Autorzy: Lena Zubik i Kacper Nowakowski

Dokumentacja etap IV

Opis

Temat projektu: Wspomaganie zarządzania ZOO

Zaprojektowana przez nas aplikacja ma na celu usprawnienie działalności, jaką jest zarządzanie placówką ZOO. W ramach aplikacji przewidujemy możliwość logowania się użytkownika/pracownika w celu sprawdzenia grafiku zadań i obowiązków. Wszelkie informacje o inwentarzu oraz dane użytkowników będą przechowywane w bazie danych.

Przykładowe funkcje

- możliwość wyświetlenia grafiku zadań dla pracownika
- wyświetlanie informacji o danym gatunku (np. opis, gdzie ma wybieg, karmienie)
- wprowadzanie zmian w systemie przez pracownika
- dodawanie zadań dla pracownika przez admina

Architektura aplikacji

Aplikacja została podzielona na trzy główne warstwy:

- Warstwę persystencji
- Warstwę logiki
- Warstwę prezentacji

Warstwa persystencji

Warstwa persystencji jest to warstwa zajmująca się dostępem do danych. Wybrana przez nas baza danych to relacyjna baza MySQL zainstalowana lokalnie. W bazie znajdują się takie encje jak: User, ZooKeeper, Task, Species, Animal, które za pomocą narzędzia Hibernate korzystającego ze standardu JPA są odwzorowaniem obiektowo-relacyjnym (ORM) odpowiednich encji utworzonych w kodzie w pakiecie entity. Implementacją tej warstwy w kodzie jest zbiór repozytoriów znajdujących się w pakiecie repository. Zajmują się one przetwarzaniem danych pod kątem zapisu ich do bazy danych.

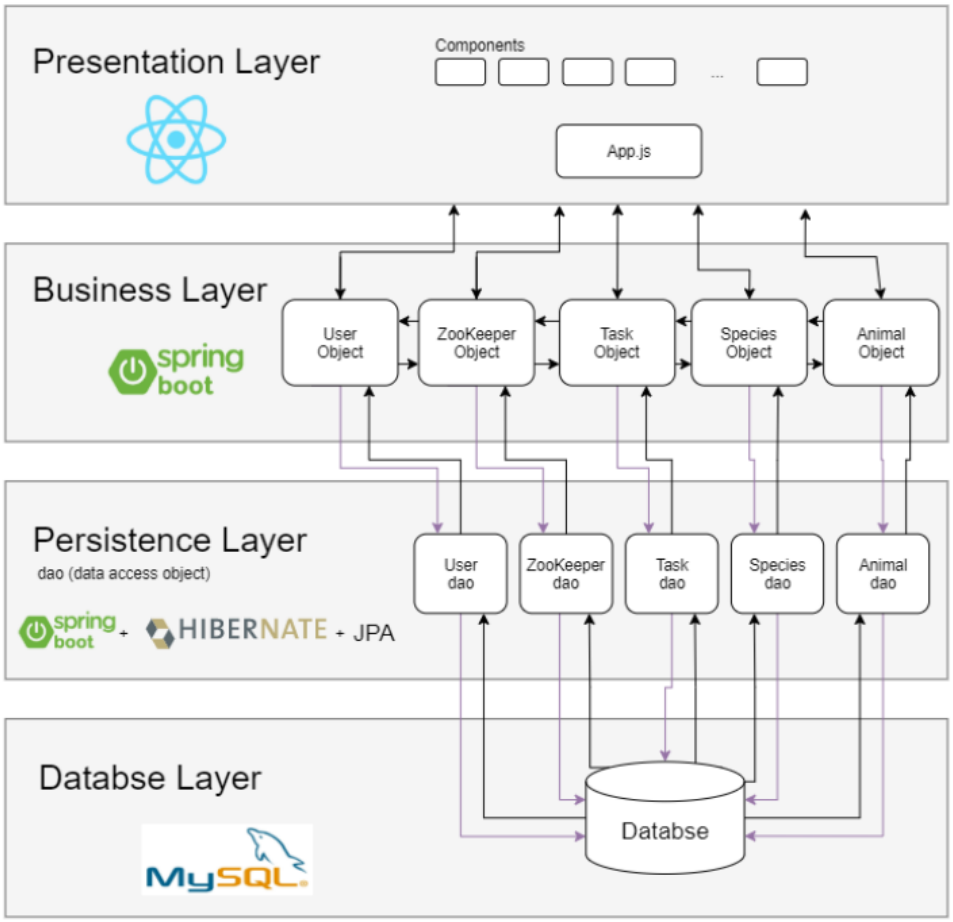
Warstwa logiki

Warstwa logiki aplikacji składa się na całą funkcjonalność aplikacji. Znajdują się w niej obiekty biznesowe - encje - jak i serwisy zdefiniowane dla każdego z nich w pakiecie service. Są to obiekty, które udostępniają operację w danej dziedzinie biznesowej. Jedną do tej pory zrealizowaną ważną funkcją to sprawdzanie unikalności nazw w tabeli niektórych encji np. pilnowanie, aby nie dodać do tabeli przetrzymującej gatunki (Species) takiego gatunku, który w niej już jest.

