

Zaawansowane Techniki Programowania

Dokumentacja projektu końcowego

Temat: Aplikacja schroniska dla zwierząt

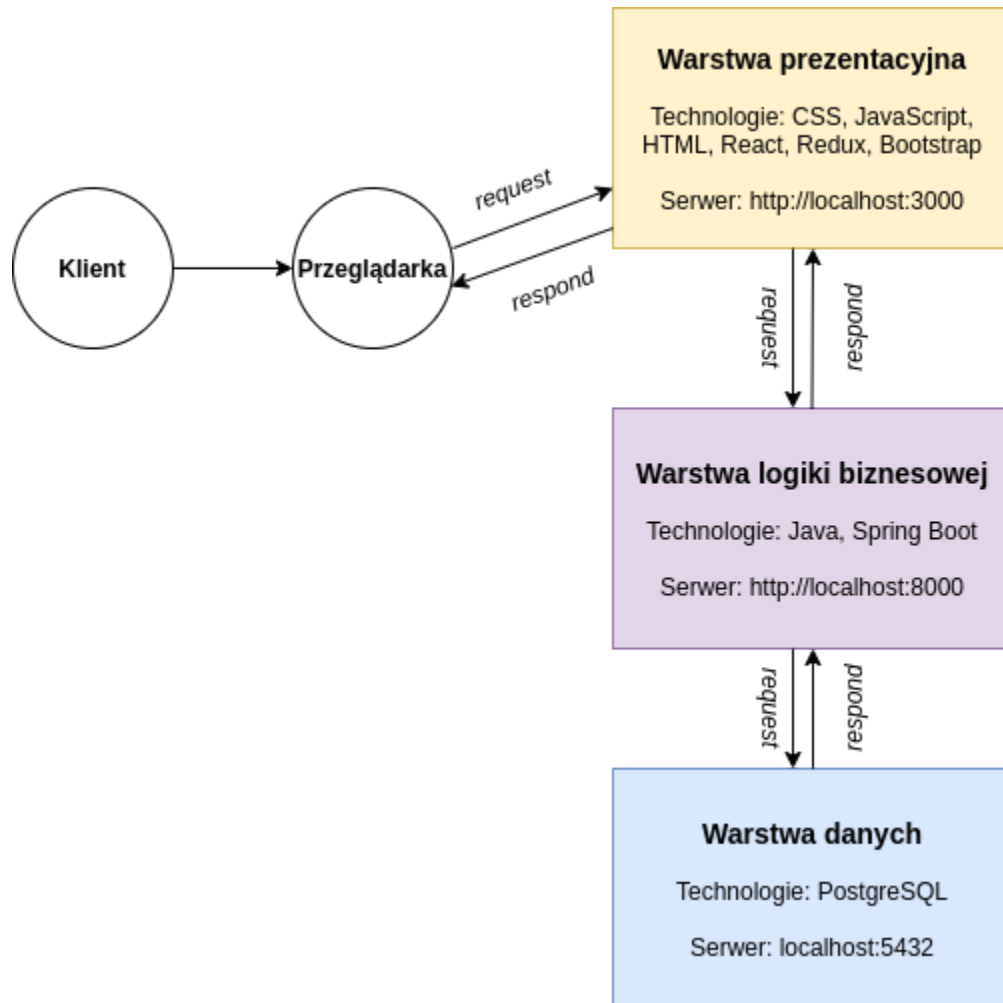
Radosław Bujak 125583

Michał Borowski 125578

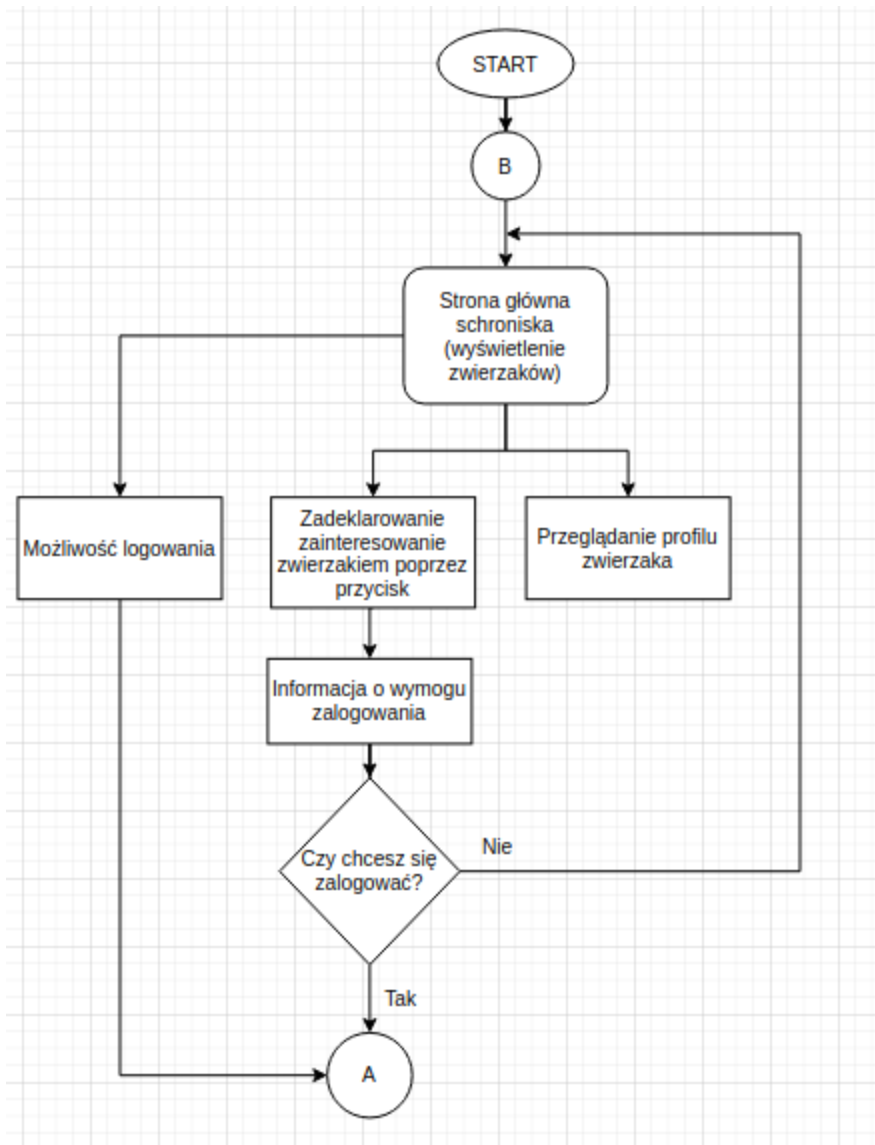
Opis funkcjonalności aplikacji:

W ramach projektu została wykonana aplikacja wspomagająca schroniska dla zwierząt. Użytkownicy mogą wyświetlać dostępne do adopcji zwierzęta oraz deklarować zainteresowanie danym pupilem. Aplikacja zbiera informację, które zwierzęta cieszą się największą popularnością tak, aby pomóc każdemu z nich znaleźć swój nowy dom. Pracownik schroniska (recepjonista) może dodawać nowo przybyłe do schroniska zwierzęta oraz usuwać ogłoszenia o zaadaptowanych już zwierzętach. Admin może przeglądać informacje o użytkownikach aplikacji, usuwać konta użytkowników oraz dodawać nowych pracowników schroniska.

