

**Politechnika Śląska**

**Wydział Automatyki, Elektroniki i Informatyki**

**Kierunek Informatyka**

##### Projekt inżynierski

###### Domowy organizer

Autor: Piotr Kapski

Kierujący pracą: dr inż. Daniel Kostrzewa

Gliwice, Grudzień 2019

Załącznik Nr 2 do Zarz. Nr 97/08/09

**Oświadczenie**

Wyrażam zgodę/nie wyrażam\* zgody na udostępnienie mojej pracy dyplomowej/rozprawy doktorskiej\*

…………….………., dnia …………………………

……………………………………..……………….……

*(podpis)*

……………………………………………………………

*(poświadczenie wiarygodności podpisu przez Dziekanat)*

*\* właściwe podkreślić*

**Oświadczenie promotora**

Oświadczam, że praca „Tytuł pracy dyplomowej inżynierskiej” spełnia wymagania formalne pracy dyplomowej inżynierskiej.

|  |  |
| --- | --- |
| Gliwice, dnia ……………………… | ………………..……………….……  *(podpis)* |

Spis treści

[1. Wstęp 1](#_Toc526887085)

[2. Analiza tematu 5](#_Toc526887086)

[3. Wymagania i narzędzia 8](#_Toc526887087)

[4. Specyfikacja zewnętrzna 13](#_Toc526887088)

[5. Specyfikacja wewnętrzna 30](#_Toc526887089)

[6. Weryfikacja i walidacja 45](#_Toc526887090)

[7. Podsumowanie i wnioski 50](#_Toc526887091)

[Bibliografia i](#_Toc526887092)

[Spis skrótów i symboli iii](#_Toc526887093)

[Zawartość dołączonej płyty iv](#_Toc526887094)

[Spis rysunków v](#_Toc526887095)

[Spis tabel vi](#_Toc526887096)

# Wstęp

Jak wynika z raportu „Digital 2019”, ludzie na świecie spędzają średnio   
6 godzin i 42 minuty dziennie w Internecie [1]. Ciągły rozwój i wzrost popularności Internetu ma wpływ na wiele dziedzin życia. Jego oddziaływanie zauważyć można nie tylko na płaszczyźnie zawodowej, lecz także w życiu prywatnym, w tym także w obszarze życia domowego i rodzinnego, stanowiącego ważny element życia człowieka. W celu zwiększenia efektywności swoich działań i zaoszczędzenia czasu, ludzie szukają rozwiązań, które pozwolą na przyspieszenie codziennych czynności.

Celem pracy jest napisanie aplikacji internetowej, która ułatwi użytkownikom organizację codziennego życia rodzinnego. Aplikacja internetowa to program komputerowy pracujący na serwerze. Komunikuje się poprzez sieć komputerową z hostem użytkownika przy użyciu przeglądarki internetowej, będącej klientem aplikacji. Aplikacja umożliwi rodzinie tworzenie notatek, list oraz uzupełnianie wspólnego kalendarza. Dzięki dużej powszechności urządzeń elektronicznych, wszystkie elementy aplikacji będą dla użytkowników łatwo dostępne bez względu na miejsce i czas, co zapewni wygodę użytkowania oraz sprawi, że aplikacja będzie praktycznym narzędziem do organizacji życia rodzinnego. Aplikacja umożliwi sprawniejszą komunikację w rodzinie, a także wpisze się w trend cyfryzacji, a co za tym idzie – przyniesie pozytywne skutki dla środowiska poprzez rezygnację z papierowych notatek

Praca będzie obejmowała zarówno aplikację serwerową jak i kliencką (wygląd i zachowanie aplikacji). Część serwerowa opiera się na komunikacji   
z bazą danych oraz wykonywaniem na niej operacji typu zapis czy odczyt. Aplikacja kliencka odpowiada za to co widzi użytkownik. Korzysta ona   
z danych zwróconych przez serwer oraz wyświetla je na stronie w jak najczytelniejszy sposób.

Rozdział drugi przedstawia dokładniejszą analizę wybranego tematu oraz literatury, a także porównuje podobne istniejące już rozwiązania. W rozdziale trzecim przedstawione są wymagania funkcjonalne oraz niefunkcjonalne,   
a także opis użytych narzędzi programistycznych oraz metodyka pracy. Rozdział czwarty skupia się na wyglądzie aplikacji oraz jego sposób obsługi   
z perspektywy użytkownika. Piąta część pracy opisuje działanie aplikacji od jej środka – jej logikę, wykorzystane biblioteki czy podział komponentów. Rozdział szósty skupia się na testowaniu aplikacji oraz walidacji poprawności jej działania. Przedstawione są tam zarówno sposoby testowania, jak również przykłady usuniętych błędów. Dokument kończy podsumowanie całego procesu tworzenia, zaplanowanie potencjalnych przyszłych kierunków prac, a także opis problemów napotkanych w trakcie implementacji.

Autor pracy jest jej jedynym twórcą i wszystkie elementy w niej zawarte zostały przez niego zaimplementowane.

# Analiza tematu

Organizer domowy z założenia powinien usprawnić organizowanie czasu rodzinie. Korzystać z niego będą osoby w każdym wieku, dlatego aplikacja powinna być prosta i intuicyjna. Zbytnie skomplikowanie możliwości lub samego interfejsu może spowodować, że dzieci lub ludzie mniej związani   
z komputerami, mogliby mieć problemy z obsługą aplikacji. Nie jest to dopuszczalne, ponieważ powinna ona jak najbardziej wspierać każdego członka rodziny.

Biorąc pod uwagę wyżej wymienione założenia, aplikacja webowa wydaje się być optymalnym rozwiązaniem. Aplikacje internetowe do działania wymagają jedynie komputera lub telefonu, dostępu do Internetu oraz zainstalowanej przeglądarki, co w obecnych czasach jest standardem w większości domów. Mają one również ogromne możliwości, są wieloplatformowe, a aktualizacje mogą być wprowadzane bez żadnej ingerencji użytkownika.

W perspektywie przyszłości, warto tak rozwijać aplikację, by była otwarta na możliwości rozwoju. W tym celu, dobrym rozwiązaniem jest napisanie RESTful API (ang. Representational State Transfer Application Programming Interface). API jest sposobem komunikacji pomiędzy różnymi elementami oprogramowania (interfejs). REST natomiast, to styl architektury oprogramowania, opisany przez Roy’a Fieldinga w 2000 roku, określający zasady projektowania API [7]. Dużym plusem zastosowania go, jest możliwość użycia jednej logiki do wielu widoków. Mogą one być napisane   
w różnych językach programowania, ponieważ dane transportowane są uniwersalnymi formatami jak JSON czy XML. Możliwe dzięki temu będzie napisanie w przyszłości np. aplikacji mobilnej do projektu.

Najważniejsze jednak jest, jakie możliwości oferuje aplikacja. Ludzie przez nadmiar obowiązków, by zapamiętać ważne dla nich rzeczy, mogą utrwalać je   
w następujące sposoby:

* Krótkie wiadomości tekstowe (np. notatki)
* Listy (np. lista zakupów, lista rzeczy na wakacje)
* Wydarzenia zapisywane w kalendarzu

Aplikacja by dawać jak największą optymalizację czasu, powinna dawać wszystkie te możliwości. Umożliwi to przechowywanie wszystkich ważnych dla rodziny wiadomości oraz wydarzeń w jednym miejscu, co nie tylko uporządkuje je, lecz także pomoże chronić środowisko. Korzystając ze strony internetowej, możemy także dowolnie edytować nasze wiadomości oraz dawać im dodatkowe funkcjonalności jak wygasanie po czasie. By usprawnić działanie systemu, najlepsze będzie stworzenie tzw. Single-Page Application. Nawigacja na takiej stronie polega na asynchronicznym ładowaniu poszczególnych elementów, ograniczając w ten sposób przeładowywanie całych stron. W tym celu, dobrym rozwiązaniem jest użycie frameworków takich jak React czy Angular.

## Opis znanych rozwiązań

Aktualnie na rynku jest wiele rozwiązań, oferujących tworzenie notatek lub uzupełnianie kalendarza. Są one jednak zazwyczaj tworzone jako dwie osobne aplikacje. Tworzenie notatek i list umożliwia między innymi:

* Google Keep
* ColorNote
* OneNote
* Evernote

Mają one jednak, z perspektywy projektu, parę znaczących wad. Po pierwsze nie zawierają funkcjonalności kalendarza, który jest bardzo ważną częścią wielu gospodarstw domowych. Po drugie, nie oferują tworzenia ich dla wcześniej zdefiniowanej grupy ludzi. Część z nich umożliwia udostępnianie zasobów przez np. portale społecznościowe, znacząco utrudnia to jednak komunikacje.

Aplikacje umożliwiające tworzenie kalendarza to na przykład:

* Kalendarz Google
* Kalendarz PL
* Business Calendar
* Microsoft Outlook Calendar

Kalendarz Google lub Microsoft Outlook Calendar spełniają podobną funkcje, jaką chciałby uzyskać projekt. Są to jednak odosobnione aplikacje, co zmniejsza wygodę użytkownika. Produktem łączącym oba mechanizmy jest aplikacja Any.do, która jednak nie umożliwia tworzenia grup, co jest sprzeczne z podstawowym celem projektu.

# Wymagania i narzędzia

## Wymagania funkcjonalne

* Użytkownik może zarejestrować się do systemu podając nazwę użytkownika, e-mail oraz hasło
* Użytkownik może uwierzytelniać się za pomocą nazwy użytkownika i hasła
* Użytkownik może edytować wszystkie swoje dane oraz hasło
* Każdy użytkownik ma równe prawa
* Użytkownicy mogą tworzyć lub dołączać do istniejącego gospodarstwa domowego (ang. household)
* Użytkownicy mogą tworzyć, edytować i usuwać wspólne dla domostwa notatki i listy oraz uzupełniać wspólny kalendarz

## Wymagania niefunkcjonalne

* Aplikacja kompatybilna z przeglądarkami: IE, Firefox oraz Chrome
* Aplikacja powinna być wieloplatformowa
* Serwer aplikacji powinien być zabezpieczony przed niezautoryzowanym dostępem
* Aplikacja powinna być jak najbardziej wydajna
* Aplikacja powinna być dla użytkownika łatwa w obsłudze   
  i intuicyjna

## Opis narzędzi

### Java

Język programowania, w którym został napisany serwer aplikacji. Jest to jeden z najpopularniejszych języków na świecie do pisania aplikacji webowych, którego dużą zaletą jest multiplatformowość. Java posiada obszerną liczbę platform programistycznych (ang. Framework), które znacząco ułatwiają pracę.

### IntelliJ IDEA

Najpopularniejsze zintegrowane środowisko programistyczne dla Javy stworzone przez firmę JetBrains. Posiada ogromną liczbę narzędzi wspomagających pisanie kodu w takich sferach takich jak refaktoryzacja, automatyczne podpowiedzi, zaawansowany debugger oraz wsparcie dla licznych frameworków, czy narzędzi jak np. GitHub. IntelliJ został stworzony tak, aby zmaksymalizować produktywność programistów i ograniczyć użycie myszki przez stworzenie licznych skrótów klawiszowych. Jedynym większym konkurentem tego środowiska jest Eclipse, którego największą zaletą jest posiadanie otwartych źródeł (ang. Open source) i to, że jest całkowicie darmowe nawet do zastosować komercyjnych. Środowisko stworzone przez JetBrains jest jednak zdaniem autora zdecydowanie wygodniejsze, dlatego zostało wybrane do tego projektu.

### Spring framework

Spring jest szkieletem tworzenia aplikacji dla platformy Java EE (Enterprise Edition) [2]. Przez ogrom możliwości i usprawnień, które oferuje, jest on jednym z ważniejszych frameworków dla Javy i jest używany przez dużą ilość programistów tego języka. Spring składa się z wielu projektów, które mogą być używane niezależnie np. Spring Cloud, Spring Data czy Spring Batch. W tym projekcie, wykorzystano:

* Spring Boot – zdaniem autora, jeden z najważniejszych części frameworku. Oferuje szybkie stworzenie gotowej do użycia aplikacji, która zawiera wszystkie potrzebne komponenty.   
  W trakcie programowania oferuje znaczne uproszczenia kodu, np. przez rezygnacje z plików XML do konfiguracji.
* Spring Data – używany do komunikacji oraz obsługi bazy danych, a także do mapowania obiektowo-relacyjnego. Jest alternatywą do popularnego Hibernate’a, który jednak przez sporą liczbę wad, zostaje powoli wyparty przez nowsze technologie takie jak Spring Data.
* Spring Security – zapewnia autoryzację oraz autentykacje dla aplikacji. Jest to potężny framework o dużych możliwościach konfiguracji, posiadający dobrze napisaną dokumentację.

