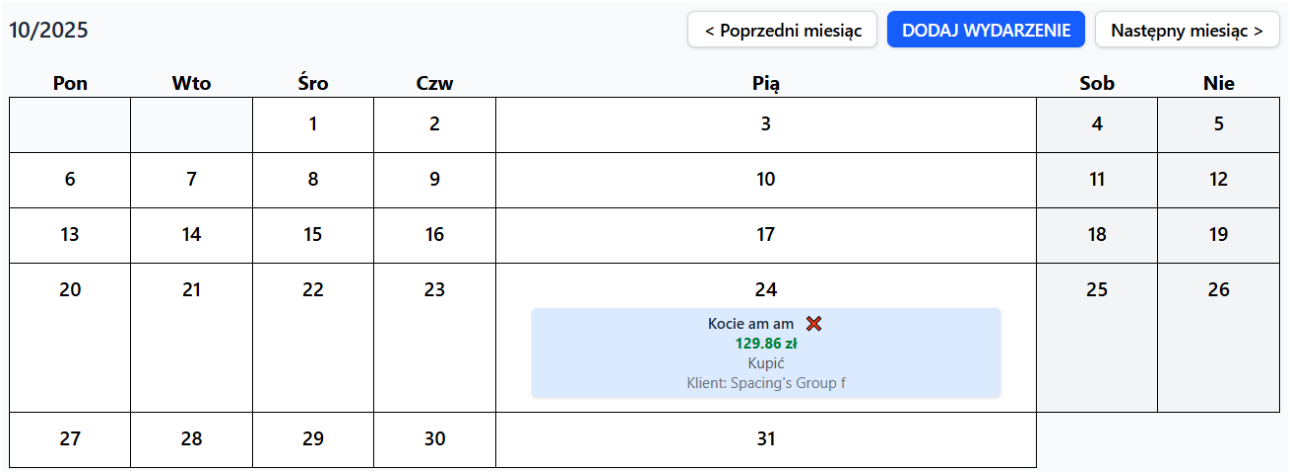
RenderMonth

To komponent, który buduje ciało kalendarza, czyli każdy wiersz i komórkę.

1. Na początku tworzy pustą tablicę **monthToRender**, czyli tablicę komórek całego miesiąca do wyrenderowania. Wpisuje do niego kilka „komórek zapychaczy", czyli odpowiednią ilość dni tak, żeby miesiąc zaczynał się od odpowiedniego dnia miesiąca
2. Następnie używa pętli for, aby dodać do tablicy wszystkie dni miesiąca (dodaje ich tyle, ile jest dni w miesiącu, czyli używa stałej daysInMonth pobranej z propsów)
    a) Szuka w miesiącu (tablicy obiektów) dnia, którego numer jest równy obecnemu wykonaniu tablicy (w tym przypadku **n_day** /* day number */ ale jest to odpowiednik **i**) używając do tego funkcji **find** (przekazuje wewnętrzną funkcję, która iteruje po wszystkich elementach, każdy element to **d**, następnie porównuje numer wykonania pętli **n_day** z numerem dnia **d.day**)
    b) Tworzy komórkę tabeli td, w której znajdują się:
        - numer dnia n_day

- divy z wydarzeniami, używa do tego funkcji **map** – iteruje po kopiach wszystkich elementów każdy element to **event** o unikalnym identyfikatorze **i**, następnie dla każdego elementu tworzy div z przyciskiem do usunięcia danego wydarzenia (funkcja **removeEvent**, do której jest przekazywany obiekt daty oraz tytuł tego wydarzenia) oraz wszystkie dane danego wydarzenia – tytuł, cena, opis oraz klient

3. Na samym końcu renderuje całe ciało tablicy ponownie używając funkcji **map** na tablicy **monthToRender** (każdy element to **day** o identyfikatorze **i**)
   a) W przypadku, gdy identyfikator i jest podzielny przez 7 (łącznie z gdy i = 0) to najpierw tworzy nowy wiersz tabeli, a potem używa funkcji **slice** żeby podzielić monthToRender i zostawić tylko elementy o identyfikatorze od i do i+7 (czyli od poniedziału do niedzieli) i dopiero wtedy renderuje siedem komórek wiersza, każdy wiersz osobno
   b) Jeżeli jest niepodzielny przez siedem, to nic nie robi, pomija wykonanie



3. Zamówienia

Na tej podstronie użytkownik może dodawać zamówienia do kalendarza

!!! przed dodaniem nowego zamówienia należy upewnić się, że dodało się odpowiedniego klienta w zakładce Klienci !!!

Stałe

Funkcja deklaruje trzy stałe: **addEvent** (z kontekstu kalendarza), **clients** (lista klientów z konteksutu klientów) oraz **navigate**, czyli funkcja pozwalająca nawigować automatycznie między stronami dzięki hookowi **useNavigate**.