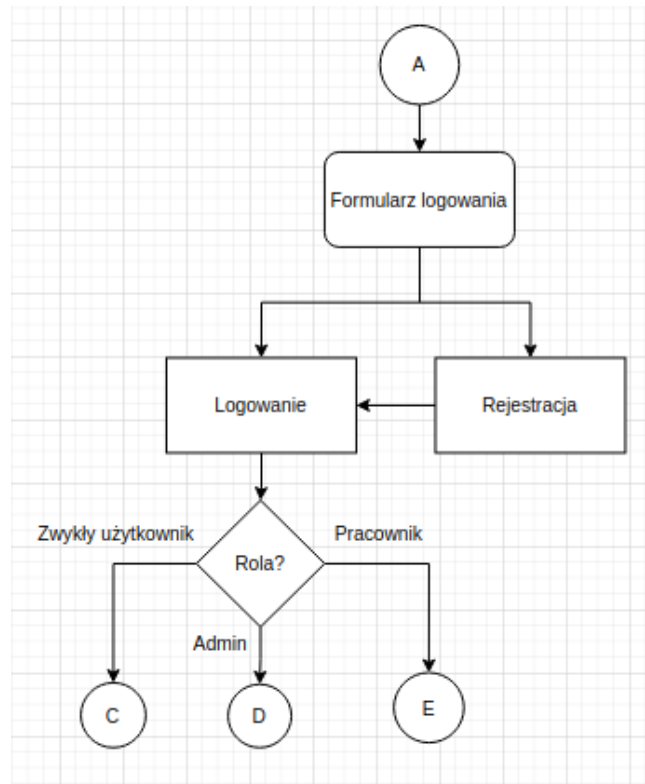
Schemat działania aplikacji:



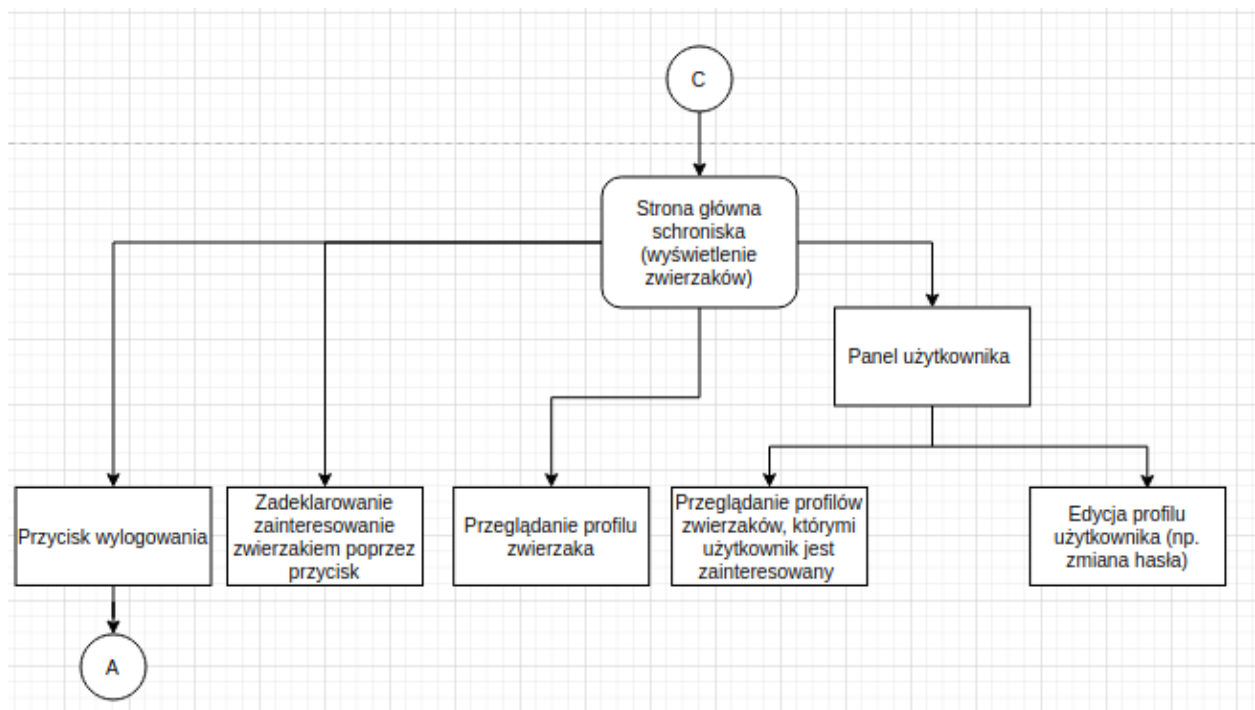
Rys. 1.1 Architektura aplikacji shelter.



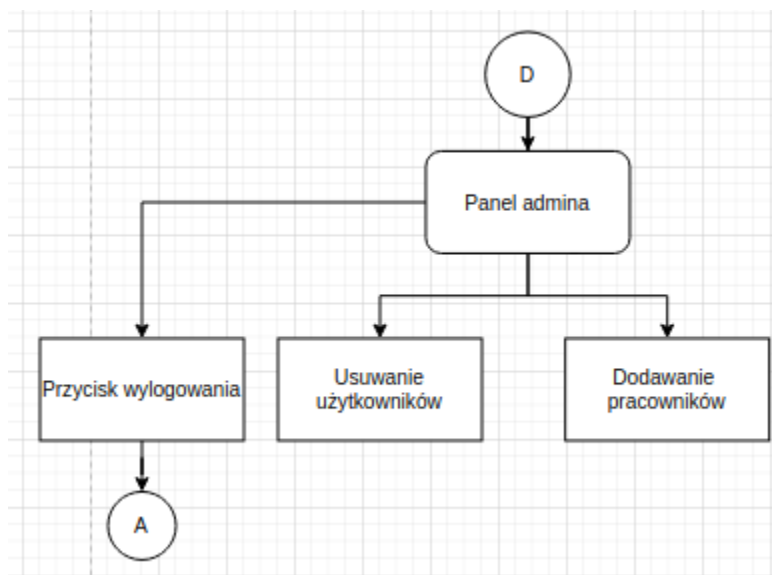
Rys. 1.2 Diagram niezalogowanego użytkownika, scenariusz B.