### Angular

Angular to otwarty framework napisany w języku TypeScript, stworzony   
i wspierany przez Google [3]. Można dzięki niemu szybko i łatwo zbudować aplikacje internetową, nie tracąc przy tym na ilości możliwości, którymi dysponuje. W projekcie był on używany w wersji 8 do napisania aplikacji klienckiej. Alternatywą dla Angulara jest między innymi React, który jest biblioteką JavaScript. Obie technologie są jednymi z ważniejszych w świecie front-endu oraz mają swoje plusy i minusy, dlatego każdy wybór oferował wystarczające możliwości.

### Baza danych MongoDB

MongoDB jest nierelacyjną bazą danych, czyli nie posiada sztywno określonej struktury danych w niej przechowywanych. W przeciwieństwie do klasycznego, relacyjnego podejścia, dane składowane są w postaci dokumentów w formacie JSON, który jest dla aplikacji bardzo naturalną strukturą. Mongo jest przy tym bezpieczne, szybkie i bardzo skalowalne, a także jest dobrze wspierane przez Spring Data. Jest to aktualnie najczęściej używana nierelacyjna baza danych, a jej popularność rośnie z roku na rok [4].

### Swagger

Swagger jest znany przede wszystkim jako narzędzie do tworzenia dokumentacji [5]. W projekcie został jednak również użyty do generowania API (Interfejs Programowania Aplikacji) wraz z encjami modelu. Połączenie tych dwóch funkcjonalności daje bardzo pomocne narzędzie w tego typu aplikacjach.

### Robo 3T

Robo 3T to graficzny interfejs do zarządzania bazą danych [6]. Jest to lekki i darmowy odpowiednik bardziej rozbudowanego Studio 3T, jednak na potrzeby projektu Robo jest zdecydowanie wystarczające.

## Metodyka pracy

Podczas pracy nad projektem wykorzystywane było narzędzie do kontroli wersji Git. Każde nowe zadanie niosło za sobą stworzenie nowej gałęzi   
(ang. branch), aby zasymulować pracę w realnym projekcie jak i umożliwiało to rozwijanie paru funkcjonalności jednocześnie. Praca rozpoczęta została od stworzenia wstępnego serwera, który był później na bieżąco modyfikowany na potrzeby wymagań powstałych w trakcie implementacji.

# Specyfikacja zewnętrzna

Aplikacja do działania wymaga jedynie dostępu do Internetu oraz posiadania jednej z następujących przeglądarek:

* IE 11+
* Google Chrome
* Mozilla Firefox
* Safari 7+
* Edge 13+

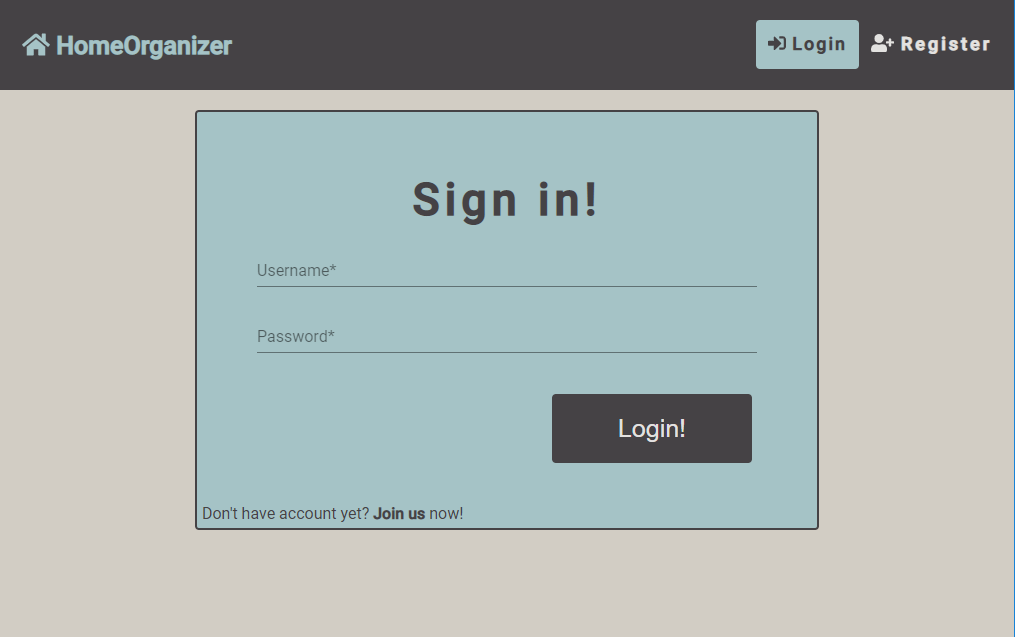
W systemie wyróżniamy tylko jedną kategorie użytkowników – użytkownik systemu. Może on być w dwóch stanach:

* Użytkownik niezalogowany – jedynymi akcjami jakie może wykonać jest rejestracja oraz logowanie.
* Użytkownik zalogowany – gość po zalogowaniu otrzymuje dostęp do wszystkich funkcji. Wszyscy użytkownicy mają równe prawa.

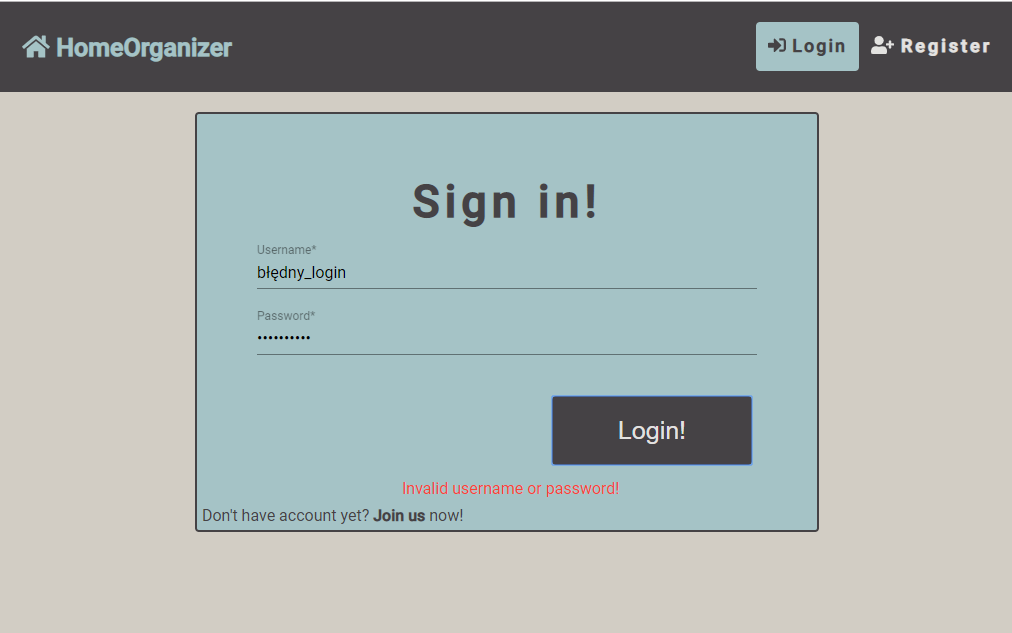
## Użytkownik niezalogowany

### Logowanie

Po wejściu do aplikacji, pierwszą stroną ukazującą się gościowi jest formularz logowania przedstawiony na rysunku 4.1. Musi on podać poprawną nazwę użytkownika i hasło, w przeciwnym razie, zostanie wyświetlony stosowny komunikat (Rys.4.2). W przypadku sukcesu, użytkownik zostaje zalogowany i przekierowany do menu głównego (Rys. 4.5).