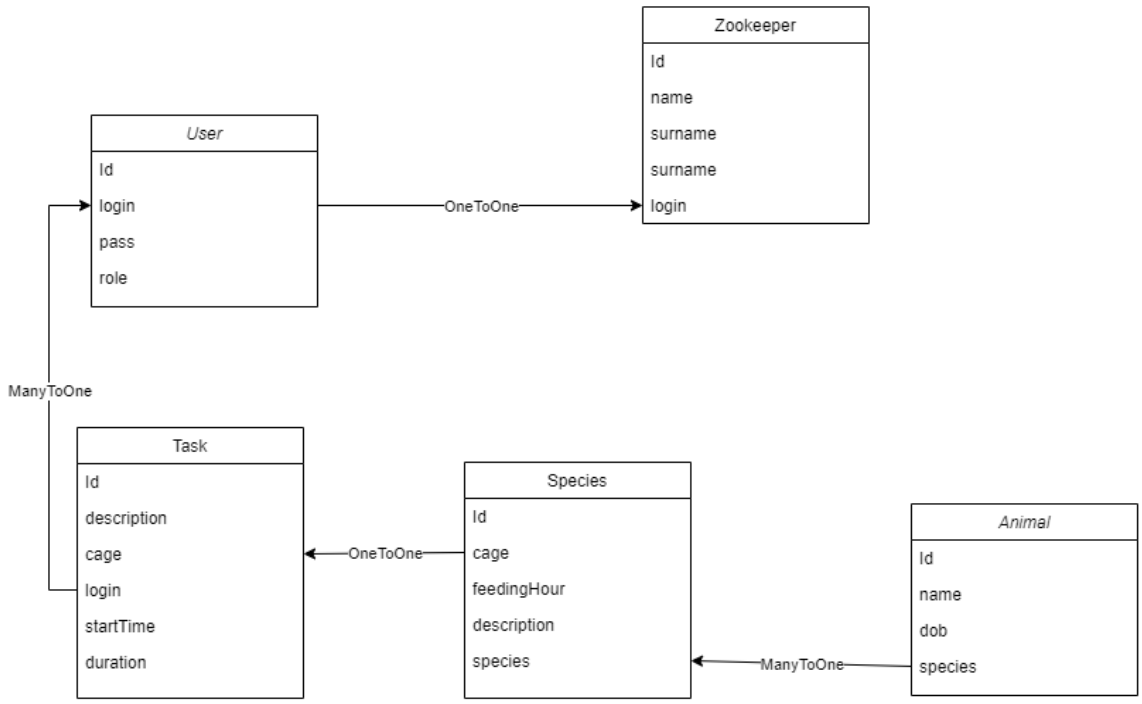
Warstwa prezentacji

Zadaniem warstwy prezentacji jest wystawianie danych i przetwarzanie danych wejściowych. Za tę warstwę odpowiadają zdefiniowane dla każdego obiektu biznesowego kontrolery znajdujące się w pakiecie controller. Do wystawiania i pobierania danych wejściowych użyliśmy REST API (na porcie 8080), dzięki temu w prosty sposób jesteśmy w stanie je zaprezentować na interfejsie użytkownika stworzonym przy pomocy JavaScript z wykorzystaniem biblioteki React.

Stos technologii użytych przy tworzeniu aplikacji



Baza Danych



Stworzona przez nas aplikacja webowa bazuje głównie na CRUD – funkcjach odczytu, wpisania, zaktualizowania oraz usunięcia elementu – bazy danych, w której znajdują się informacje potrzebne do zarządzania instytucją ZOO. Aby umożliwić i ułatwić użytkownikowi wymienione operacje, obiekty, którymi się zajmuje zostały podzielone na 5 encji:

User – obiekty tej encji są odpowiednikami kont wszystkich użytkowników, zawierają w sobie informację o loginie, haśle i roli użytkownika. Na jej podstawie walidowane są dane podczas logowania się pracownika. Należy zaznaczyć, że hasło nie jest jawnie przechowywane w bazie. Jest zakodowanie przy użyciu funkcji BCryptPasswordEncoder.

Zookeeper – obiekty tej encji odpowiadają obiektom pracowników przynależnych do kont stworzonych na podstawie klasy User. W tej encji znajdują się takie pola jak: imię, nazwisko i login. Encja User i Zookeeper są ze sobą połączone (istnieje pomiędzy nimi relacja) relacją OneToOne. Jedno konto w systemie przynależy do jednego pracownika, jest to sprawdzane dla wartości unikalnych pól jakimi są loginy. Warto dodać, że użytkownikowi z poziomu interfejsu nie można usunąć danych o pracowniku przynależącym do danego konta, natomiast przy usunięciu konta informacje o jego posiadaczu także są usuwane z tabeli Zookeeper.

Task – encja przedstawiająca zadanie. Zawiera pola takie jak: opis, klatkę, której dotyczy to zadanie, login pracownika, który się zajmie tym zadaniem, czas rozpoczęcia zadania oraz czas trwania. Jest to encja szczególnie przydatna dla użytkowników posiadających konta z rolą ADMIN. To właśnie oni przydzielają zadania pracownikom (zadanie może zostać przydzielone tylko pracownikami z kontem z rolą USER). Encja ta posiada dwie relacje ManyToOne z encją User (każdy użytkownik z rolą USER może mieć przypisanych wiele zadań) i OneToOne z encją Species (każde zadanie może dotyczyć dokładnie jednego gatunku).

Species – encja symbolizująca gatunek, zawierająca informacje o klatce, w której dany gatunek przebywa, godzinę karmienia, opis gatunku, oraz nazwę gatunku.

Animal – encja symbolizująca poszczególne zwierzęta przebywające w ZOO. Zawiera informacje takie jak: imię zwierzaka, data urodzenia i nazwa gatunku, jakiemu podlega. Encje Species i Animal są połączone relacją OneToMany (wiele zwierząt o tym samym gatunku podlega encji Species).