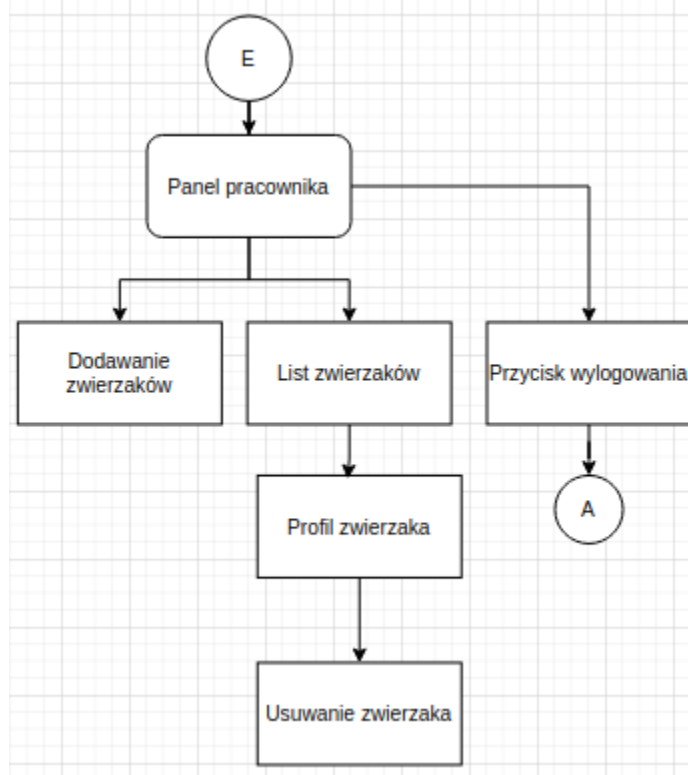
Rys. 1.3 Diagram logowania, schemat A.



Rys. 1.4 Diagram zalogowanego użytkownika, rola użytkownik, schemat C.



Rys. 1.5 Diagram zalogowanego użytkownika, rola admin, schemat D.



Rys. 1.6 Diagram

<https://medium.com/coding-blocks/creating-user-database-and-adding-access-on-postgresql-8bfcd2f4a91e>alogowanego użytkownika, rola pracownik, schemat E.

Opis wykorzystanych technologii:

Warstwa prezentacji:

Do tworzenia tego poziomu aplikacji wykorzystano React czyli darmową bibliotekę JavaScript służącą do budowy interfejsu użytkownika. Pozwala na tworzenie złożonych interfejsów z mniejszych wydzielonych części kodu zwanych komponentami. W tej technologii każdy komponent posiada stan, którego zmiana powoduje re-renderowanie drzewa Virtual DOM w wyniku czego interfejs aktualizuje się bez konieczności odświeżania strony. Do zarządzania i przesyłania stanu Reacta pomiędzy komponentami wykorzystywany jest Redux.

Wraz z React wykorzystano Bootstrap, bibliotekę CSS, która posiada gotowe style oraz ułatwia tworzenie responsywnych aplikacji przy użyciu siatki (ang. *grid*).

Do komunikacji z warstwą logiki biznesowej wykorzystano asynchroniczne zapytania tworzone przy pomocy biblioteki axios.

Warstwa logiki biznesowej:

Do tworzenia tego poziomu aplikacji wykorzystano język programowania Java oraz framework Spring Boot. Spring jako rozwiązanie działa w oparciu o kontener IoC (ang. *Inversion of Control*) i DI (ang. *Dependency Injection*). Składa się z wielu modułów, a najważniejszy z nich to core container zapewniający najbardziej podstawowe funkcjonalności Springa, w tym narzędzia tworzące kontener IoC i Dependency Injection. W projekcie wykorzystano m.in następujące zależności Springa:

- Spring Boot: Rozwiązanie z gotowymi odgórnie dobranymi modułami pozwalające na natychmiastowe uruchomienie aplikacji (domyślnie uruchamiany przy pomocy Tomcat).
- Spring Boot Starter Data JPA: Paczka narzędzi wykorzystywanych do komunikacji i obsługi bazy danych (mapowanie rezultatów, wysyłanie zapytań, tworzenie repozytoriów, encji).
- Spring Boot Starter Web: Tworzenie endpointów, kontrolerów, serwisów.
- Spring Boot Starter Security: Zabezpieczanie aplikacji przy pomocy np. roli użytkownika.
- Spring Boot Starter Test: Paczka narzędzi wykorzystywanych do testowania aplikacji (mockito, junit, mockmvc).

Warstwa danych:

W stworzonej aplikacji warstwę danych stanowi po prostu lokalnie utworzona relacyjna baza danych PostgreSQL.

Opis istotnych fragmentów kodu:

Autentykacja i Autoryzacja w oparciu o JSON Web Token:

Użytkownik rejestruje się w aplikacji wykorzystując do tego podany w formularzu adres e-mail oraz hasło. Podczas logowania system porównuje podane przez użytkownika dane z tymi przechowywanymi w bazie (hasło jest szyfrowane przy użyciu kodowania bcrypt) i jeśli są poprawne przeglądarka tworzy ciasteczko przechowujące JWT (ang. *JSON Web Token*). JWT składa się z trzech części:

- Header: Zawiera informację o tym jaki algorytm jest wykorzystywany do stworzenia podpisu.
- Payload: Zawiera dane, które programista chce przechowywać w JWT.
- Signature: Podpis tworzony na podstawie Header oraz Payload.

Na rysunku zamieszczono elementy JWT, które wykorzystano w projekcie.



Rys. 1.7 Przedstawienie elementów JWT zawartego w projekcie.

JWT przyznawany jest w części backendowej tuż po zalogowaniu użytkownika, a fragment kodu odpowiedzialny za jego tworzenie pokazano na rys. 1.8. Po utworzeniu token zostaje umieszczony w ciasteczku z nagłówkiem "Authorization" i przedrostkiem "Bearer ". Ciasteczko dodatkowo zabezpieczone jest oznaczeniem HttpOnly, które nie pozwala odczytać go w przeglądarce przez osoby niepowołane.


```
String token = JWT.create()
    .withSubject(
        ((ShelterUserDetails)
            authResult.getPrincipal()).getUsername()
    )
    .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
    .withClaim("role", authResult.getAuthorities().toArray()[0].toString())
    .sign(Algorithm.HMAC512(SECRET.getBytes()));
Cookie cookie = new Cookie(HEADER_STRING, TOKEN_PREFIX + token);
cookie.setHttpOnly(true);
response.addCookie(cookie);
```

Rys. 1.8 Tworzenie JSON Web Token w zaimplementowanej aplikacji.