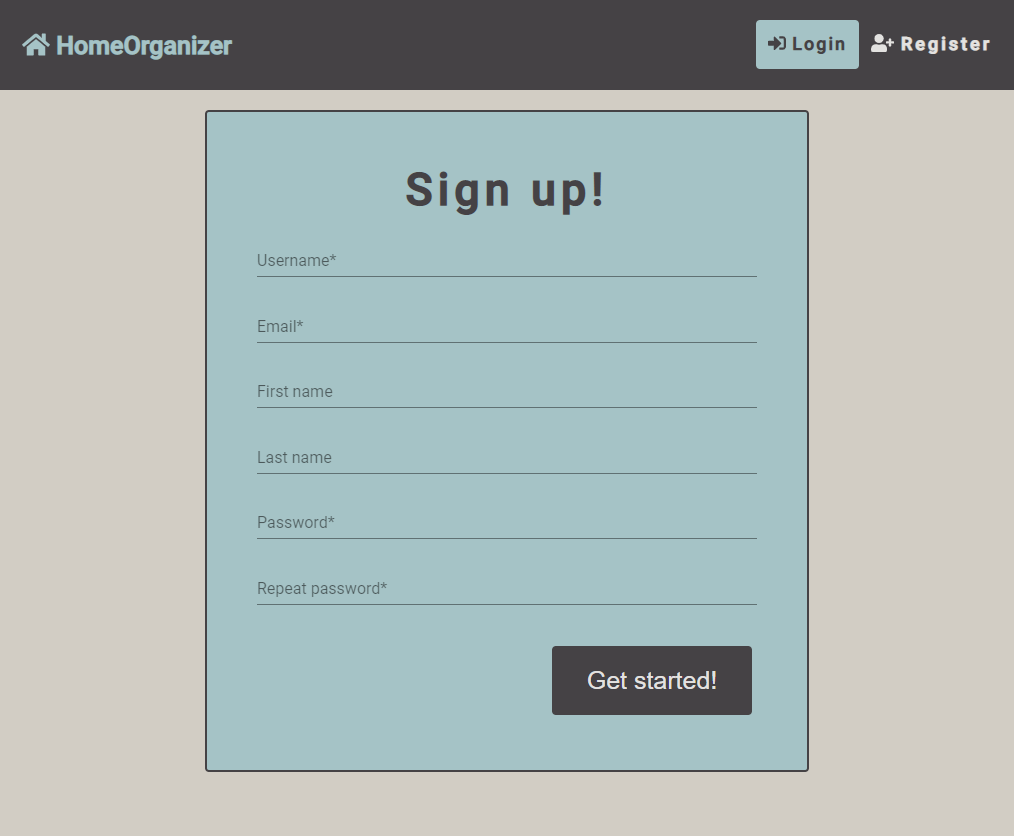


|  |
| --- |
| Rys.4.1. *Formularz logowania* |
|  |

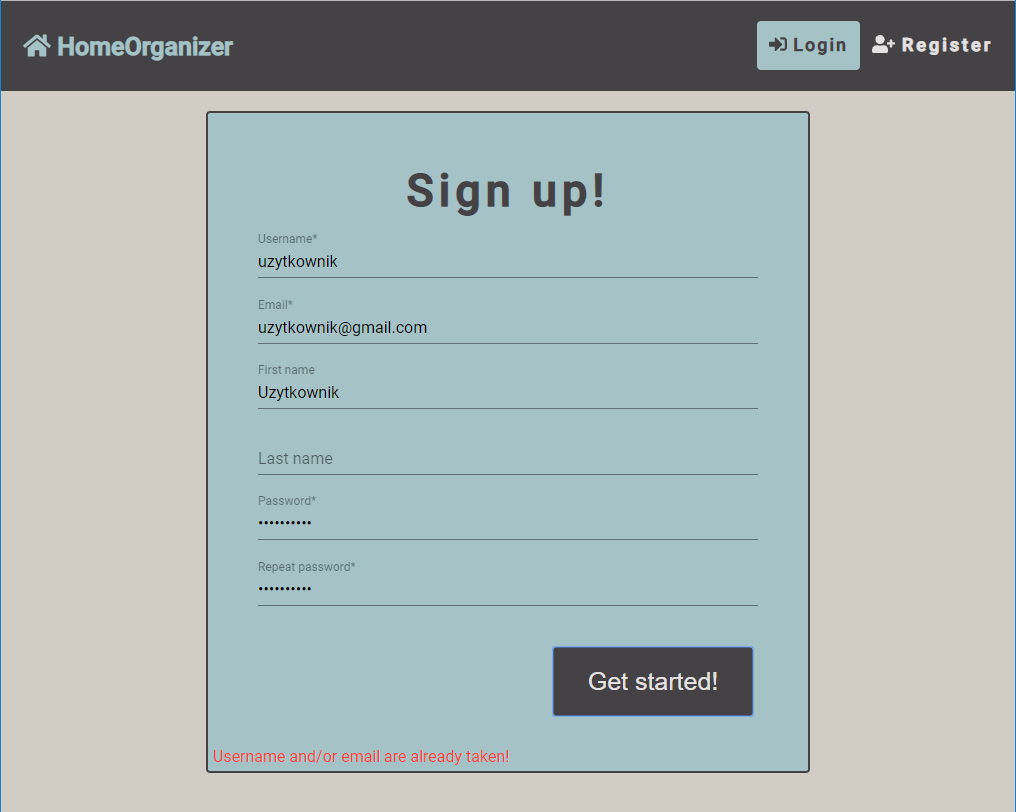
Rys.4.2. *Formularz logowania z błędem*

### Rejestracja

W przypadku, gdy użytkownik nie ma jeszcze utworzonego konta, należy nacisnąć na przycisk Register w nagłówku lub na napis **Join us** na dole formularza (Rys. 4.2). Aplikacja przekierowuje wtedy do strony rejestracji z   
rysunku 4.3. Należy w nim podać unikalną nazwę użytkownika (min. 5 znaków), unikalny email, opcjonalne imię i nazwisko oraz dwukrotnie hasło (min. 5 znaków). Po wypełnieniu pól, naciśnięcie przycisku **Get started!**   
w przypadku porażki wyświetli komunikat błędu (Rys. 4.4), w przeciwnym razie użytkownik zostaje utworzony i ma możliwość zalogowania się.

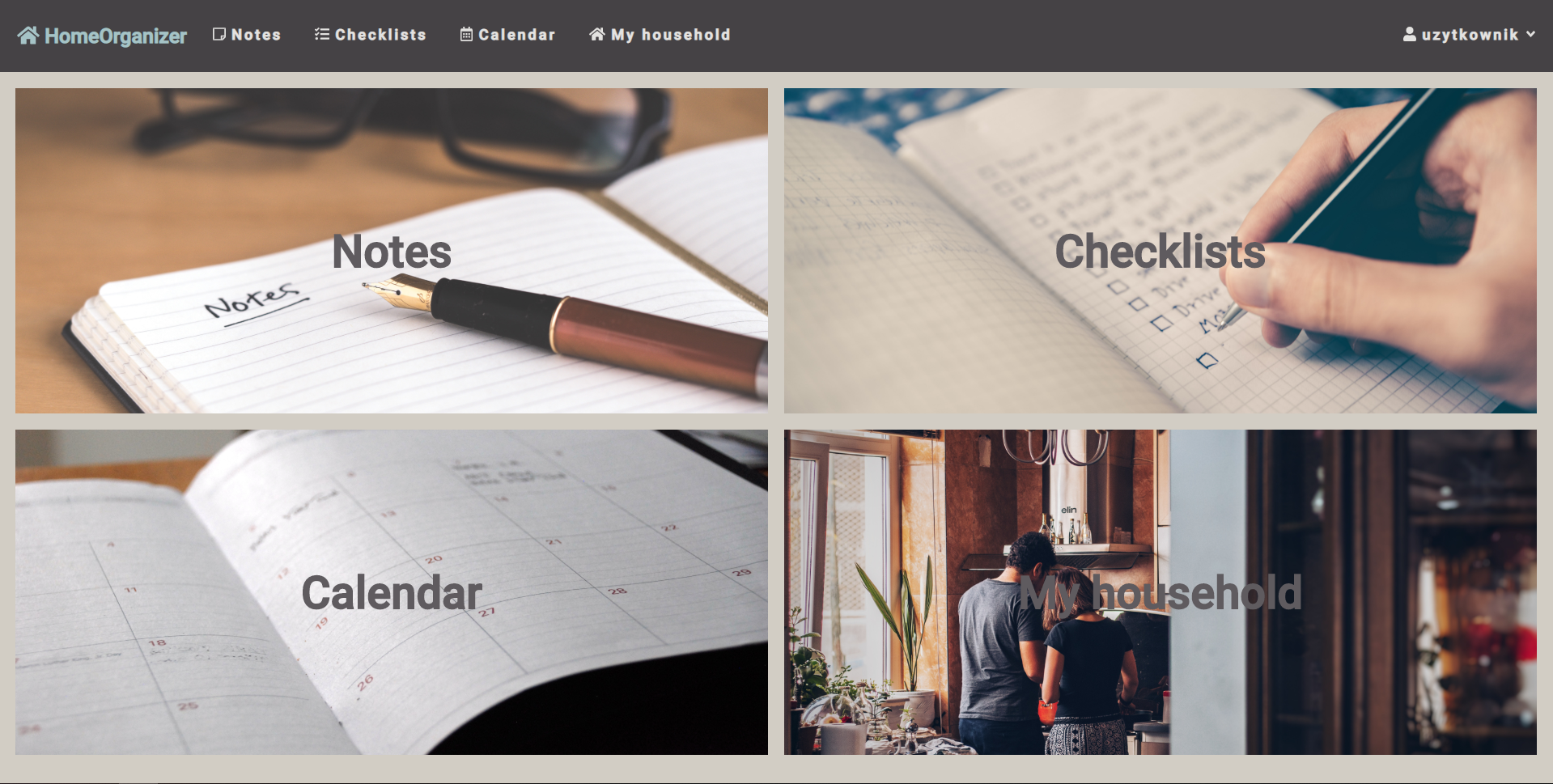


Rys.4.3. *Formularz rejestracji*



Rys.4.4. *Formularz rejestracji z błędem*

## Użytkownik zalogowany



Rys.4.5. *Strona główna użytkownika*

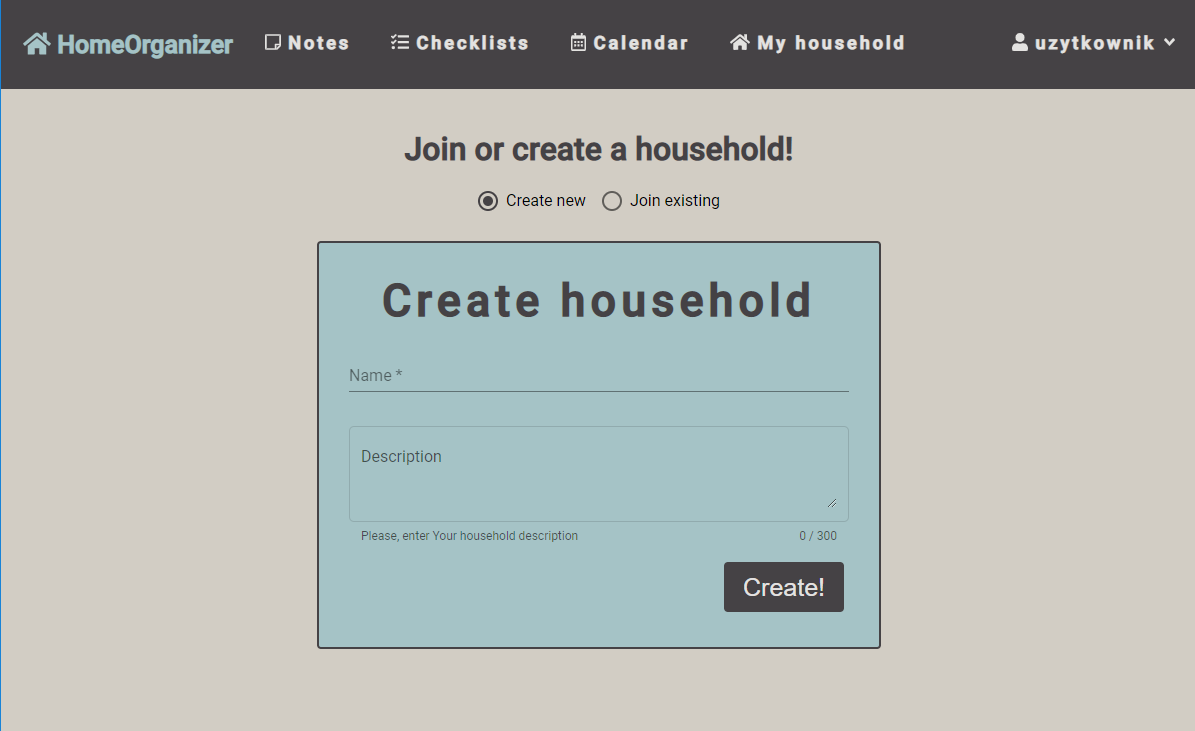
Będąc zalogowanym, użytkownik może wykonać następujące akcje:

* Zarządzać notatkami swojego household’u
* Zarządzać listami swojego household’u
* Zarządzać kalendarzem swojego household’u
* Zarządzać household’em
* Edytować swoje dane i hasło
* Wylogować się

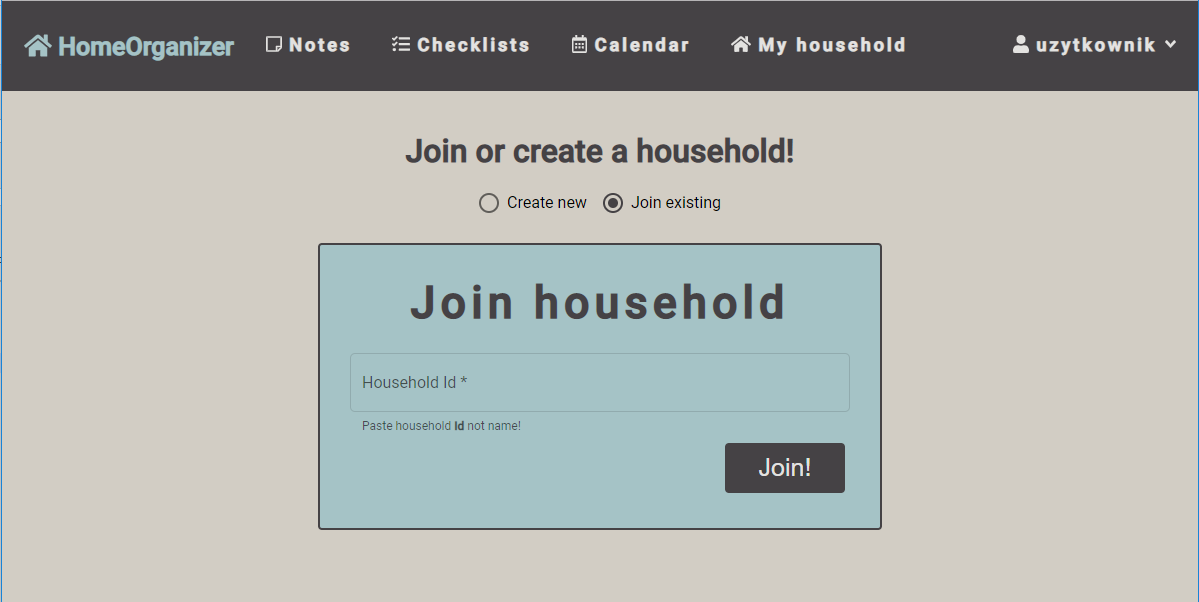
### Household

Household w aplikacji pełni role wirtualnego domu. Zawiera on nazwę, opis, unikalny identyfikator nadawany przez aplikacje oraz listę członków. Wszystkie akcje za wyjątkiem edycji konta odbywają się w obrębie domostwa i tylko jego członkowie mogą widzieć dodane notatki, listy oraz uzupełniony kalendarz. Użytkownik nie będący w żadnym domostwie, może jedynie edytować swoje dane.