Backend

Backend stworzonej aplikacji webowej opiera się na modelu warstwowym aplikacji. Encje w bazie danych tworzone są poprzez mapowanie obiektów pięciu głównych klas (przy użyciu JPA): Animal, Species, Task, User, Zookeeper. Do każdej z tych klas zostało utworzone repozytorium, zapewniające komunikację z bazą danych, serwis, odpowiadający za działania na danych pobieranych i wysyłanych, np. walidacja oraz kontroler, którego zadaniem jest komunikacja z frontendem (poprzez REST API).

Implementacja logiki aplikacji

Logika aplikacji została zaimplementowana w serwisach stworzonych encji. Elementami tej logiki są walidacja danych wpisywanych do bazy danych oraz przesyłanie ewentualnych widomości o niepowodzeniu wykonywanej operacji do interfejsu użytkownika. Ważną funkcjonalnością, którą zapewnia aplikacja, jest pilnowanie dostępności opiekuna podczas przyznawania mu zadań przez admina. Gdy do serwisu trafi zapytanie o przyznanie dostępnemu użytkownikowi jeszcze jednego zadania w tym czasie, serwis zwróci błąd o braku możliwości zrealizowania takiego żądania i prześle listę użytkowników (w skrajnych wypadkach pustą), którzy nie są zajęci w tym czasie. Koleją wartą uwagi funkcjonalnością, którą dostarcza aplikacja jest (przy włączonej o danej godzinie aplikacji, w rzeczywistości chodziłaby ona cały czas) tworzenie nieprzypisanych nikomu zadań karmienia dostępnych gatunków zwierząt. Za pomocą funkcji systemowej cron o wyznaczonej godzinie jest wysyłane żądanie o stworzenie zadań w bazie danych.

Testy

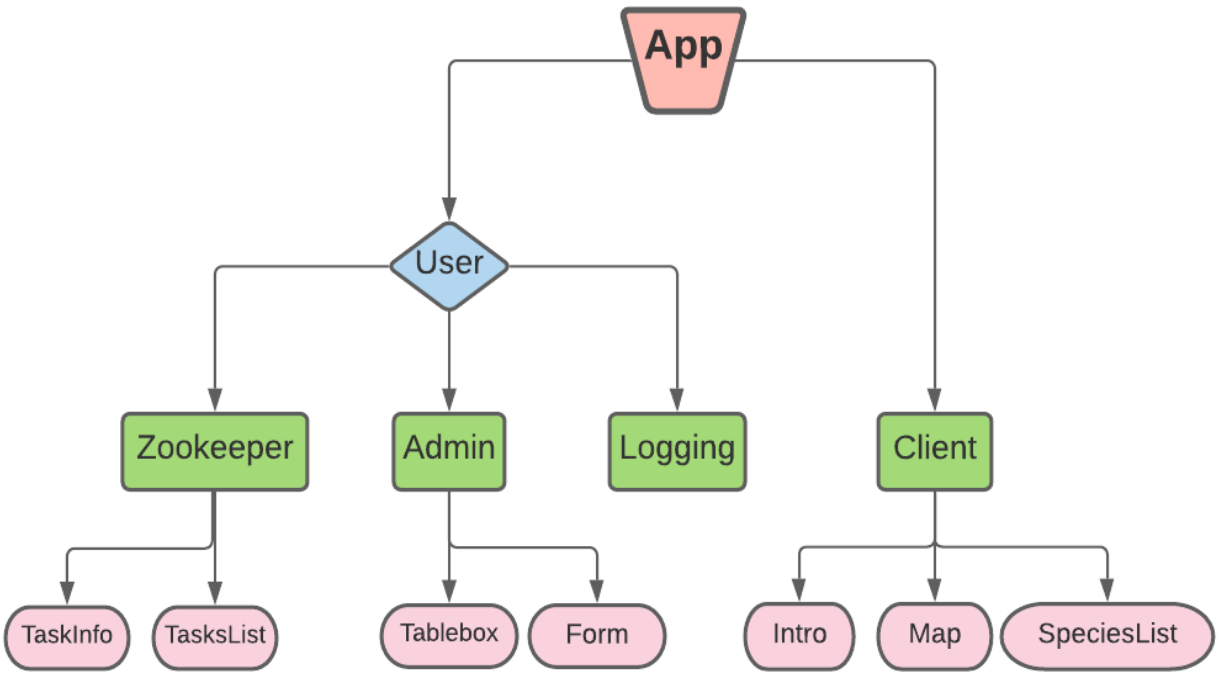
Do poprawnego funkcjonowania aplikacji niepotrzebne było tworzenie własnych zapytań, ze względu na użycie framework'a Spring Boot zapytania te zostały stworzone automatyczne, a co za tym idzie nie wymagają one ponownego testowania. Z tego powodu nie zostały wykonane testy dla metod dostępnych w repozytoriach. Ze względu na skrupulatną walidację i elementy logiki na poziomie serwisu, zostały wykonane testy właśnie tej warstwy aplikacji. Znajdują się one w katalogu test/service. Testy te nie są zwykłymi testami jednostkowymi, dla niektórych metod sprawdzenie poprawności działania odbyło się za pomocą mock testów, czyli testów jednostkowych z wykorzystaniem framework'a Mockito. Dzięki temu można

przyjąć, że poprawnie działające repozytoria są ‘mock’ (nie trzeba ich definiować) i za pomocą funkcji given() sprawdzać poszczególne zachowania dla danych wejściowych i na końcu przyrównywać wyjście do oczekiwanego stanu lub za pomocą funkcji verify() sprawdzać, czy oczekiwana przez nas na wstępie metoda została wywołana. Zgodnie z uzyskaną informacją na konsultacjach testom podlegają AnimalService i UserService, testowanie reszty serwisów jeszcze nie zostało zakończone, natomiast pewne testy już zostały przeprowadzone.