W aplikacji koniecznie jest sprawdzenie czy dany użytkownik jest zalogowany i jaką posiada rolę (w celu autoryzacji). Dzięki JWT nie trzeba przechowywać ani sesji, ani bezpośrednio wysyłać zapytania do bazy danych o rolę użytkownika. Wszystko to można odczytać z JWT (po stronie backendu). W poniższym fragmencie kodu pokazano jak za pomocą klucza oraz algorytmu, którego używano również podczas tworzenia JWT uzyskać potrzebne informacje.

```
role = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
    .build()
    .verify(token.replace(TOKEN_PREFIX, ""))
    .getClaim("role").asString();
```

Rys. 1.9 Pozyskiwanie informacji o roli użytkownika z JWT.

Do zabezpieczenia restowych endpointów wykorzystano klasę `WebSecurityConfigurer`, dziedziczącą po klasie `WebSecurityConfigurerAdapter` dostarczonej przez moduł Spring Security. Pozwala ona na zabezpieczenie wystawionych adresów URL tak, aby były dostępne tylko dla wyznaczonej grupy użytkowników. W poniższym kodzie (nie jest to pełna konfiguracja) zamieszczono funkcję `configure`, która przedstawia konfigurację Spring Security. Wyróżnić w kodzie można m.in filtry odpowiadające za właśnie autentykację i autoryzację (`JWTAuthenticationFilter` oraz `JWTAuthorizationFilter`) czy CORS.

```
protected void configure(HttpSecurity http) throws Exception
{
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

```

        .and()
        .csrf().disable().addFilterBefore(corsWebFilter(), CsrfFilter.class).
        authorizeRequests().
        antMatchers(HttpMethod.OPTIONS, "/**").permitAll().
        antMatchers(HttpMethod.PUT, "/user/changePassword").hasAuthority(USER).
        antMatchers(HttpMethod.POST, "/interest").hasAuthority(USER).
        antMatchers(HttpMethod.DELETE, "/interest/{id}").hasAuthority(USER).
        antMatchers(HttpMethod.POST, "/animal").hasAuthority(EMPLOYEE).
        antMatchers(HttpMethod.DELETE, "/animal/{id}").hasAuthority(EMPLOYEE).
        antMatchers(HttpMethod.GET, "/breed").hasAuthority(EMPLOYEE).
        antMatchers(HttpMethod.GET, "/size").hasAuthority(EMPLOYEE).
        antMatchers("/user/{id}").hasAuthority(ADMIN).
        antMatchers("/user/**").hasAuthority(ADMIN).
        anyRequest().authenticated().
        and().
        httpBasic().authenticationEntryPoint(shelterAuthenticationEntryPoint()).
        and().
        logout().logoutSuccessHandler(onSuccessfulLogoutHandler())
        .and().
        addFilter(new JWTAuthenticationFilter(authenticationManager()))
        .addFilter(new JWTAuthorizationFilter(authenticationManager()));
    }

```

Rys. 1.10 Zabezpieczenie adresów URL przy wykorzystaniu funkcji `configure` z *Spring Security*.

Tworzenie repozytoriów oraz encji:

Mapowanie rezultatów zapytań z bazy danych do obiektów Javy odbywa się przy pomocy Hibernate. Do tego potrzebne jest stworzenie encji reprezentującej tabelę z bazy danych oraz repozytorium w którym możemy skonfigurować odpowiednie zapytania. Na rys. 1.11-1.12 przedstawiono przykład rozwiązania problemu dla najprostszej tabeli dotyczącej rozmiaru zwierzaków. Dodatkowo warto zwrócić uwagę, że można wykorzystać gotowe zapytania przygotowane przez hibernate bez konstruowania ich ręcznie (w tym przypadku zastosowano jednak `nativeQuery`).

```

@Repository
public interface SizeRepo extends JpaRepository<Size, Integer>
{
    @Query(value = "SELECT * FROM sizes", nativeQuery = true)
    public List<Size> getAllSizes();
}

```

Rys. 1.11 Repozytorium obsługujące tabelę `Size` przechowującą rozmiary zwierzaków.

```

@Entity

```

```

@Table(name = "sizes")
public class Size
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_size")
    private int idSize;

    @Column(name = "name")
    private String name;
}

```

Rys. 1.12 Encja reprezentująca tabelę *Size* przechowującą rozmiary zwierzków.

Tworzenie kontrolerów:

Do wystawiania restowych endpointów służą kontrolery z adnotacją *@RestController* oraz *@RequestMapping* (deklaracja adresu url). Poniżej zamieszczono prosty przykład dla obsługi wielkości zwierzków. Adnotacja *@GetMapping* określa metodę HTTP wykorzystywaną w zapytaniu, a *ResponseEntity* zwraca odpowiedni status HTTP oraz body w formacie JSON. Cała logika komunikacji z bazą oraz procesowanie zawarte jest w wstrzykniętym serwisie *sizeService*.

```

@RestController
@RequestMapping("/size")
public class SizeController
{
    @Autowired
    SizeService sizeService;

    @GetMapping
    public ResponseEntity<?> getAllSizes()
    {
        return ResponseEntity.ok(sizeService.getAllSizes());
    }
}

```

Rys. 1.13 Zaimplementowany kontroler do obsługi endpointów związanych z wielkością zwierzaka.

Obsługa błędów restowych:

W aplikacji wykorzystano globalny handler wyjątków rzucanych przez kontrolery. Przygotowany handler wykorzystuje adnotację *@RestControllerAdvice* oraz dziedziczy po *ResponseEntityExceptionHandler*. Poniżej zamieszczono fragment utworzonego *GlobalExceptionHandler*, który w przypadku wyrzucenia przez aplikację wyjątku

SentDataIsNullException zwróci odpowiednią informację do części frontendowej (w tym wypadku informację o pustych przesyłanych danych).

```
@RestControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler
{
    @ExceptionHandler(value = SentDataIsNullException.class)
    public ResponseEntity<?> sentDataIsNullHandler(SentDataIsNullException
exception)
    {
        return ResponseEntity.status(exception.getHttpStatus())
            .body(new ErrorResponse(exception.getMessage(),
                exception.getHttpStatus()));
    }
}
```

Rys. 1.14 Zarządzanie wyjątkami w aplikacji poprzez wykorzystanie *GlobalExceptionHandler*.

Tworzenie testów restowych:

Do tworzenia testów po stronie backendu wykorzystano JUnit, Mockito raz MockMvc. Poniżej zamieszczono przykładowy test dla endpointu */animal/{id}*, który sprawdza czy wyjątek *DataDoesNotExistException* zostaje prawidłowo obsłużony i zwraca odpowiedni status.