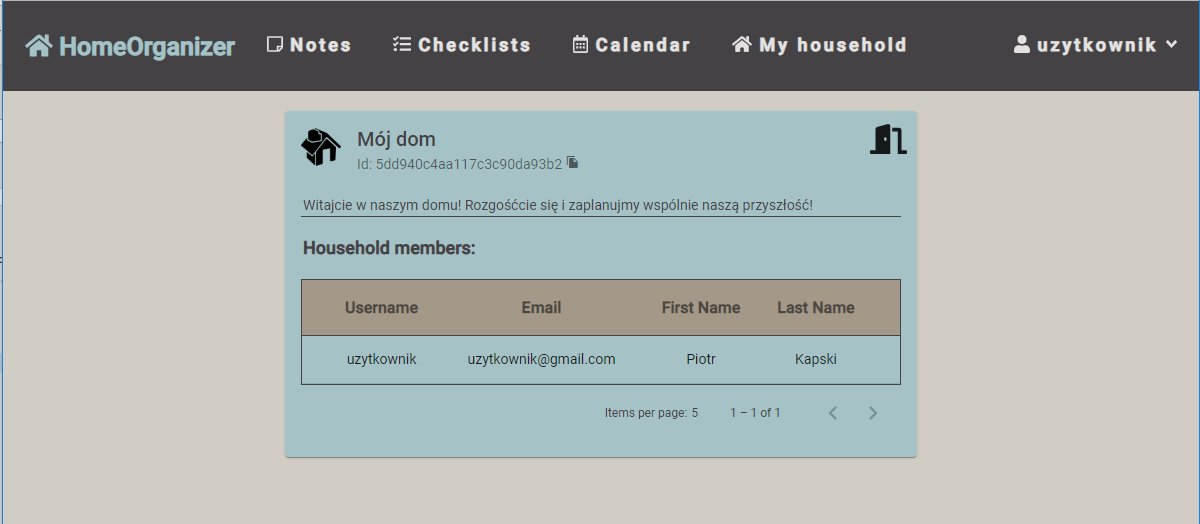
Chcąc wyświetlić swój obecny household, stworzyć nowy bądź dołączyć do już istniejącego, należy wybrać zakładkę **My Household** w nagłówku lub menu głównym. Jeśli nie jesteśmy jeszcze w żadnym domostwie, zostaniemy przekierowani do formularza tworzenia nowego, jak przedstawiono na rysunku 4.6. Należy podać nazwę, która nie musi być unikalna, oraz opcjonalnie opis, by po kliknięciu przycisku **Create!** stworzyć household i być jego pierwszym członkiem (Rys. 4.8). Jest również możliwość dołączenia do już istniejącego domostwa, przez wybranie opcji **Join existing** (Rys 4.7). Należy wtedy w puste pole wpisać identyfikator domostwa otrzymany od członka, który już się w nim znajduje (Id pod nazwą na rysunku 4.8).



Rys.4.6. *Formularz tworzenia household’u*



Rys.4.7. *Dołączanie do istniejącego household’u*



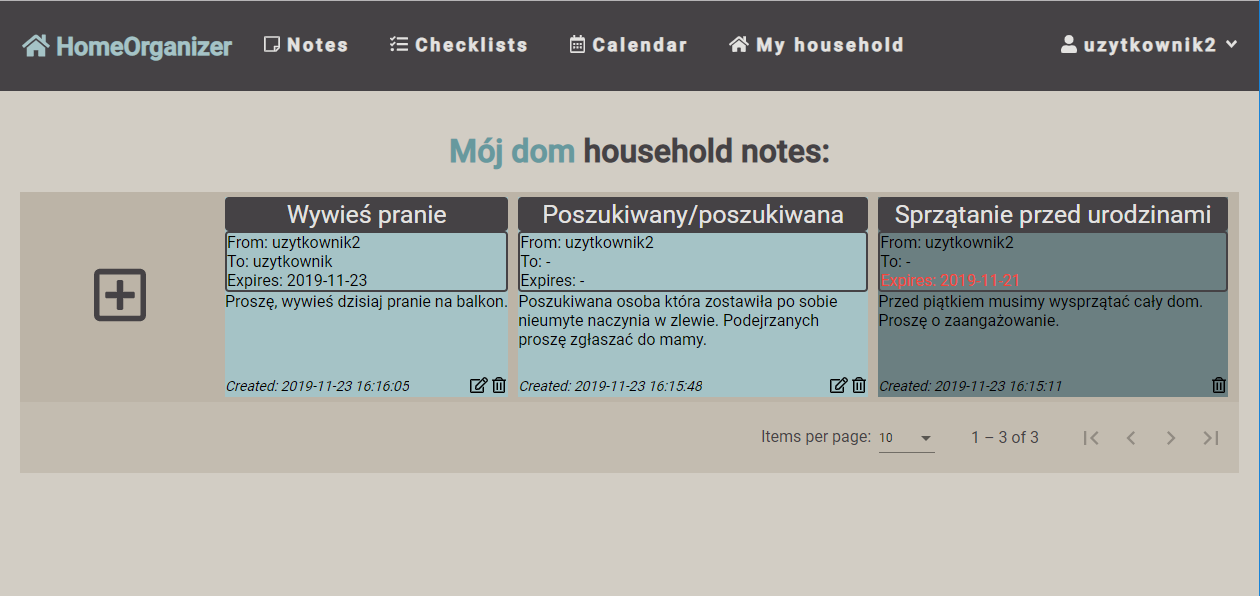
Rys.4.8. *Widok aktualnego household’u*

W widoku na rysunku 4.8 wyświetlają się szczegóły aktualnego householdu, to jest: nazwa (Mój dom), identyfikator, który możemy skopiować po kliknięciu na ikonę po jego prawej, opis (Witajcie w …) oraz lista członków. W prawym górnym roku, po kliknięciu w otwarte drzwi, jest możliwość opuszczenia domostwa.

### Notatki

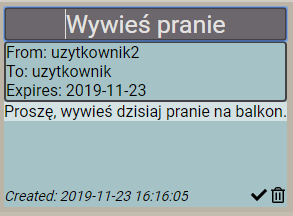
Notatki są krótkimi wiadomościami skierowanymi do całego householdu lub konkretnego jego członka. Można nimi zarządzać przez wejście z zakładkę **Notes**. Przykładowe dodane notatki przedstawione są na rysunku 4.9. Każda notka zawiera tytuł, twórcę, odbiorcę, datę wygaśnięcia oraz wiadomość. Jeśli termin notatki został przekroczony, staje się ona ciemniejsza (ostatnia notatka na rysunku 4.9) i nie można jej już edytować. Każdą karteczkę można edytować (zmienić można tylko tytuł i wiadomość, przykład na rysunku 4.10) oraz usuwać przez ikony znajdujące się w prawym dolnym rogu. Uczynić to może tylko i wyłącznie jej twórca. Po usunięciu notatki, na dole strony pojawi się   
o tym odpowiednia wiadomość, wraz z przyciskiem do cofnięcia operacji   
(Rys. 4.11). Jeśli odbiorca lub data wygaśnięcia nie zostaną ustalone, wyświetlany jest w odpowiednim miejscu znak „**-**” (druga notatka na   
rysunku 4.9).

W przypadku większej liczby notatek są one dzielone na strony. Ich ilość na jednej stronie możemy ustalić przez zmianę parametru „Items per page”.

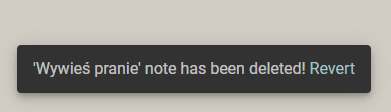


Rys.4.9. *Zakładka notes*

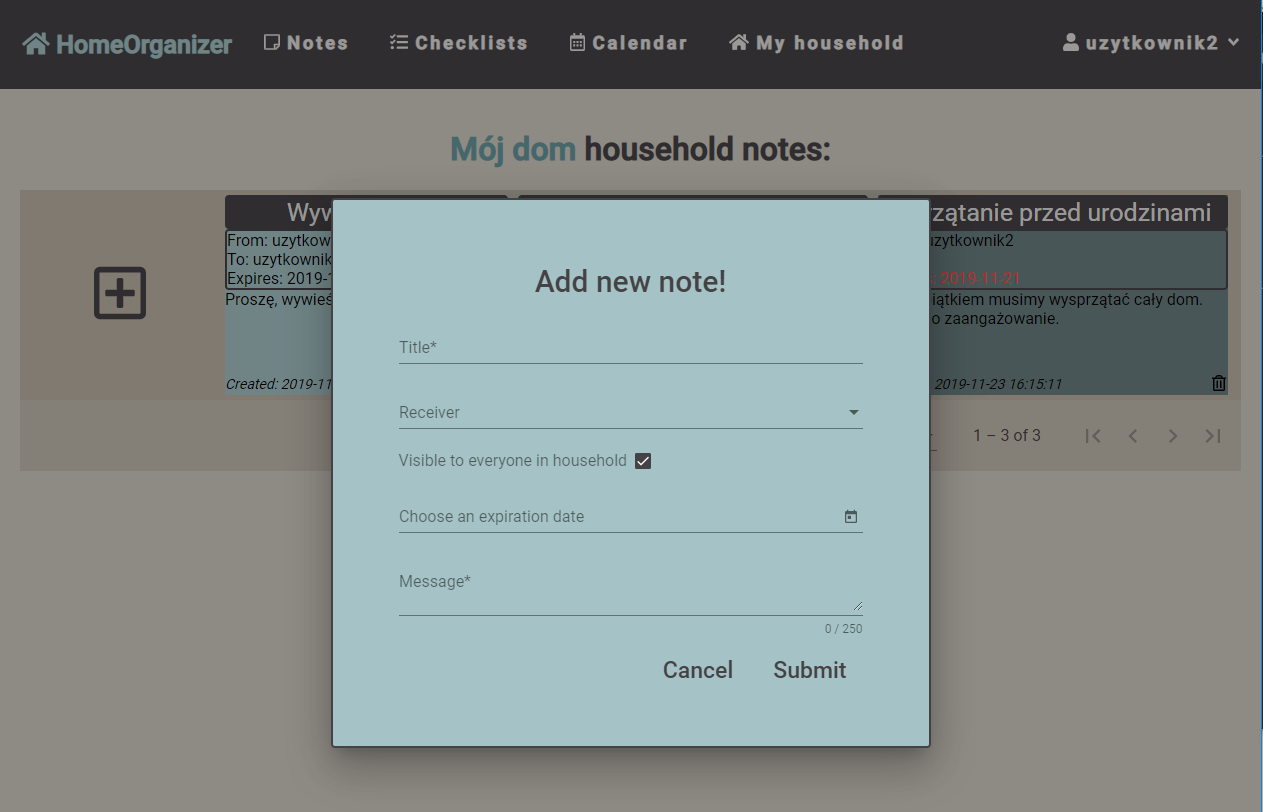
Plus po lewej stronie rysunku 4.9, umożliwia użytkownikowi dodanie nowej notatki. Po naciśnięciu przycisku, wyświetla się nowe okno przedstawione na rysunku 4.12. W kolejne pola użytkownik wpisuje kolejno: tytuł, opcjonalnie odbiorcę oraz datę wygaśnięcia, a także wiadomość (do 250 znaków). Domyślnie, odbiorcą jest całe domostwo. Zmiana tego pola, niesie za sobą również możliwość ustalenia czy wiadomość ma być widoczna tylko dla odbiorcy, czy również dla reszty członków. Po wpisaniu wymaganych informacji, dodanie notatki następuje po naciśnięciu przycisku **Submit**.