Frontend

W projektowaniu frontendu skorzystaliśmy ze wzorca Single Page Application. Oznacza to, że wszystkie pliki HTML, CSS oraz JavaScript zostają pobierane przy pierwszym uruchomieniu strony. Potem interfejs łączy się z serwerem tylko jeśli potrzebuje pobrać informacje przechowywane w bazie danych. Wszystkie zmiany wynikające z akcji użytkownika odbywają się dynamicznie. Odpowiada za to biblioteka React a dokładniej klasy utworzone z jej pomocą.

Drzewko i krótki opis klas



W powyższym diagramie przyjętą następującą konwencję:

Róż – są to klasy, które odpowiadają za renderowanie i funkcjonalność poszczególnych modułów, fragmentów strony.

Zieleń – klasy te odpowiadają za zawartość głównej części strony. Odpowiednio rozstawiają moduły zwracane przez różowe klasy a także zapewniają komunikację między nimi.

Błękit – klasy te decydują tylko o tym, która zielona klasa w danym momencie ma być wyświetlana użytkownikowi.

Czerwień – jest to główna klasa która sama z siebie generuje nagłówek. Ona też na podstawie wybranej przez użytkownika zakładki decyduje skąd będzie pobierać zawartość głównej części strony.

Strzałki – mówią one o tym między którymi klasami zachodzi komunikacja a także pobieranie zwracanych elementów.

App – Główna klasa aplikacji. Odpowiada ona za wyświetlanie nagłówka z logo, przyciskami do wyboru strony jak i zegara. W zależności od wybranej strony wyświetla na jej głównej części to co zwróciły klasy ***User*** bądź ***Client***.

User – Klasa ta sama w sobie nic nie prezentuje. Jej zadaniem jest wybór odpowiedniej zawartości do wyświetlenia w zależności od tego czy użytkownik się już zalogował a jeśli tak to jako admin czy jako opiekun.

Zookeeper – Odpowiada za obsługę konta opiekuna. To ona wysyła zapytanie do serwera o listę zadań przydzielonych dla zalogowanego użytkownika i segreguje je w zależności od pozostałego czasu. Każde z tych zleceń wysyła do klasy ***TasksList***, która wyświetla je jako kolejne kafelki. Jeśli, któryś z tych kafelków zostanie kliknięty informacja ta wysyłana jest do klasy ***Zookeeper***, która wysyła obiekt wybranego zadania do klasy ***TaskInfo***, która to wyświetla szczegóły zlecenia, pozostały czas i pozwala je zakończyć.

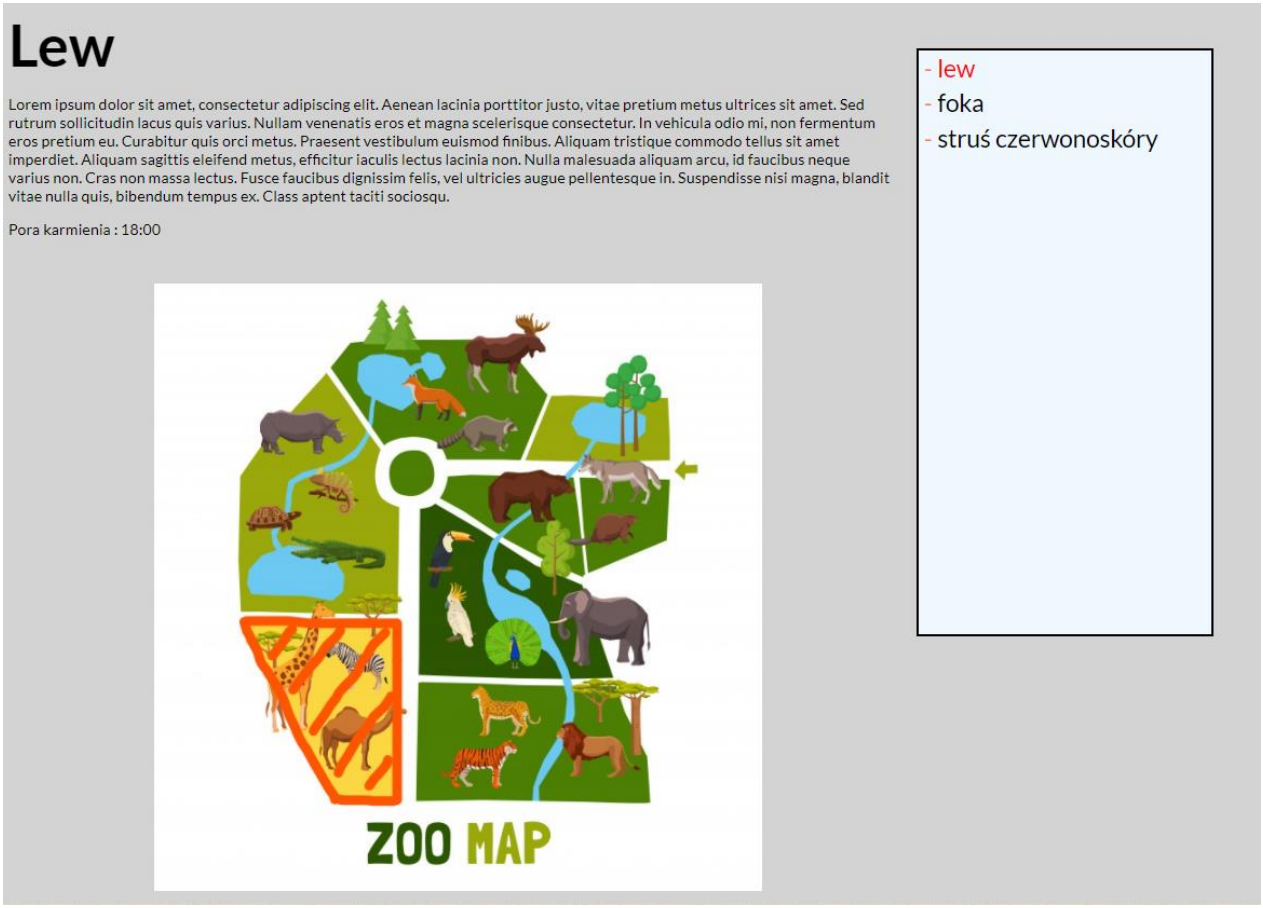
Admin – Odpowiada za obsługę konta administratora. Wyświetla ona 5 przycisków z dostępnymi kategoriami jak „zwierzęta” czy „opiekuni”. Po kliknięciu w jeden z nich pobiera ona z serwera listę obiektów należących do danej kategorii i wysyła je do klasy ***Tablebox***, która wyświetla je odpowiednio w nowo stworzonej tabeli. Pozwala ona dodanie obiektu wybranej kategorii albo usunąć lub edytować wybrany obiekt. Na podstawie wybranej akcji odpowiednie dane są wysyłane do klasy ***Form***, która wyświetla formularz do uzupełnienia. Sprawdza ona również czy wprowadzone dane są prawidłowe i wysyła odpowiednie zapytanie do serwera po zatwierdzeniu formularza.