```
@Test
public void getAnimalByIdTestIfDataDoesNotExist() throws Exception
{
    when(animalService.getAnimalById(idAnimal: 999))
        .thenThrow(new DataDoesNotExistException("Such data does not exist", 404));
    mockMvc.perform(get(urlTemplate: "/animal/{id}", ...uriVars: 999).contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().is(status: 404)).
        andExpect(r -> assertTrue(r.getResolvedException() instanceof DataDoesNotExistException)).
        andDo(print()).
        andReturn();
}
```

Rys. 1.15 Test sprawdzający działanie endpointu */animal/{id}* podczas wyrzucenia wyjątku *DataDoesNotExistException*.

Ustawienia routingu

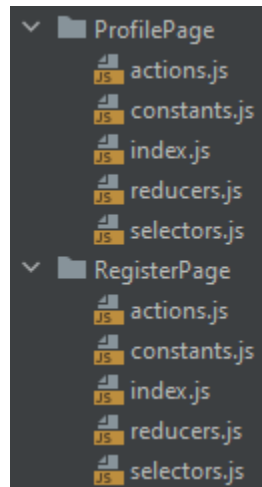
W projekcie zainstalowana została biblioteka *react-router-dom*. Pozwala ona na ustawienie routingu. W aplikacji dostępne jest kilka adresów. W zależności od roli użytkownika ma on dostęp tylko do odpowiednich ścieżek. Aplikacja zostaje uruchomiona pod adresem `http://localhost:3000`. Tam znajduje się strona główna. Podanie ścieżki w adresie nie pasującej do żadnej z poniższych spowoduje wyświetlenie komunikatu o braku takiego url: *404 Not Found!*. Pod każdym adresem kryje się osobny komponent, który wyświetla odpowiednią zawartość strony.

```
function App() {  
  return (  
    <div className="App">  
      <Router>  
        <Switch>  
          <Route exact path="/" component={HomePage}/>  
          <Route exact path="/animal/:id" component={AnimalPage}/>  
          <Route exact path="/login" component={LoginPage}/>  
          <Route exact path="/registration" component={RegisterPage}/>  
          <Route exact path="/profile" component={ProfilePage}/>  
          <Route exact path="/admin" component={AdminPage}/>  
          <Route exact path="/employee" component={EmployeePage}/>  
          <Route>404 Not Found!</Route>  
        </Switch>  
      </Router>  
    </div>  
  );  
}
```

Rys. 1.16 Główna funkcja *App()* z wypisanymi dostępnymi ścieżkami

React Redux

Każdy z komponentów został rozbity na 5 plików:



Rys. 1.17 Komponenty wraz z plikami

W pliku `index.js` znajduje się klasa rozszerzająca klasę `Component`. Renderuje ona wygląd poszczególnych stron za pomocą metody `render()`. W pliku tym znajdują się również dostępne dla danej klasy metody:

```
const mapDispatchToProps = {
  getAnimals,
  isUserLoggedIn,
  getUserInterests,
  checkIfUser,
  deleteLike,
  logout,
  getUserEmail,
  changePassword,
  setError,
};
```

Rys. 1.18 Metody dostępne w klasy przy użyciu `props`

Metody te są dostępne w ciele klasy przy użyciu `props`-ów, dzięki Jest to jedna z mechanik działania biblioteki `react-redux`.

W pliku `actions.js` umieszczone są funkcje do komunikacji z serwerem oraz funkcje do zmiany stanu danej zmiennej:

```

export function setUserInterests(ids) {
  return {
    type: ActionTypes.SET_USER_INTERESTS,
    ids
  }
}

export const getAnimals = (isEmployee, isLoggedIn) => (dispatch) => {
  axios.get( url: "http://localhost:8080/animal")
    .then((response : AxiosResponse<any> ) => {
      dispatch(setAnimals(response.data));
      dispatch(getAnimalsLikes(response.data));
      if(!isEmployee && isLoggedIn) {
        dispatch(getUserInterests());
      }
    })
    .catch((e) => {
      alert("Nie możemy w tej chwili pokazać Ci naszych zwierząt. Przepraszamy.")
    });
};

```

Rys. 1.19 Przykład funkcji do zmiany stanu aplikacji oraz komunikacji z serwerem.

Do funkcji zmiany stanu przekazywany jest typ - rodzaj operacji do wykonania oraz nowa wartość stanu. Typy umieszczone są w pliku *constants.js*:

```

export const ActionTypes = {
  SET_ANIMALS: "src/containers/HomePage/SET_ANIMALS",
  SET_RECENT_ANIMALS: "src/containers/HomePage/SET_RECENT_ANIMALS",
  SET_LIKES: "src/containers/HomePage/SET_LIKES",
  SET_USER_INTERESTS: "src/containers/HomePage/SET_USER_INTERESTS",
};

```

Rys. 1.20 Przykładowa zawartość pliku *constants.js*.

Po wywołaniu funkcji zmiany stanu następuje przekierowanie do funkcji odpowiedzialnej za przypisanie nowej wartości do zmiennej. Funkcja ta zwana jest również *reducerem* i znajduje się w pliku *reducers.js*.

```

const defaultState = {
  animals: [],
  recentAnimals: [],
  likes: {},
  userInterests: [],
};

export default function homePageReducer(state : {...} = defaultState, action) {
  switch (action.type) {
    case ActionTypes.SET_ANIMALS:
      return {
        ...state,
        animals: action.animals
      };
  }
}

```

Rys. 1.21 Przykładowa zawartość pliku *reducers.js*.

Przy tworzeniu się komponentu posiada on stany o wartościach domyślnych zdefiniowanych wyżej jako *defaultState*. Wywołanie metody do zmiany stanu z odpowiednim typem powoduje wykonanie się wskazanego bloku *case*, w którym przypisywany jest nowy stan danej zmiennej. Komponent ma dostęp do danych zasobów zawartych w selektorze znajdującym się w pliku *selectors.js*.

```

import {createSelector} from "reselect"

const homePageState = (state) => state.homePage;
const loginPageState = (state) => state.loginPage;

export const makeSelectResponse = createSelector(
  homePageState,
  loginPageState,
  {
    combiner: (homePage, loginPage) => ({...homePage, ...loginPage})
  }
);

```

Rys. 1.22 Przykładowa zawartość pliku *selectors.js*.

homePage oraz *loginPage* odpowiadają odpowiednim *reducer-om* w plikach *reducers.js*. Zmienne zawarte w tych plikach umieszczane są w jednym miejscu - w *store*. Przechowuje on wszystkie stany dostępne w *reducer-ach*.