Rys.4.10. *Edytowanie notatki*



Rys.4.11. *Wiadomość po usunięciu wiadomości*

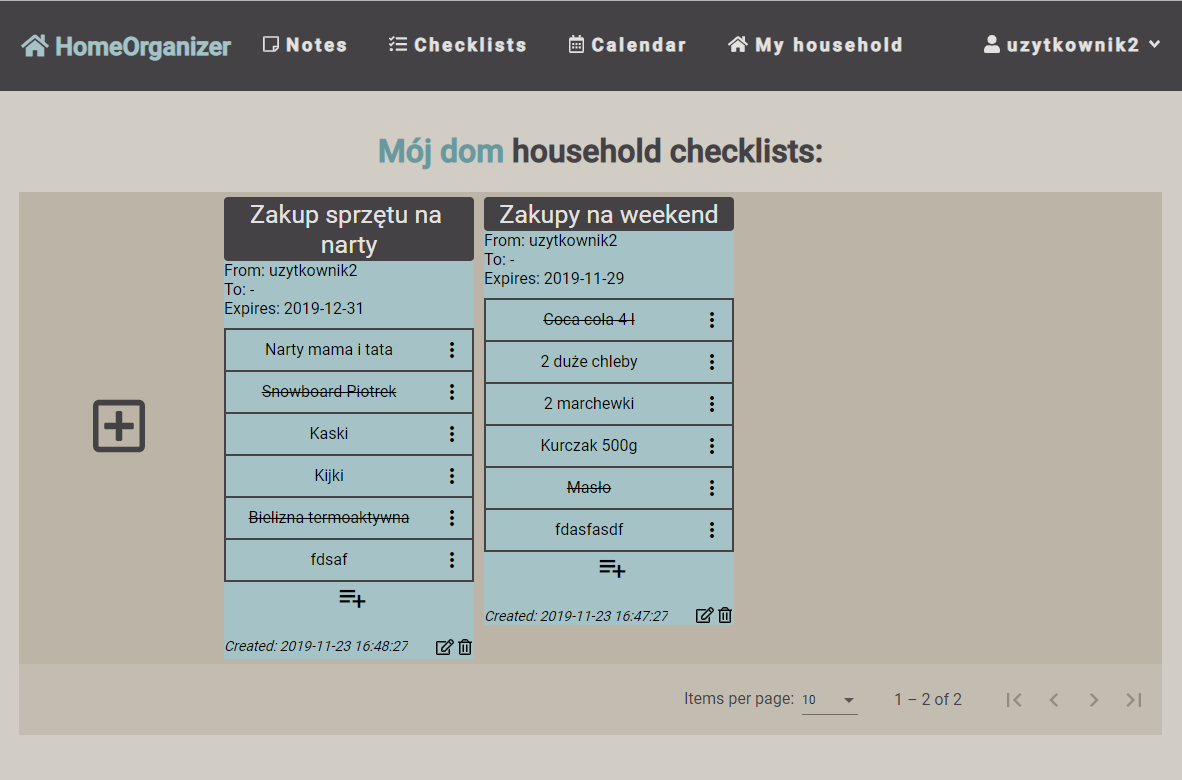


Rys.4.12. *Okno dodawania nowej notatki*

### Listy

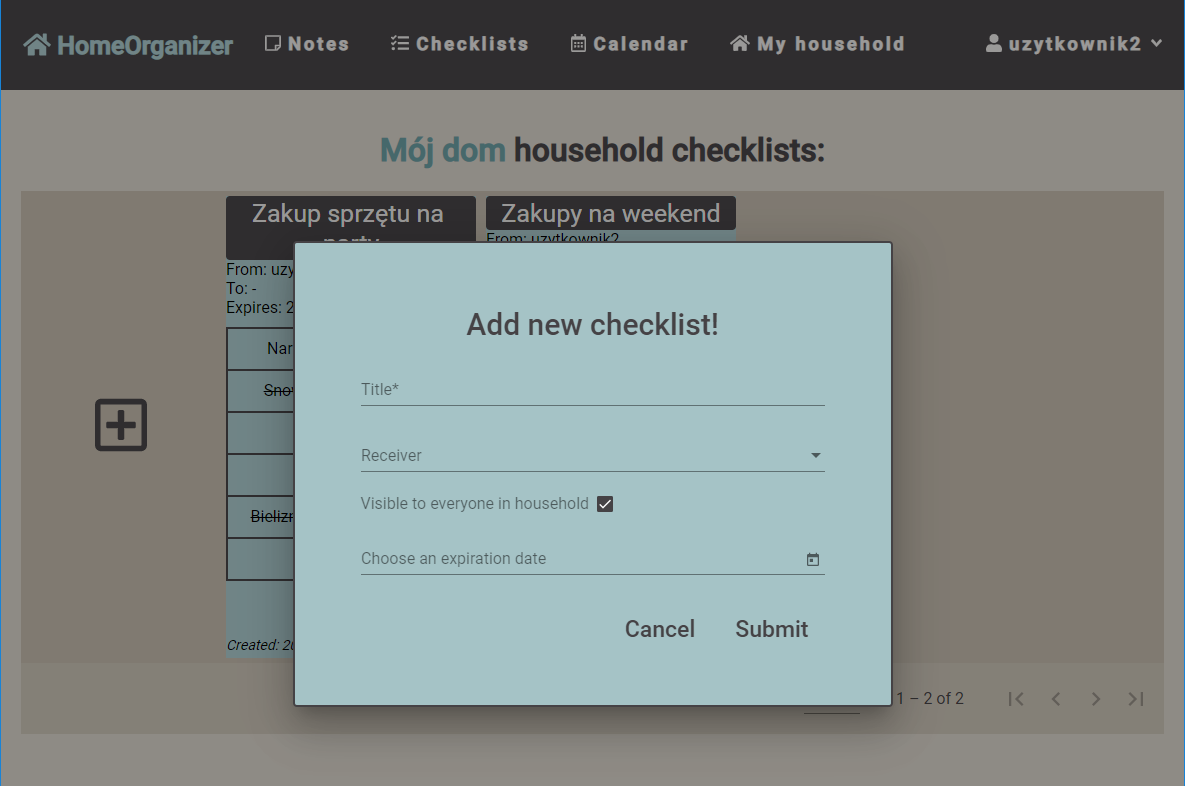
Listy są listami kontrolnymi tworzonymi przez całe domostwo lub konkretnego jego członka. Przykładem zastosowania takiej struktury może być np. wspólna lista zakupów. Panel zarządzania nimi dostępny jest po wejściu *Kalendarz -*w zakładkę **Checklists** (Rys. 4.13). Każda lista zawiera tytuł, twórcę, odbiorcę, datę wygaśnięcia oraz listę przedmiotów (ang. Item). Jeśli odbiorcą jest cały dom (znak „**-**”), wtedy edytować oraz dodawać nowe elementy może każdy. *Kalendarz -*W sytuacji jednak, gdy adresatem jest konkretna osoba, tylko on i twórca może zmieniać listę.

Checklista może być edytowana (tytuł) oraz usunięta (z możliwością przywrócenia jak w notatkach, rysunek 4.11). Przedmiot natomiast może być skreślony (odkreślony), edytowany oraz usunięty. Nowe elementy dodaje się znakiem notatki z plusem znajdującym się na dole spisu przedmiotów, natomiast nową listę – plusem w kwadracie. Listy są również stronicowane.



Rys.4.13. *Zakładka checklists*

Po naciśnięciu przycisku dodania nowej listy, wyświetla się nowe okno przedstawione na rysunku 4.14. Pola formatki są analogiczne do notatek, a chęć dodania potwierdzana jest przyciskiem **Submit**.