Logging – Wyświetla typowy formularz potrzebny do zalogowania. Po zatwierdzeniu wysyła wprowadzone dane do serwera i czeka na odpowiedź czy są one poprawne. Jeśli nie - wyświetla stosowną informację. Jeśli tak – powiadamia o tym klasę ***User***.

Client – Odpowiada za obsługę strony dla zwiedzającego. Pobiera ona listę wszystkich gatunków z serwera i wysyła je do klasy ***SpeciesList*** po to, by wyświetlić je jako wybieralną listę. Początkowo klasa ***Intro*** wyświetla informację ogólną jednakże po wybraniu gatunku przedstawia jego opis i porę karmienia. Klasa ***Map*** domyślnie przedstawia mapę całego Zoo ale po wybraniu gatunku wyświetla mapę z zaznaczonym sektorem gdzie można znaleźć dane zwierzęta.

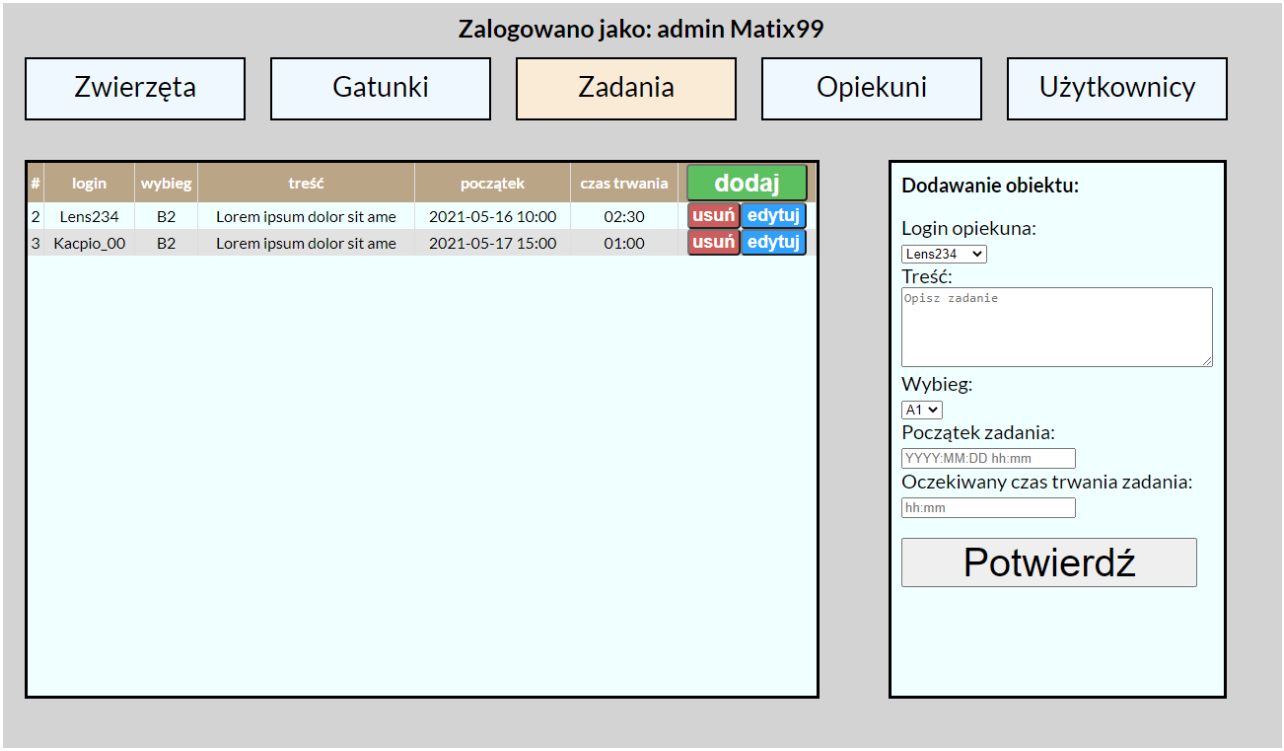