```
const reducers = combineReducers( reducers: {homePage, loginPage, animalPage, registerPage, adminPage, employeePage, profilePage});
export default createStore(reducers, applyMiddleware(thunk));
```

Rys. 1.23 Tworzenie store dla wszystkich reducer-ów.

Komponenty mogą korzystać ze store dzięki funkcji *connect* z biblioteki *react-redux*.

```
export default connect(makeSelectResponse, mapDispatchToProps)(LoginPage);
```

Rys. 1.24 Export komponentu *LoginPage* z użyciem metody *connect*

Oprócz klas będącymi głównymi “kontenerami” aplikacji w projekcie znajdują się również mniejsze komponenty, z których składają się te główne.

Tworzenie testów frontendowych

Do tworzenia testów po stronie frontendu użyto Reactowej biblioteki do testów *@testing-library/react*. Poniżej znajduje się przykładowy test sprawdzający jaką klasę będzie miał kontener w zależności od podanych wcześniej wartości:

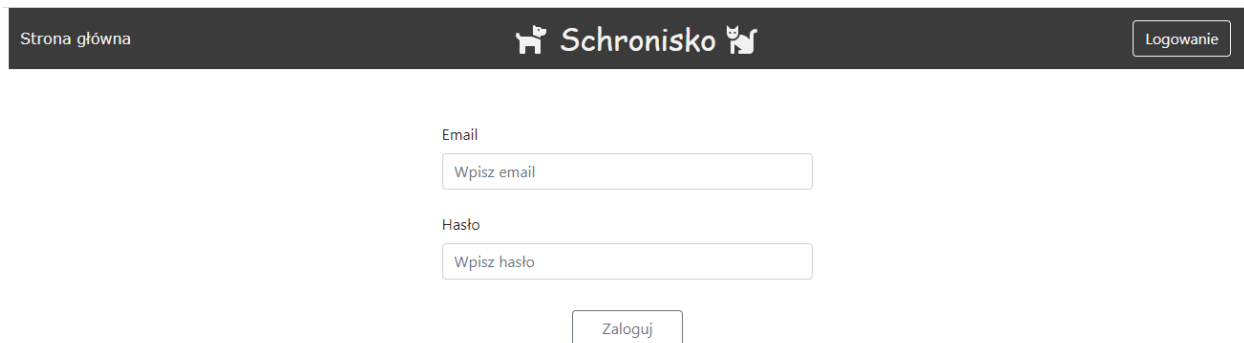
```
it( name: "test interests counter color for interested user", fn: () => {
  render(<AnimalCard isInterested={true} animal={[]}/>)
  const container = document.getElementById( elementId: "animalCardCounter");
  expect(container.className).toBe( expected: "interestCounter blueish");
})
```

Rys. 1.25 Test sprawdzający poprawność koloru kontenera, gdy użytkownik jest zainteresowany danymi zwierzęciem

Przedstawienie funkcjonalności aplikacji w postaci zrzutów ekranu:



Rys. 1.27 Widok strony głównej niezalogowanego użytkownika



Rys. 1.28 Widok logowania

Email

Hasło

Powtórz hasło

Zarejestruj się

Rys. 1.29 Widok rejestracji

Email

Hasło

Powtórz hasło