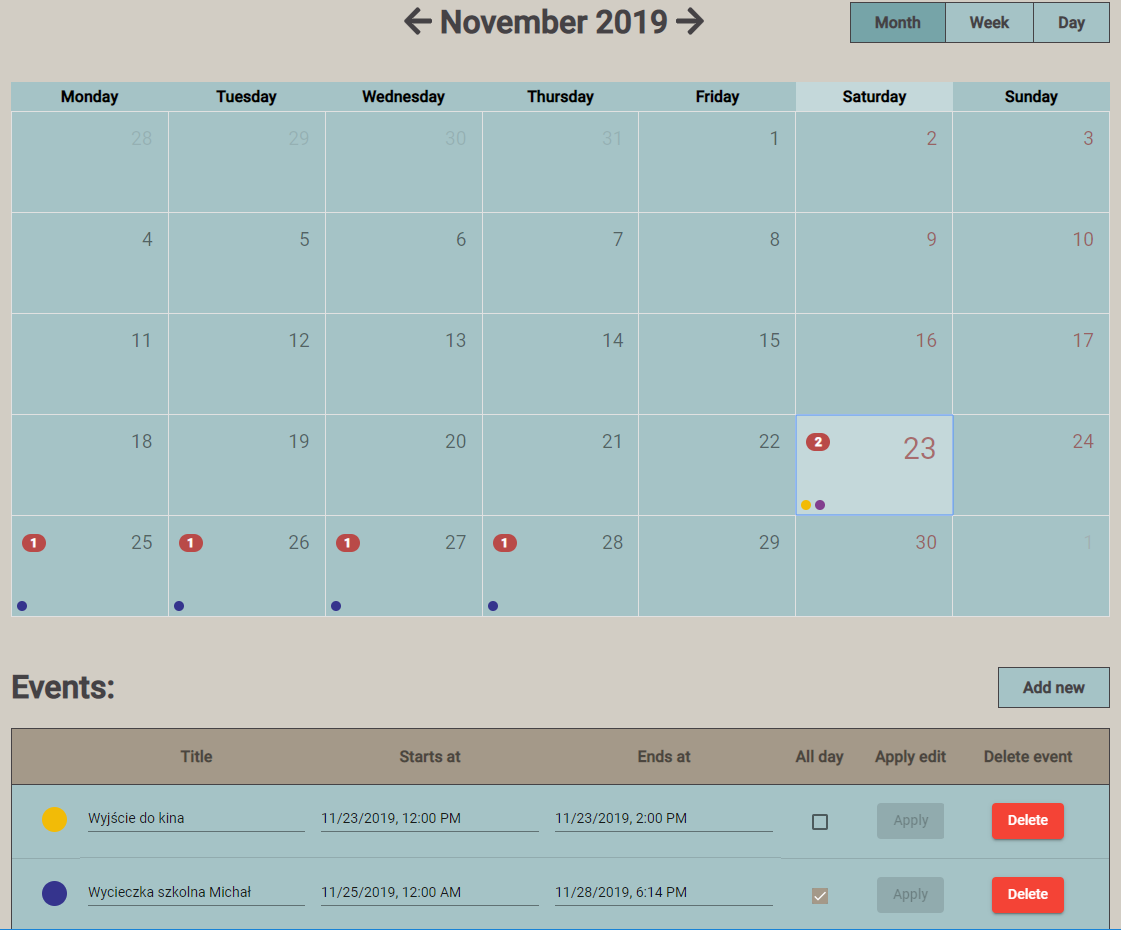


Rys.4.14. *Okno dodawania nowej listy*

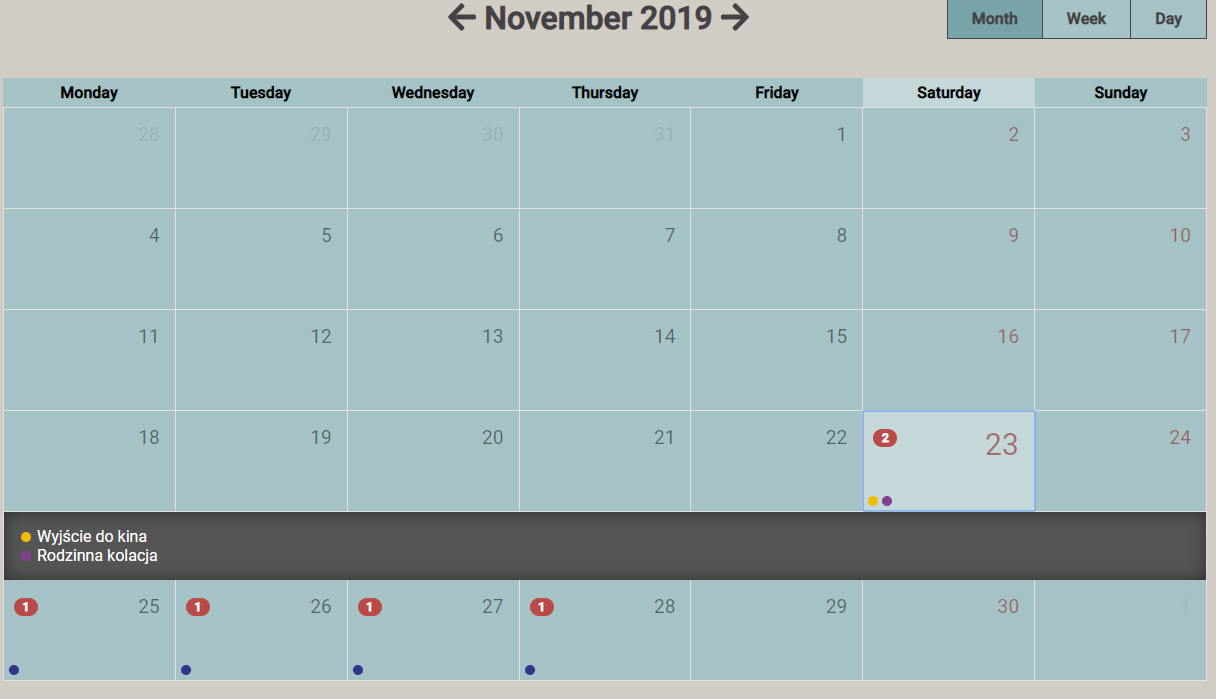
### Kalendarz

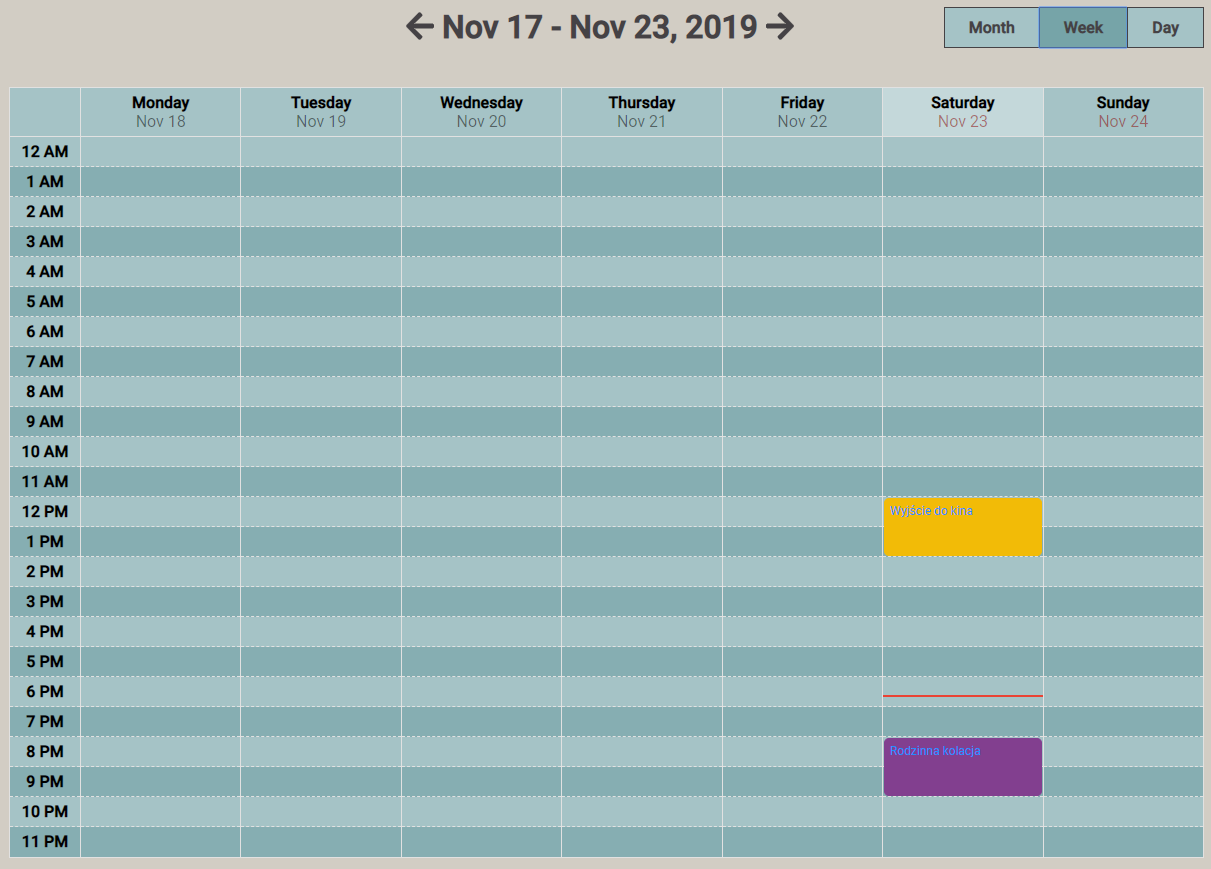
Do wspólnego kalendarza można dostać się przez wybranie zakładki **Calendar**. Umożliwia on dodawanie nowych wydarzeń (ang. Event), również tych trwających parę dni, a następnie przeglądanie ich w widoku miesięcznym, tygodniowym oraz dniowym. Jak widać na rysunku 4.15 (widok miesięczny), wydarzenia zaznaczane są na kalendarzu kolorowymi kropkami o kolorach takich jak w tabelce na dole strony. Dodanie nowego wydarzenia następuje po naciśnięciu przycisku **Add new**. Możemy je następnie edytować i zapisać zmiany przyciskiem **Apply** lub usunąć przyciskiem **Delete**. Jeśli event trwa przez parę dni, na każdym dniu w kalendarzu pojawi się o nim informacja. Przeglądany okres może być zmieniany przez strzałki nad kalendarzem.

Po naciśnięciu na dzień, w którym jest już zaplanowane jakieś zdarzenie, wyświetla się ich lista z nazwami (Rys. 4.16). Wydarzenia z tej listy można przeciągać na inny dzień, by zmienić ich dzień startu. Kalendarz można również przełączać przyciskami w prawym górnym roku na widok miesięczny (Rys. 4.15), tygodniowy (Rys. 4.17) i dzienny (Rys. 4.18).

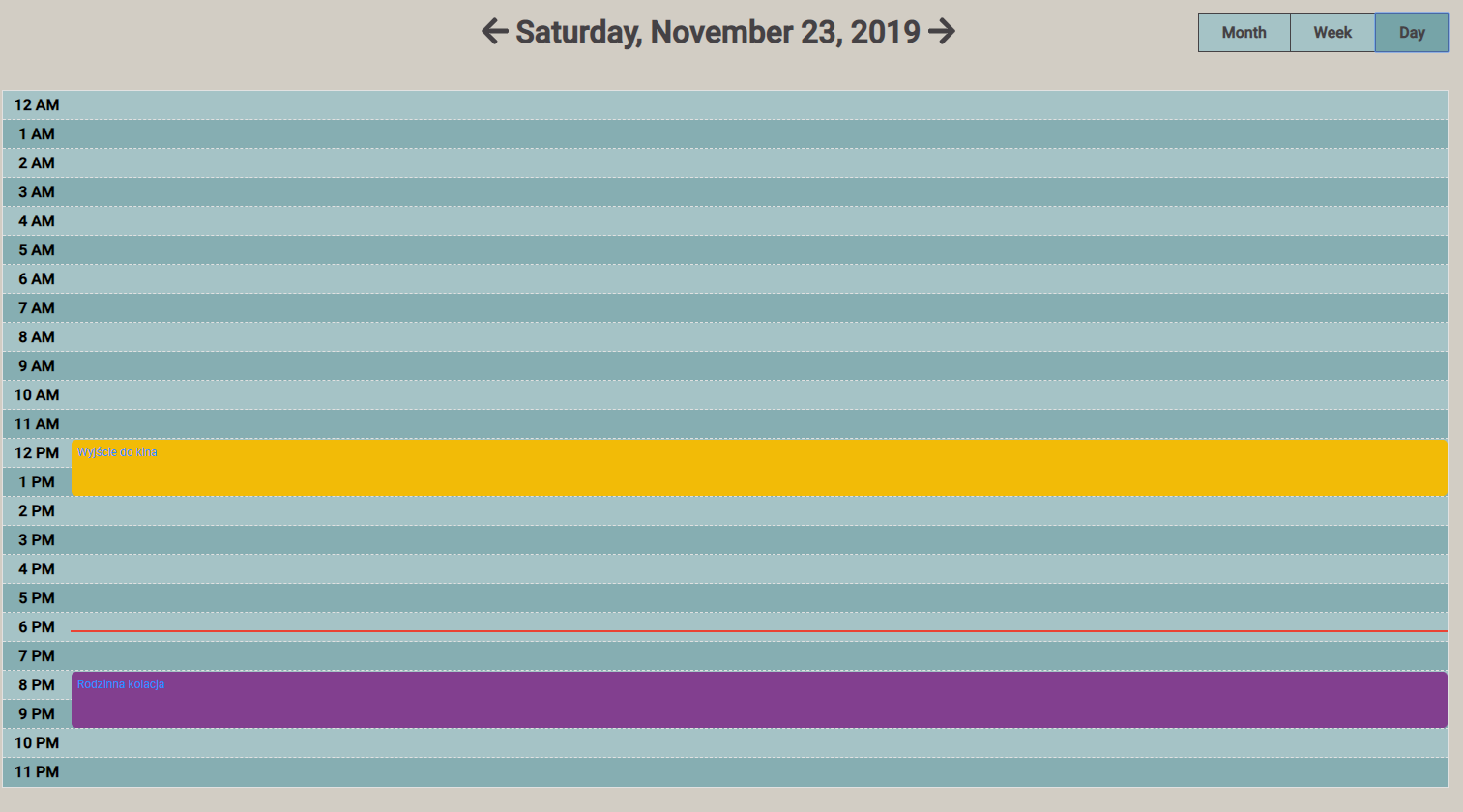


Rys.4.15. *Kalendarz - widok miesięczny*

Rys.4.16. *Lista wydarzeń w dany dzień*



Rys.4.17. *Kalendarz – widok tygodniowy*

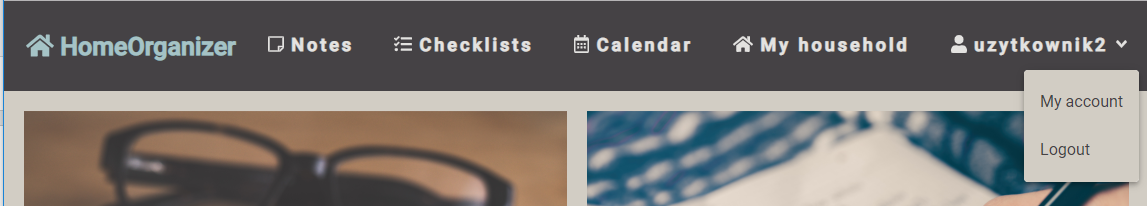


Rys.4.18. *Kalendarz – widok dzienny*

Zarówno w widoku dziennym jak i tygodniowym, wszystkie wydarzenia mogą być skracane/wydłużane przez przeciągnięcie górnej lub dolnej krawędzi bloczka, jak również przeciągane na inne godziny lub dni.

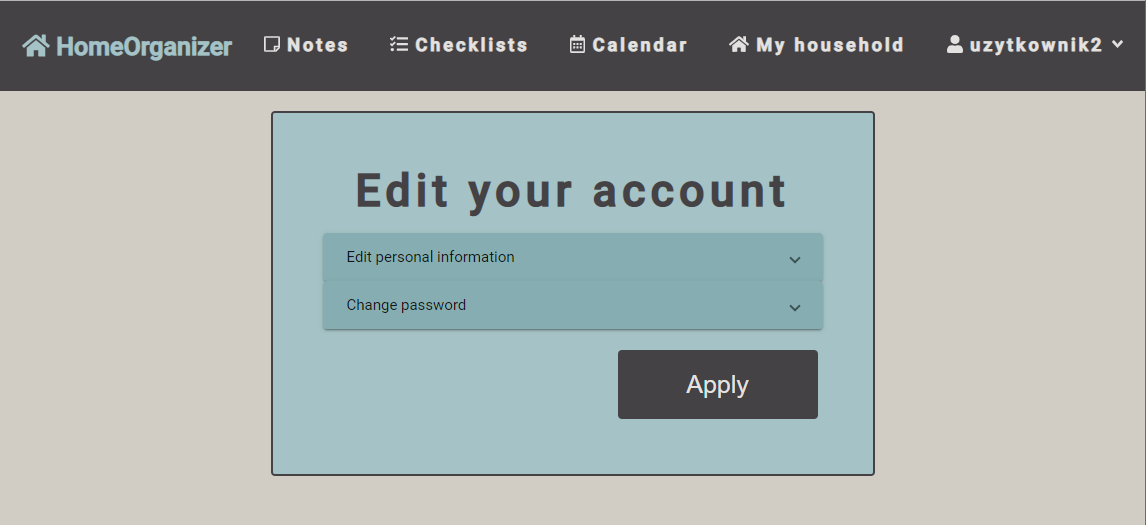
### Edycja konta i wylogowanie

Użytkownik w panelu edycji może zmienić wszystkie swoje dane, łączenie z hasłem. By dostać się do tej zakładki, należy nacisnąć na swoją nazwę użytkownika w prawym górnym rogu i z listy rozwijanej wybrać pozycje **My account** (Rys. 4.19). W liście tej znajduje się również pozycja **Logout** umożliwiająca wylogowanie się użytkownika.