Reprezentacja graficzna

Interfejs uruchamia się na stronie „dla odwiedzających” bez wybranego gatunku

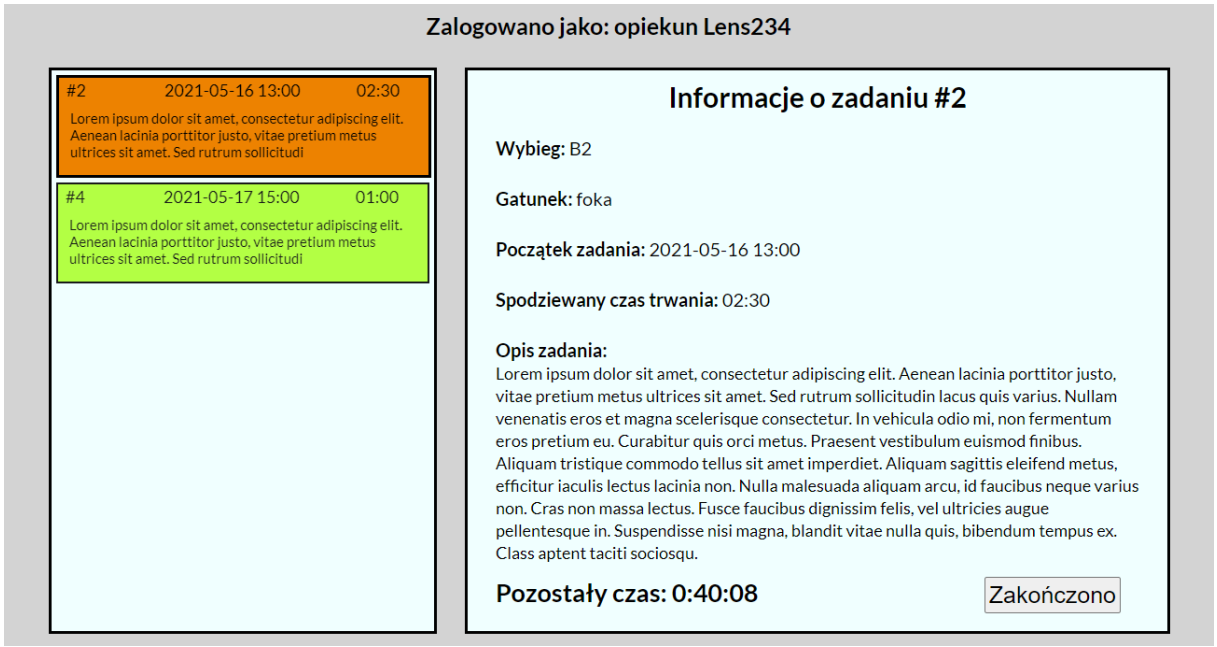


Po wybraniu gatunku zamiast ogólnego wpisu pojawia się opis dotyczący gatunku jak i nowa mapa z zaznaczonym obszarem w zależności od wybiegu gatunku.

Po wybraniu zakładki „Dla pracowników” pojawia się okienko logowania

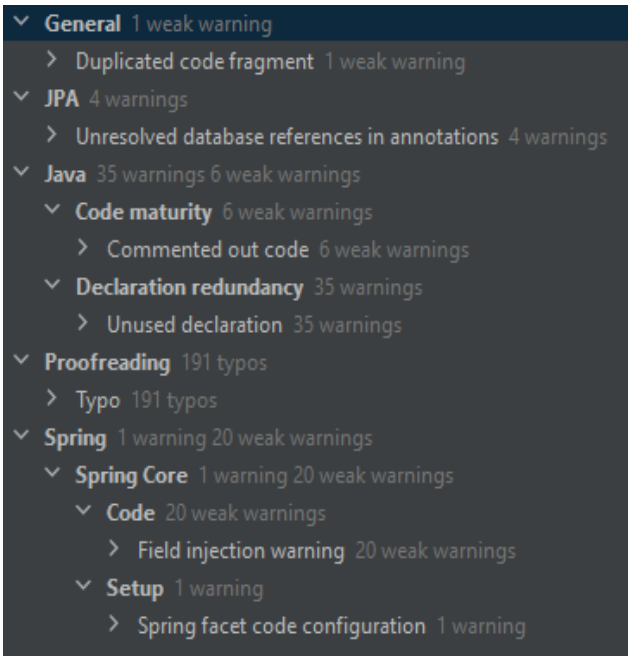


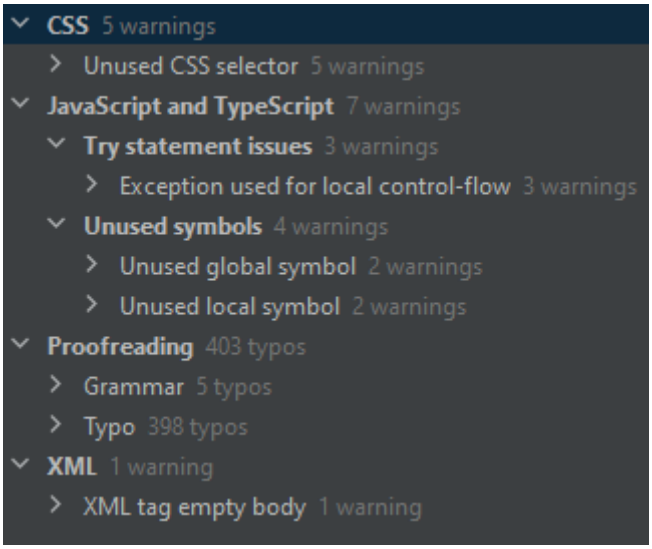
Po zalogowaniu jako administrator można wybrać jedną z pięciu kategorii co skutkuje wyświetleniem się odpowiedniej tabeli. Następnie można dodać, usunąć lub edytować obiekt poprzez kliknięcie odpowiedniego przycisku. Otwiera to formularz w prawej części interfejsu.