Hasła nie są identyczne

Zarejestruj się

Rys. 1.30 Widok rejestracji przy niepoprawnie podanych hasłach.

Komunikat ze strony localhost:3000

Rejestracja przebiegła pomyślnie. Możesz się teraz zalogować

OK

Email

user3@cc.pl

Hasło

.....

Powtórz hasło

.....

Zarejestruj się

Rys. 1.31 *Pomyślna rejestracja użytkownika.*

Email

user@email.com

Hasło

.....

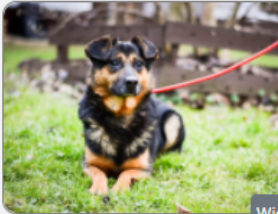
Zły login lub hasło

Zaloguj

[Nie posiadasz konta? Zarejestruj się!](#)

Rys. 1.32 *Logowanie z niepoprawnie podanymi danymi.*

Nasze zwierzaki:




Kordek 1 🐾

Płochliwy psiak, który dopiero w schronisku uczy się jak powinny wyglądać relacje z ludźmi...

Nie jestem już zainteresowany

Więcej




Rudi 1 🐾

Najlepiej będzie się czuł w domu z ogrodem. Preferowany dom bez dzieci. Pewny siebie...

Nie jestem już zainteresowany

Więcej




Hieronim 1 🐾

Bardzo przyjaźnie nastawiony do ludzi psiak, energiczny (ale nie nadpobudliwy). Jest nieco...

Jesteś zainteresowany? Daj znać innym! Kliknij!

Więcej



Sierpień 0 🐾

Przyjęto go do schroniska 1.08.2018 r. po wypadku. Niestety, obrażenia były bard...


Jesteś zainteresowany? Daj znać innym! Kliknij!

Więcej

Rys. 1.33 Wyświetlenie zwierzaków przez zalogowanego użytkownika.

Strona główna Schronisko Profile Wyloguj

Rudi



Data przybycia: 2018-05-02
Rasa: Mieszaniec
Rozmiar: średni

1 🐾

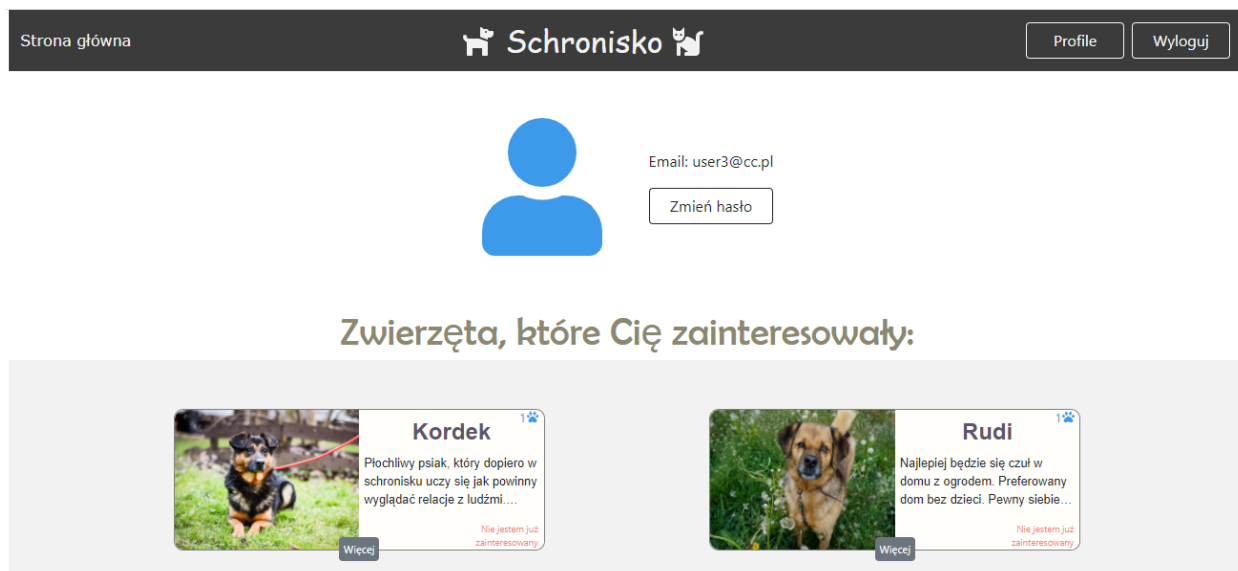
Najlepiej będzie się czuł w domu z ogrodem. Preferowany dom bez dzieci. Pewny siebie, energiczny. Potrzebuje doświadczonego opiekuna. Sprawdzi się w psich sportach.

Nie jestem już zainteresowany

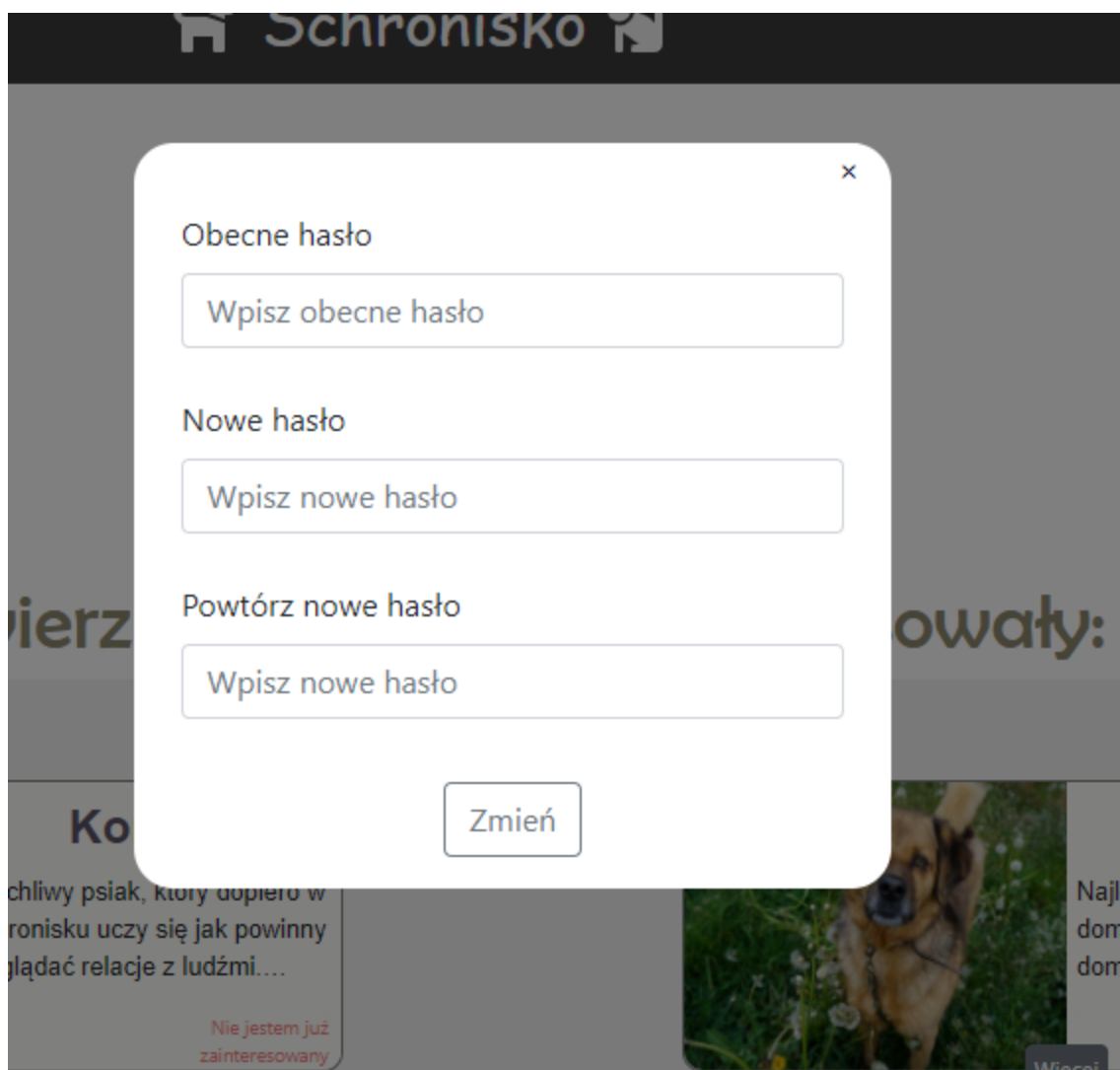