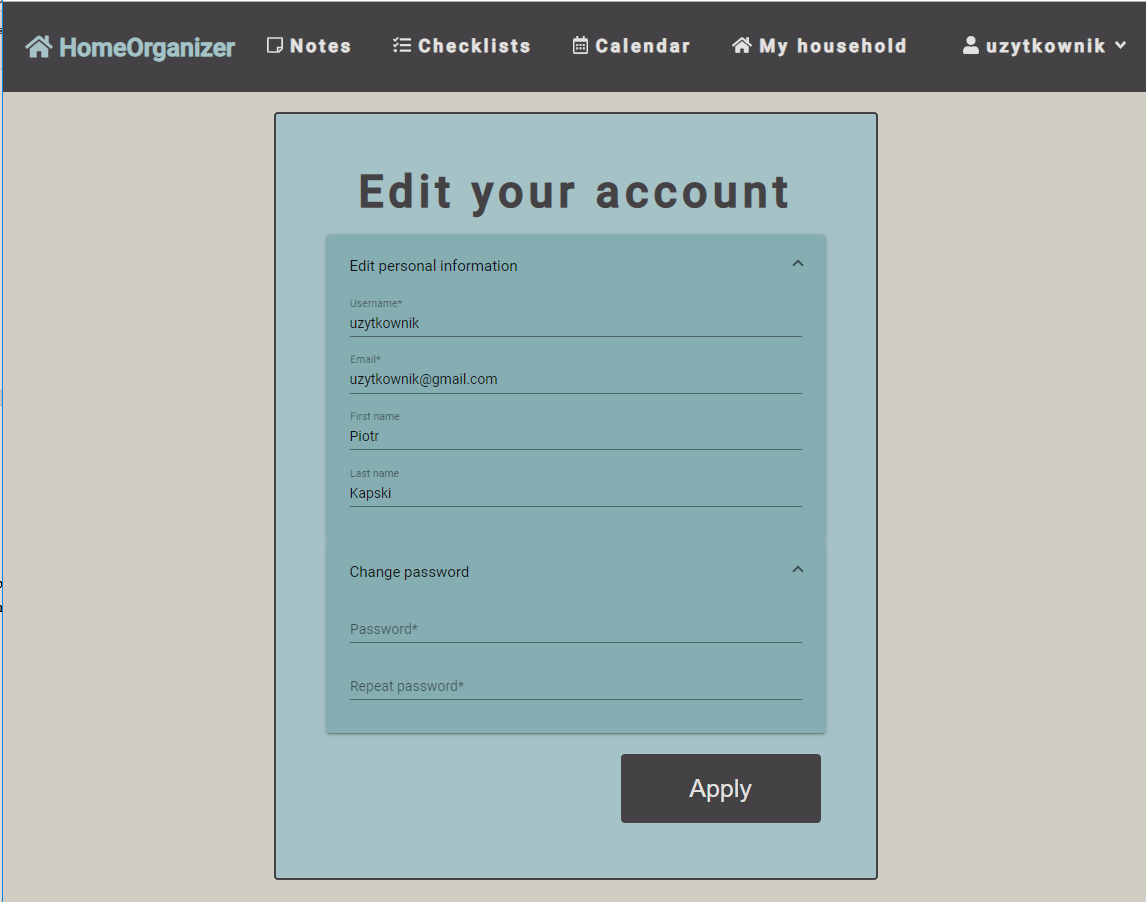
Rys.4.19. *Lista rozwijana do edycji i wylogowania się*

Po wybraniu opcji edycji pojawi się okno podzielone na dwie kategorie: edycja danych personalnych oraz zmiana hasła, jak na rysunku 4.20. Mimo iż są one oddzielone, możliwa jest jednoczesna zmiana danych w obydwu kategoriach.



Rys.4.20. *Okno edycji danych – zamknięte*

Po rozwinięciu zakładek, możliwa jest edycja danych (Rys. 4.21). Należy pamiętać, że przy edycji trzeba zachować wszystkie reguły jak przy rejestracji, np. nazwa użytkownika musi mieć minimum 5 znaków i być unikalna. Po uzupełnieniu wszystkich pożądanych pól, konieczne jest potwierdzenie operacji przyciskiem **Apply**. Jeśli wystąpił błąd przy edycji, zostanie wyświetlony odpowiedni komunikat. W przeciwnym wypadku, na dole strony pojawi się potwierdzenie sukcesu.



Rys.4.21. *Okno edycji danych – otwarte*

# Specyfikacja wewnętrzna

Projekt podzielony jest na 2 aplikacje:

* Serwer aplikacji
* Aplikacja kliencka

## Serwer aplikacji

Główna logika całego systemu, reaguje na żądania dostarczane przez klienta i zwraca odpowiedź za pomocą RESTful API. Aplikacja jest napisana w języku Java i korzysta z platformy programistycznej Spring. Zawiera ona liczne projekty, lecz wykorzystywane w programie były tylko: Spring Boot, Spring Security oraz Spring Data. Do automatyzacji budowy programu został użyty Maven. Całość pisana była zgodnie z zasadami czystego kodu na podstawie książki Clean Code Roberta C. Martina [14]. Serwer podzielony jest na następujące moduły:

* **Api** – moduł zawiera API oraz encje wygenerowane przez Swaggera. Znaleźć można tu jedynie interfejsy, ich implementacja znajduje się w następnym opisanym module.
* **Logic** – zawiera całą logikę serwisów. Jest to miejsce, w którym przeprowadza się walidacj oraz tworzone są zapytania do bazy danych, które są następnie wywoływane przez repozytoria  
  MongoRepository by uzyskać lub zapisać dane.
* **App** – moduł zawierający klasę uruchamiającą serwer. Znajduje się tu także plik application.yaml konfigurujący niektóre funkcje aplikacji takie jak port czy URI (uniform resource identifier) bazy danych.
* **Security** – zawiera wszystkie potrzebne klasy do poprawnego zabezpieczenia aplikacji. Dokładniejszy opis znajduje się   
  w rozdziale 5.1.2.

### Spring Boot

Spring Boot - pozwala na szybką inicjalizacje całego projektu wraz   
z potrzebnymi zależnościami. Podczas implementacji oferuje szeroką gamę usprawnień i metod optymalizacji ilości kodu. Dzięki niemu całkowicie zrezygnowano z konfiguracji za pomocą XML na rzecz kodu Javy. Spring Boot zapewnia również wbudowany kontener aplikacji webowych Tomcat, co znacząco przyspiesza proces przygotowania projektu. Do rozpoczęcia programowania, wystarczy adnotacja nad metodą main - @SpringBootApplication oraz projekt stworzony przez inicjalizator Springa [12].

### Bezpieczeństwo

Bezpieczeństwo serwera zapewniane jest przez Spring Security wraz z JWT (Java Web Token). Odpowiadają one za zabezpieczenie całego API przez nieautoryzowanym dostępem. Klient w celu uzyskania danych musi najpierw wysłać zapytanie o pozwolenie wraz ze swoimi   
kwalifikacjami – w tym przypadku loginem i hasłem. Jeśli są one poprawne, zostanie zwrócony token, który będzie jednoznacznie identyfikował użytkownika. Klient przy każdej próbie pozyskania danych musi podać otrzymany token w nagłówku zapytania, by się zautoryzować. W przeciwnym razie, serwer odrzuci żądanie i zwróci błąd 403 – zasób zakazany. Główny fragment konfiguracji przedstawiony jest na listingu 5.1. W linii 7,8 oraz 9 zauważyć można ścieżki oraz metody, które nie podlegają autentykacji. Są to m.in. miejsca w których użytkownik nie jest jeszcze zalogowany. Linie 12 oraz 13 określają filtry wymagane do zarówno autoryzacji, jak i autentykacji.

Listing 5.1 *Konfiguracja Spring Security*

1 @Override

2 protected void configure(HttpSecurity http) throws Exception {

3 http

4 .cors().and()

5 .csrf().disable()

6 .authorizeRequests()

7 .antMatchers(AUTH\_WHITELIST).permitAll()

8 .antMatchers(HttpMethod.OPTIONS, "/\*\*").permitAll()

9 .antMatchers(HttpMethod.POST,"/users/").permitAll()

10 .anyRequest().authenticated()

11 .and()

12 .addFilter(new JwtAuthenticationFilter(authenticationManager()))

13 .addFilter(new JwtAuthorizationFilter(authenticationManager()))

14 .sessionManagement()

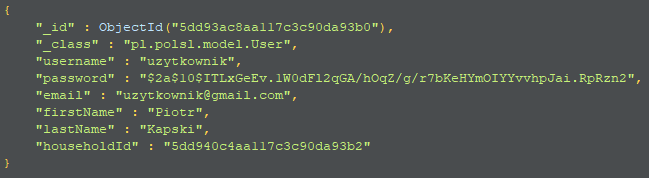
15 .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

16}

### Baza danych

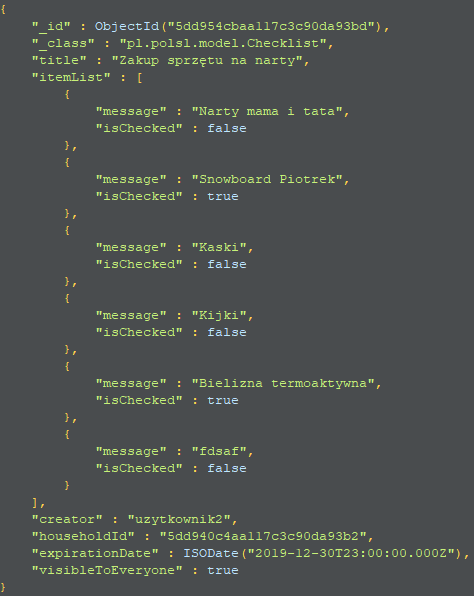
Spring Data zapewnia komunikację oraz obsługę bazy danych. Odpowiada za mapowanie obiektowo-relacyjne zasobów oraz za poprawny ich zapis oraz odczyt. Bazą danych w projekcie jest baza MongoDB, która nie posiada tabel, a przechowuje tzw. dokumenty w formacie JSON. Umożliwia to bardzo łatwe tworzenie dokumentów zagnieżdżonych np. list, co znacząco przypomina struktury używane w popularnych językach programowania. Przy dodawaniu do bazy dokumentu, nadawany jest mu unikalny identyfikator, który przechowuje w sobie również datę powstania, która jest wykorzystywana po stronie klienta. Warto również dodać, że dokumenty są określonego typu. Baza danych aplikacji składa się z pięciu kolekcji:

* **Users** – zawiera dokumenty klasy User, opisujące dane użytkownika. Składa się z następujących pól: nazwa użytkownika, hasło (zaszyfrowane funkcją haszującą BCrypt), e-mail, imię, nazwisko oraz id domostwa. Przykładowy dokument przedstawiony jest za rysunku 5.1.

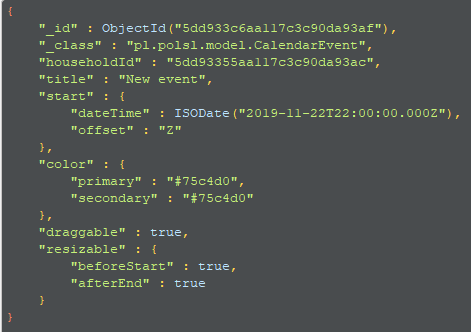


Rys.5.1. *Przykładowy dokument użytkownika w bazie MongoDB*

* **Notes** – dokumenty są notatkami stworzonymi przez użytkownika w aplikacji. Każda notatka zawiera tytuł, wiadomość, pseudonim twórcy, pseudonim odbiorcy, datę wygaśnięcia, id domostwa oraz wartość binarną określającą czy wiadomość jest widoczna dla wszystkich, czy tylko dla odbiorcy. Data utworzenia notatki nie jest wymaganym polem, ponieważ wartość tę można uzyskać z id encji.
* **Checklists -** kolekcja ta zawiera zdefiniowane przez użytkownika listy. Dokumenty w niej występujące zawierają te same pola co notatki, jednak zamiast wiadomości zawierają obiekt ItemList. ItemList jest listą obiektów typu Item, które składają się z dwóch pól: wiadomości oraz wartości typu boolean – isChecked określającej czy dany przedmiot jest skreślony czy nie. Na przykładzie tej encji można dostrzec przydatność tworzenia dokumentów zagnieżdżonych. Przykład takiego obiektu obrazuje rysunek 5.2.