Po zalogowaniu jako opiekun ukazuje się klikalna lista aktualnie przydzielonych zadań. Po wybraniu jednego z nich w nowym okienku pojawiają się szczegółowe informacje, pełny opis jak i możliwość ukończenia zlecenia.

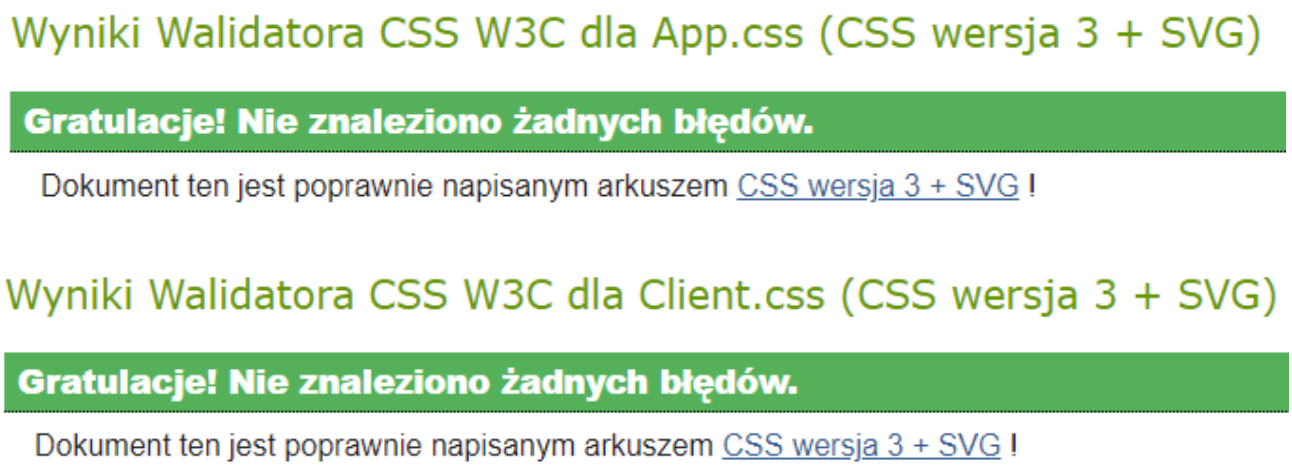
Raport CheckStyle

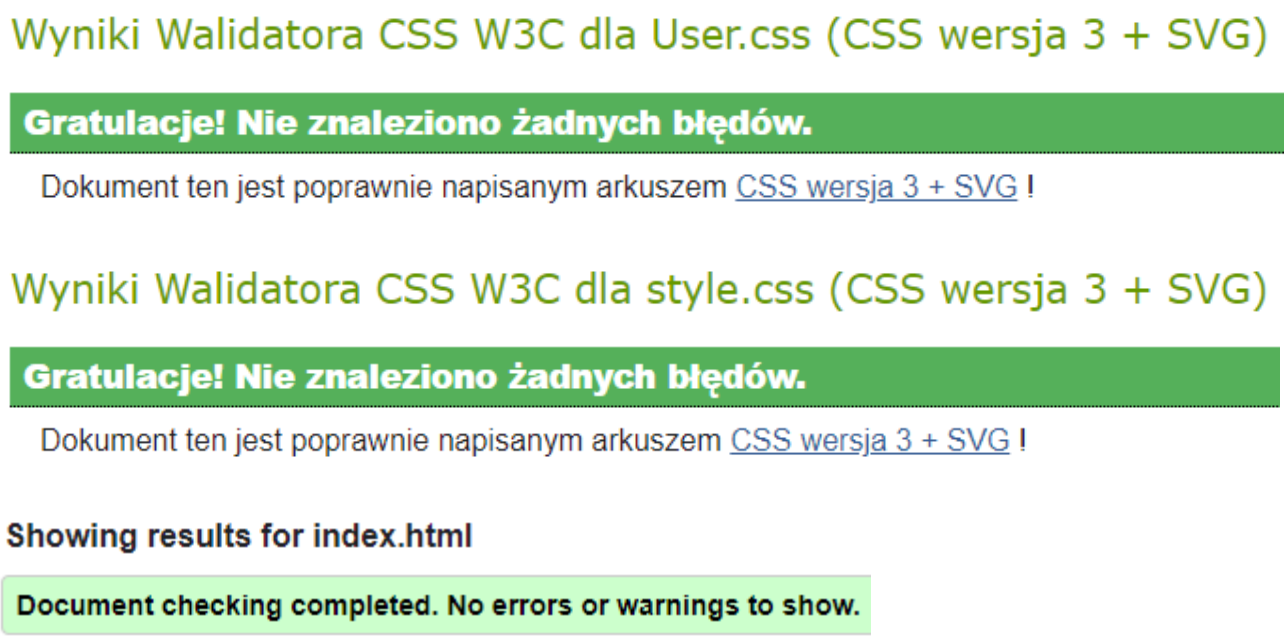




Ogromna część ostrzeżeń wynika z tego, że CheckStyle np. nie widzi, że zdefiniowane funkcje, które są serwisami są wywoływane przez kontrolery przez co uważa, że są bezużyteczne. Podobny problem jest z niektórymi polami zdefiniowanymi w CSS. W tym wypadku również uważa, że z pewnych zdefiniowanych zestawów stylów nie korzystam mimo, że jest inaczej. Dlatego też nie jesteśmy w stanie zmniejszyć pojawiających się ostrzeżeń do zera.