Metody

ConfirmedNewSubmit

To funkcja, która odpowiada za dodanie nowego wydarzenia i przekierowanie użytkownika do kalendarza. Pobiera ona dane z formularza

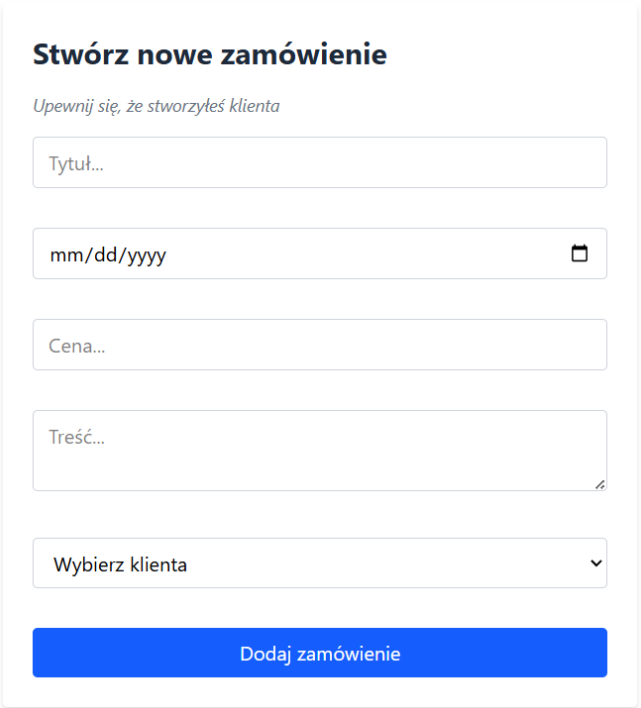
- Na początku dzieli datę na rok, miesiąc i dzień przy pomocy funkcji **split** i przekazując znak podziału (dla input type=”date” jest to myślnik „-”)
- Dodaje wydarzenie
- Automatycznie przekierowuje do kalendarza

Schemat formularza

Aby utworzyć nowe zamówienie należy odpowiednio wypełnić formularz:

- Tytuł: tekst, obowiązkowy
- Data: tekst, obowiązkowa
- Cena: liczba, obowiązkowa, nieujemna
- Opis: tekst, obowiązkowy
- Klient: tekst, obowiązkowy, musi być wybrany z listy klientów

```
.oneOf(
  clients.map((c) => `${c.name} ${c.surname}`),
  "Taki klient nie istnieje"
)
```



4. Klienci

Na tej podstronie użytkownik może dodać nowego klienta. Jeżeli już dodał jakiegokolwiek, to wyświetla ich prostą listę.

Składa się z podstron **clients** oraz **newClient**

Metody

setClients, setCreateClient

To dwie metody, które wykorzystują **useState** z Reacta, aby zapisywać listę klientów oraz dane o nowym kliencie. Tablica clients oraz funkcja setClients jest dostarczana przez kontekst **ClientsContext**.

deleteClient

Odpowiada za usuwanie klienta z listy. Najpierw pyta użytkownika o potwierdzenie usunięcia, a następnie filtruje listę

```
const deleteClient = (client) => {
  const confirmDelete = window.confirm(
    "Czy jesteś pewien, że chchesz usunąć " +
      client.name +
      " " +
      client.surname +
      "?"
  );
  if (confirmDelete)
    setClients((prev) => prev.filter((_client) => _client !== client));
};
```

Używa do tego funkcji **filter** – zostawia tylko te elementy, które nie są równe klientowi do usunięcia.

confirmedNewSubmit

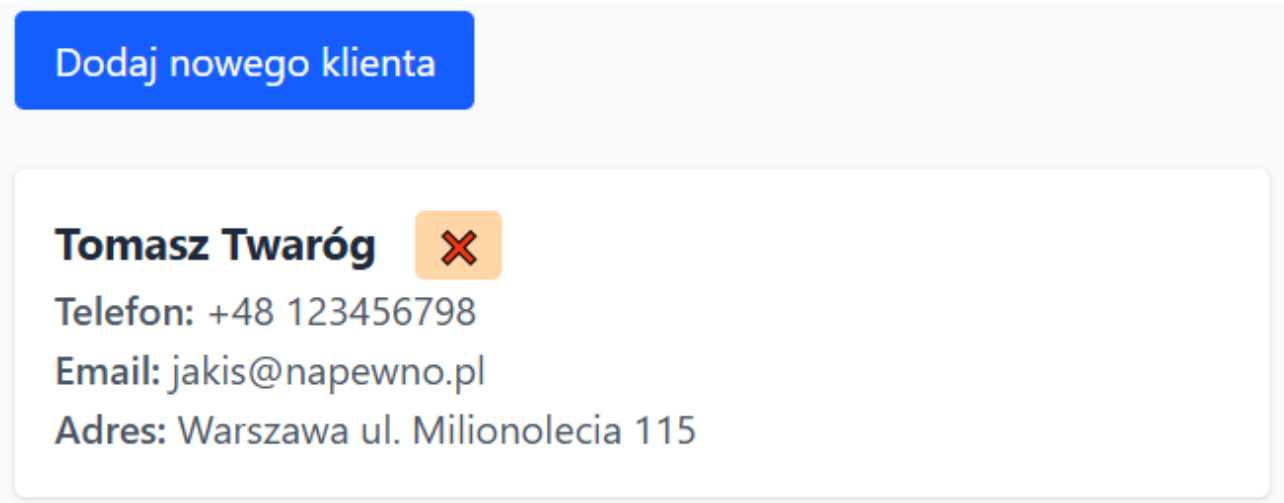
Dodaje nowego klienta na końcu listy oraz czyści flagę createClient

```
const confirmedNewSubmit = (data) => {
  setCreateClient(false);
  setClients((prev) => [...prev, data]);
};
```