Rys. 1.34 Wyświetlenie profilu konkretnego zwierzaka.



Rys. 1.35 Wyświetlenie osób zainteresowanych zwierzęciem po najechaniu na licznik w profilu zwierzęka.




Rys. 1.36 Wyświetlenie profilu zalogowanego użytkownika.



Rys. 1.37 Wyświetlenie pop-up ze zmianą hasła użytkownika w jego profilu.

Formularz zwierzaka

Imię:

Data: 













Rozmiar:

Rasa:

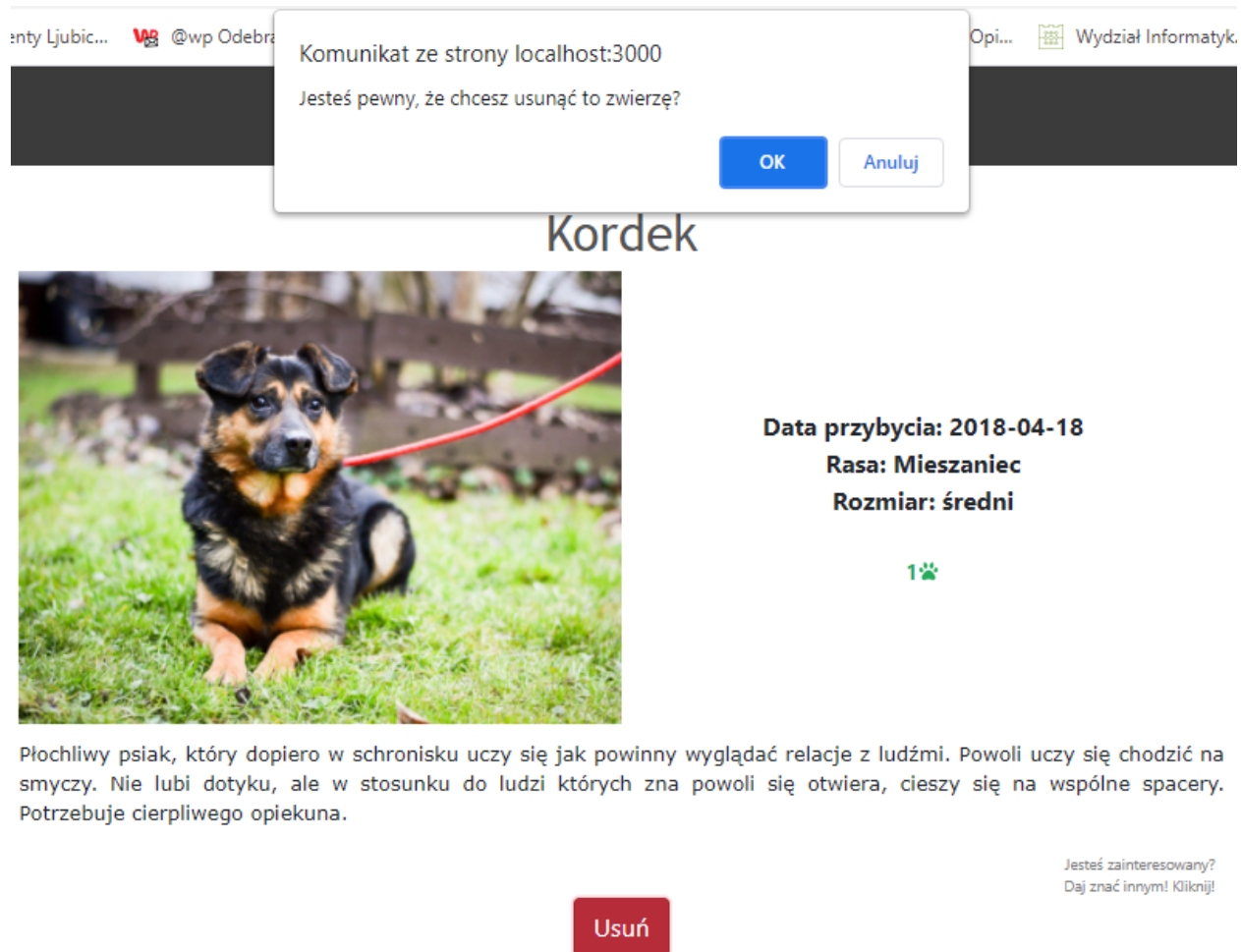
Opis:

Opis zwierzaka

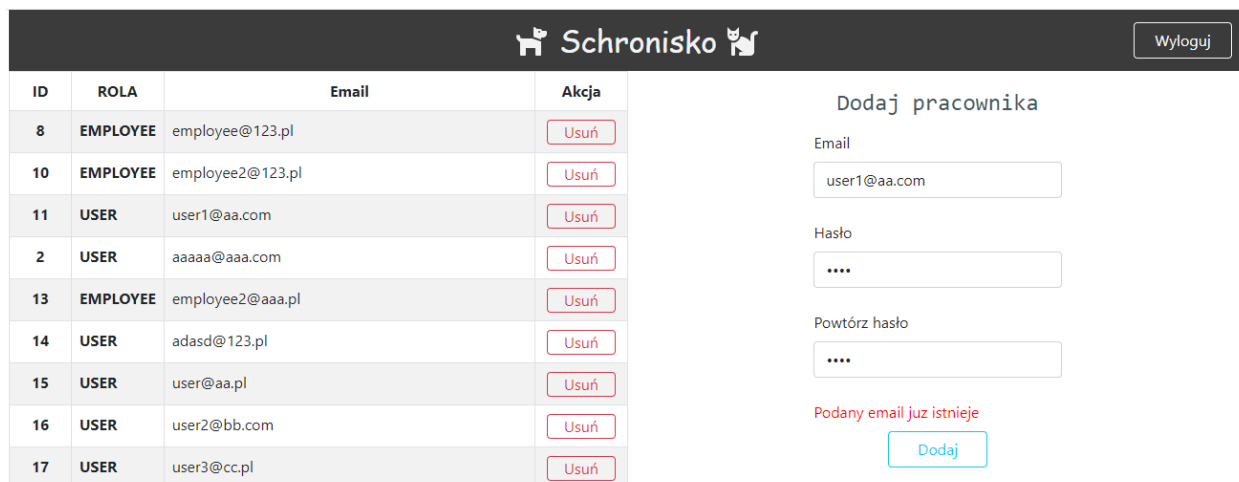
Nasze zwierzaki:

 <p>Goran 1 </p> <p>Ten pies to wulkan energii, świetnie aportuje – może godzinami biegać za piłką....</p> <p>Jesteś zainteresowany? Daj znać innym! Kliknij!</p> <p><input type="button" value="Więcej"/></p>	 <p>Kordek 1 </p> <p>Płochliwy psiak, który dopiero w schronisku uczy się jak powinny wyglądać relacje z ludźmi....</p> <p>Jesteś zainteresowany? Daj znać innym! Kliknij!</p> <p><input type="button" value="Więcej"/></p>	 <p>Rudi 1 </p> <p>Najlepiej będzie się czuł w domu z ogrodem. Preferowany dom bez dzieci. Pewny siebie....</p> <p>Jesteś zainteresowany? Daj znać innym! Kliknij!</p> <p><input type="button" value="Więcej"/></p>
 <p>Lary 0 </p> <p>Średniej wielkości uroczy</p> <p><input type="button" value="Więcej"/></p>	 <p>Hieronim 1 </p> <p>Bardzo przyjaźnie nastawiony</p> <p><input type="button" value="Więcej"/></p>	 <p>Sierpień 0 </p> <p>Przyjeżdża do schroniska</p> <p><input type="button" value="Więcej"/></p>

Rys. 1.38 Wyświetlenie profilu pracownika.



Rys. 1.39 Usuwanie zwierzęcia z bazy dostępnych do adopcji zwierząt.



Rys. 1.40 Panel admina z możliwością usuwania użytkowników oraz dodawania pracowników.