Raport W3C dla CSS i HTML





Wymagania środowiskowe oraz instrukcja zbudowania i uruchomienia aplikacji z kodu źródłowego

Aby uruchomić aplikację z kodu źródłowego należy zaopatrzyć się w IDE np. Intellij, VS Code lub Eclipse. Należy też stworzyć lokalnie bazę danych o nazwie zoo i ewentualnie zmienić port w pliku properties (przy korzystaniu z innej bazy niż MySQL), należy także pobrać program Nodejs, aby mieć możliwość uruchomienia frontendowej części aplikacji. Po połączeniu się z bazą danych należy uruchomić aplikację i środowisko Reacta (komenda npm start w folderze zooreact). Powinno to otworzyć okienko ze wstępną wersją interfejsu aplikacji w przeglądarce z adresem localhost://port:3000.

Znane ograniczenia

Zmiana roli użytkownika z ***user*** na ***admin*** nie jest możliwa. Uznaliśmy, że taka ważna zmiana nie powinna być zbyt prosta i powinna wymagać choć trochę więcej zaangażowania i czasu. Kolejnym nałożonym ograniczeniem jest to, że najpierw trzeba dodać gatunek, by dopiero potem móc dodać zwierzęta do niego należące. Powodem jest to, że dodając pierwsze zwierzę danego gatunku administrator musiałby uzupełnić oddzielny formularz dotyczący gatunku co jest niezgodne z przyjętą przez nas zasadą: jedna wybrana kategoria (w tym wypadku byłyby to zwierzęta) – jeden odpowiadający jej formularz. Następnym mechanizmem, który można uznać za ograniczenie jest to, że najpierw tworzymy tylko konto z rolą danego użytkownika a oddzielnie uzupełniamy dane osobowe. Chcieliśmy, by podczas logowania wszystkie potrzebne informacje zawierały się w jednej encji bazy danych. Ostatnim istotnym ograniczeniem jest dostępność tylko jednego rodzaju zadania. Otóż aktualnie można dodać tylko takie zadania, które rozpoczynają się o jakiejś porze ale nie takie, które trzeba skończyć do pewnej godziny. Taki typ zadań wymagałby o wiele trudniejszego algorytmu

weryfikującego dostępność opiekunów. Jednakże jest to ograniczenie, które można rozwiązać poprzez rozwijanie kodu.

Rozwiązane problemy

Naszym największym rozwiązanym problemem a jednocześnie funkcja, która jest chyba najbardziej zaawansowana to automatyczne dodawanie zadań związanych z karmieniem zwierząt do bazy danych przez serwer. Kolejną problematyczną funkcją była weryfikacja dostępności opiekunów czyli czy dodawane zadania nie kolidują już z przypisanymi. Chcieliśmy także, by nasza aplikacja była choć trochę bezpieczna. Osiągnęliśmy to poprzez enkrypcję haseł i zapisywanie ich zakodowanych form w bazie danych. Ważnym dla nas początkowym założeniem było, by to serwer wykonywał jak najwięcej logiki związanej z weryfikacją. Uznaliśmy, że to lepsza opcja niż ciągłe pobieranie potrzebnych informacji z bazy danych na komputer użytkownika.

Możliwości dalszego rozwoju aplikacji

Funkcjonalność, której możliwość rozwoju nasuwa się jako najbardziej oczywista to wspomniana przez nas wcześniej możliwość dodawania zadań z terminem ukończenia. Wydaje nam się, że taka funkcjonalność byłaby bardzo użyteczna ze względu na ogólną naturę zadań. Zazwyczaj znany jest termin i przewidywany czas trwania. Wiązałoby się to z totalną przebudową dotychczasowego algorytmu oraz uważamy, że do poprawnego działania powinno dodać się godziny pracy każdego z pracowników. Inną ciekawą funkcją byłoby przypisanie do każdego z opiekunów listy gatunków, którymi się opiekuje. Powinno to przyśpieszyć proces wyboru opiekuna do danego zadania. Wyświetlana lista dostępnych opiekunów dla danego zadania dostosowywałaby się do wybranego gatunku, którego dane zadanie by się tyczyło. Zupełnie innym rodzajem rozwoju byłoby polepszenie wyglądu interfejsu. Można byłoby skorzystać ze specjalnych funkcji zawartych w bibliotekach czy specjalnych arkuszy stylu.

Inne wnioski i spostrzeżenia

Bardzo chcielibyśmy wiedzieć czego jeszcze potrzebuje nasz projekt aby rzeczywiście mógł zostać użyty w prawdziwym ZOO. Co musi być zmienione a co dodane. Które z naszych zaproponowanych rozwiązań można uznać za profesjonalne, a które za nieodpowiednie. Czy nasz projekt jest skalowalny i czy można nazwać go łatwo rozwijalnym. Największą trudnością było to, że nie mieliśmy żadnego doświadczenia w tworzeniu tego typu projektów zwłaszcza z użyciem języków Java oraz JavaScrypt i wszystkie znalezione przez nas informacje i podpowiedzi musieliśmy konsultować ze sobą, by mieć pewność, że osiągniemy za ich pomocą odpowiednie rozwiązanie. Uważamy też, że gdyby nie ten projekt nie mielibyśmy okazji ani motywacji, by na własną rękę stworzyć coś podobnego i nauczyć się przy tym paru rzeczy.