Schemat formularza

Aby utworzyć nowego klienta należy odpowiednio wypełnić formularz:

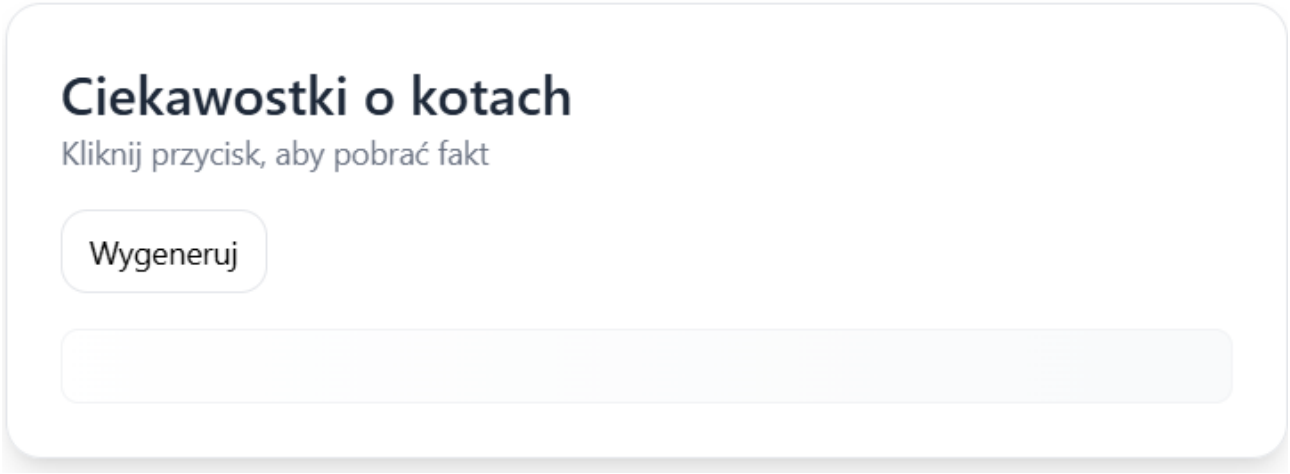
- Imię: tekst, obowiązkowe
- Nazwisko: tekst, obowiązkowe
- Numer telefonu: tekst, obowiązkowy, co najmniej 9 znaków
- Email: tekst, obowiązkowy, musi być prawidłowym mailem
- Adres: tekst, obowiązkowy



5. Kociekawostka

To ostatnia podstrona i zarazem komponent. Używa ona **Axios** i API, aby pobrać losowy fakt o kotach ze strony catfact.ninja i wyświetlić go na stronie (w oryginalnym, języku angielskim). Używa ona do tego również **useState** z Reacta o nazwie fact

```
const generateFact = () => {
  Axios.get("https://catfact.ninja/fact").then((response) => {
    setFact(response.data.fact);
  });
};
```



Aby wygenerować wykres wykorzystuje **useEffect** z Reacta, aby pobrać dane raz przy załadowaniu strony. Następnie w metodzie asynchronicznej **fetchBreeds** (to znaczy wykonującej się niezależnie od budowania strony) pobierane są dane ze strony api.thecatapi.com i przypisywane do **breedData** – useState z Reacta przechowującego tablicę obiektów o polach **name** i **weight**.

```
  const fetchBreeds = async () => {
    const response = await Axios.get("https://api.thecatapi.com/v1/breeds");
    const breeds = response.data.slice(0, 8).map((breed) => ({
      name: breed.name,
      weight: parseInt(breed.weight.metric.split(" - ")[1]),
    }));
    setBreedData(breeds);
  };

  useEffect(() => {
    fetchBreeds();
  }, []);
```

Następnie dane są wyświetlane przy pomocy pobranej biblioteki **recharts**

```
import { ResponsiveContainer, BarChart, Bar,
XAxis, YAxis, Tooltip } from "recharts";

<ResponsiveContainer
  width="100%"
  height="100%"
  minWidth={0}
  minHeight={0}
  >
  <BarChart data={breedData}>
    <XAxis dataKey="name" hide />
    <YAxis />
    <Tooltip />
    <Bar dataKey="weight" fill="#60a5fa" />
  </BarChart>
</ResponsiveContainer>
```

- ResponsiveContainer – „pudełko” na wykres
- BarChart – główny komponent
- Bar – Pojedyncza seria danych
- XAxis – oś X (nazwy)
- YAxis – oś Y (wartości)
- Tooltip – informacja po najechaniu