Rys.5.2. *Przykładowy dokument typu Checklist w bazie MongoDB*

* **CalendarEvents** – dokumenty znajdujące się w tej kolekcji reprezentują pojedyncze wydarzenie wpisane do kalendarza. Każdy wpis zawiera 3 obiekty zagnieżdżone określające parametry danego wydarzenia. Pola przedstawione na rysunku 24 to kolejno: id dokumentu, klasa dokumentu, id domostwa, tytuł wydarzenia, czas startu składający się z czasu oraz strefy czasowej, kolor składający się z koloru głównego i drugorzędnego (w aplikacji te kolory są tożsame), wartość określająca czy wydarzenie można przesuwać kursorem oraz obiekt określający możliwości zmiany godzin przez zmianę rozmiaru bloku wydarzenia. Przez możliwość zagnieżdżania dokumentów, wpis będzie bezpośrednio mapowany do przygotowanej w Angularze klasy o tych samych polach, co znacznie upraszcza wczytywanie obiektów z bazy danych.

Rys.5.3. *Dokument klasy CalendarEvent w bazie MongoDB*

* **Households –** przechowuje informacje o zdefiniowanych w systemie domostwach. Dokumenty składają się tylko z nazwy oraz opisu, ponieważ householdy są rozpoznawane głównie po identyfikatorze.

### REST API

Interfejs programistyczny aplikacji został napisany w stylu architektonicznym REST (Representational State Transfer). Polega on na tworzeniu tzw. punktów końcowych (ang. Endpoint), które oczekują na zapytania pochodzące od klienta. Klient wysyła do API zapytanie najczęściej w postaci protokołu HTTP, a serwer zwraca na nie odpowiedź w postaci danych i/lub kodu odpowiedzi.

Do generowania API w projekcie używane było narzędzie Swagger Codegen. Umożliwia ono generowanie zarówno endpointów jak i encji na podstawie plików o rozszerzeniu yaml. Fragment takiego pliku przedstawiony jest na listingu 5.2. W linii 2 zdefiniowana jest ścieżka do endpointu,   
a w linii 3 – metoda http. Linie 8-24 to zdefiniowane parametry, które można podać w zapytaniu, natomiast linie 26-31 to możliwe odpowiedzi.

Listing 5.3 przedstawia przykładowy model, z którego wygenerowana zostanie pełna encja pełniąca role odpowiedzi na zapytanie. Można w nim definiować takie parametry jak maksymalny rozmiar tekstu czy wymagane pola. Generowany kod realizuje wzorzec delegacji, w którym kontroler zamiast wykonywać operacje, deleguje wykonanie ich delegatowi. Delegat w tym przypadku jest interfejsem, który implementujemy w module Logic. Dzięki temu stworzony punkt końcowy jest zupełnie oddzielony od implementacji. Wadą użycia narzędzia do generowania jest fakt, że tworzona jest duża ilość niepotrzebnego kodu, jednak znacznie przyśpiesza to implementacje API.

Listing 5.2. *Przykładowy endpoint zdefiniowany w pliku yaml*

1 paths:

2 /notes:

3 get:

4 summary: Gets notes

5 operationId: getNotes

6 description: Return list of notes

7 parameters:

8 - name: username

9 in: query

10 type: string

11 - name: householdId

12 in: query

13 type: string

14 - name: sortingDirection

15 in: query

16 type: string

17 enum:

18 - ASC

19 - DESC

20 - name: sortedField

21 in: query

22 type: string

23 - $ref: 'commons.yaml#/parameters/firstResult'

24 - $ref: 'commons.yaml#/parameters/maxResult'

25 responses:

26 200:

27 description: List of notes

28 schema:

29 $ref: '#/definitions/NotesPaging'

30 500:

31 $ref: 'commons.yaml#/responses/Standard500ErrorResponse'

Listing 5.3. *Przykładowy model zdefiniowany w pliku yaml*

1 definitions:

2 Household:

3 required:

4 - name

5 properties:

6 id:

7 description: Household Id providing uniqueness in database

8 type: string

9 readOnly: true

10 name:

11 description: Name of the household, doesn't have to be unique

12 type: string

13 maxLength: 20

14 description:

15 description: Description of a household

16 type: string

17 maxLength: 250

## Aplikacja kliencka

Wygląd całego projektu napisany został z wykorzystaniem frameworku Angular. Większość gotowych komponentów interfejsu takie jak przyciski, tablice, czy wyskakujące okna zostały wzięte z biblioteki Angular Material [10]. Komponenty te są proste w obsłudze, posiadają dobrą dokumentację,   
a zarazem ogromne możliwości. Wszystkie elementy były jednak zmieniane, by dopasować się motywem kolorystycznym do reszty strony. Kolory oraz właściwości poszczególnych elementów programu ustalane były w plikach   
z rozszerzeniem scss. Są to pliki w pełni kompatybilne ze składnią CSS, jednak rozszerzone o dodatkowe funkcjonalności. Cała struktura strony opisywana jest przez pliki HTML, natomiast skrypty tworzone są w TypeScript, jako że jest to obowiązkowy format plików od wersji Angular 2. Aplikacja podzielona jest na następujące komponenty:

* Account – odpowiada za widok edycji danych.
* Calendar – komponent odpowiadający za kalendarz oraz wydarzenia z nim związane. Do stworzenia kalendarza została wykorzystana biblioteka angular-calendar, która znacząco przyśpieszyła proces implementacji [[8](#_Bibliografia)]. Wadą takiego rozwiązania była sztywno ustalony sposób użycia i małe możliwości modyfikacji.
* Checklists – panel zarządzania listami. Zawiera w sobie również komponent AddChecklistDialog, który jest oknem dialogowym służącym do dodawania nowej listy (Rys. 4.14).
* Header – nagłówek całej aplikacji, jest widoczny w każdym widoku i odpowiada za nawigacje. Składa się on głównie z napisów zawierających parametr routerLink, które przenoszą użytkownika we wskazane miejsce. Przykładem może być tekst  
  <a routerLink=”/notes”>Notes</a>, który po naciśnięciu przenosi użytkownika do zakładki z notatkami.
* Households – główny komponent do zarządzania domostwem. Składa on się z kilku mniejszych części:

1. NewHousehold – widok tworzenia nowego domostwa, rysunek 4.6
2. JoinHousehold – widok dołączania do już istniejącego domostwa, rysunek 4.7
3. MyHousehold – informacje o aktualnym domostwie (Rys. 4.8). Głównym elementem tego komponentu jest lista członków, podzielona na strony za pomocą mat-paginatora. Obok identyfikatora stworzony jest także przycisk   
   w kształcie kartki, umożliwiający skopiowanie go do schowka. Kod kopiowania przedstawiony jest na   
   listingu 5.3. W liniach 2-7 tworzony jest obszar tekstowy, który inicjalizowany jest przez id. Kolejnym krokiem jest dodanie go do dokumentu (linia 8), wybranie go (linia 9 i 10), a następnie wykonanie komendy copy. Po operacji, blok tekstu jest usuwany.

Listing 5.4. *Kod kopiowania tekstu do schowka*

1 copyId(id: string) {

2 const selBox = document.createElement('textarea');

3 selBox.style.position = 'fixed';

4 selBox.style.left = '0';

5 selBox.style.top = '0';

6 selBox.style.opacity = '0';

7 selBox.value = id;

8 document.body.appendChild(selBox);

9 selBox.focus();

10 selBox.select();

11 document.execCommand('copy');

12 document.body.removeChild(selBox);

13 }

1. ConfirmationDialog – okno dialogowe, pojawiające się przy próbie opuszczenia householdu. Po potwierdzeniu chęci wyjścia z domostwa, wiadomość ta jest przesyłana do komponentu MyHousehold, który przeprowadza operacje opuszczenia.

* Index – strona główna aplikacji, przedstawiona na rysunku 4.5. Składa się ona w większości z fotografii przekierowujących na odpowiednie strony. Widok został skomponowany za pomocą układu Grid, zawierającego 2 kolumny (listing 5.5). Każdy obrazek ma dzięki temu ten sam rozmiar i są zachowane równe odstępy pomiędzy nimi.

Listing 5.5. *Układ Grid strony głównej*

1.grid {

2 display: grid;

3 grid-template-columns: auto auto;

4 height: 85vh;

5 grid-row-gap: 20px;

6 grid-column-gap: 20px;

7}

* Login – komponent odpowiedzialny za logowanie użytkownika (Rys. 4.1). Widok ten jest odpowiedzialny za autentykacje użytkownika w systemie oraz zapisanie informacji w lokalnym magazynie (ang. Local storage). Przechowywane są: token autoryzacji, czas wygaśnięcia tokenu (po tym czasie użytkownik zostanie automatycznie wylogowany), nazwa aktualnie zalogowanego użytkownika oraz id domostwa, do którego uczęszcza. Ostatnia informacja jest możliwa do uzyskania przez zapytanie do serwera, lecz przez częste wykorzystanie jej, bardziej optymalne jest tymczasowe przechowywanie jej po stronie klienta.
* Notes – zarządzanie notatkami, przedstawione na rysunku 4.9. Jest to widok bardzo podobny do list, z różnicą w sposobie przedstawiania wiadomości. Tutaj również znajduje się odosobniony komponent odpowiedzialny za dodawanie nowej listy (rys.12).
* Register – formatka służąca rejestracji nowego użytkownika   
  (Rys. 4.2). Jej głównym zadaniem jest walidacja wprowadzanych danych i wyświetlanie odpowiednich błędów. Zrealizowane jest to za pomocą klasy FormGroup, która umożliwia rejestracje walidatorów do każdego pola osobno. Każde wejście sprawdzane jest więc w czasie rzeczywistym, przez co wiadomość o błędzie pojawia się od razu. Jeśli po wprowadzonych wszystkich danych nie występuje żaden błąd, po naciśnięciu przycisku **Get started!** wysyłane jest zapytanie z metodą POST o dodanie nowego użytkownika. Kod 200 zwrócony z serwera oznacza, że użytkownik został pomyślnie dodany i klient jest przenoszony na stronę logowania. W sytuacji, gdy kod jest inny, oznacza to, że wystąpił błąd, np. przy walidacji unikalności nazwy użytkownika lub emaila. Wyświetlany jest wtedy odpowiedni komunikat, a użytkownik musi zmienić wprowadzone dane.
* Snackbar – mała wiadomość na dole ekranu (przykład na rysunku 4.11). Do komponentu przekazuje się wiadomość oraz czas trwania, po czym wyświetla się on przez wcześniej określony okres.

Oprócz komponentów, w aplikacji znajdują się też liczne serwisy odpowiedzialne za wykonywanie zapytań do API. Reagują one na napotkane błędy i zwracany jest kod błędu zależny od odpowiedzi serwera. Przykład metody realizującej takie zapytanie przedstawiony jest w listingu 5.6.   
W linii 1 ustalany jest adres, pod który kierowany będzie request. Następnie ustalany jest nagłówek określający format przekazywanych danych (w tym przypadku jest to JSON). Linie 8 i 9 jest to wysłanie zapytania, natomiast linia 10 odpowiada za wyłapywanie błędów. Jeśli serwer zwróci error, zapytanie wysyłane jest ponownie. Za drugim razem, przekazywany jest on do obsługi błędów.

Listing 5.6. *Przykład zapytania HTTP do serwera*

1 baseurl = 'http://localhost:8080/users/';

2

3 httpHeader = new HttpHeaders({

4 'Content-Type': 'application/json'

5 });

6

7 postUser(user: User) {

8 return this.http.post(this.baseurl, user,{headers:

9 this.httpHeader})

10 .pipe(retry(1),catchError(this.errorHandler));

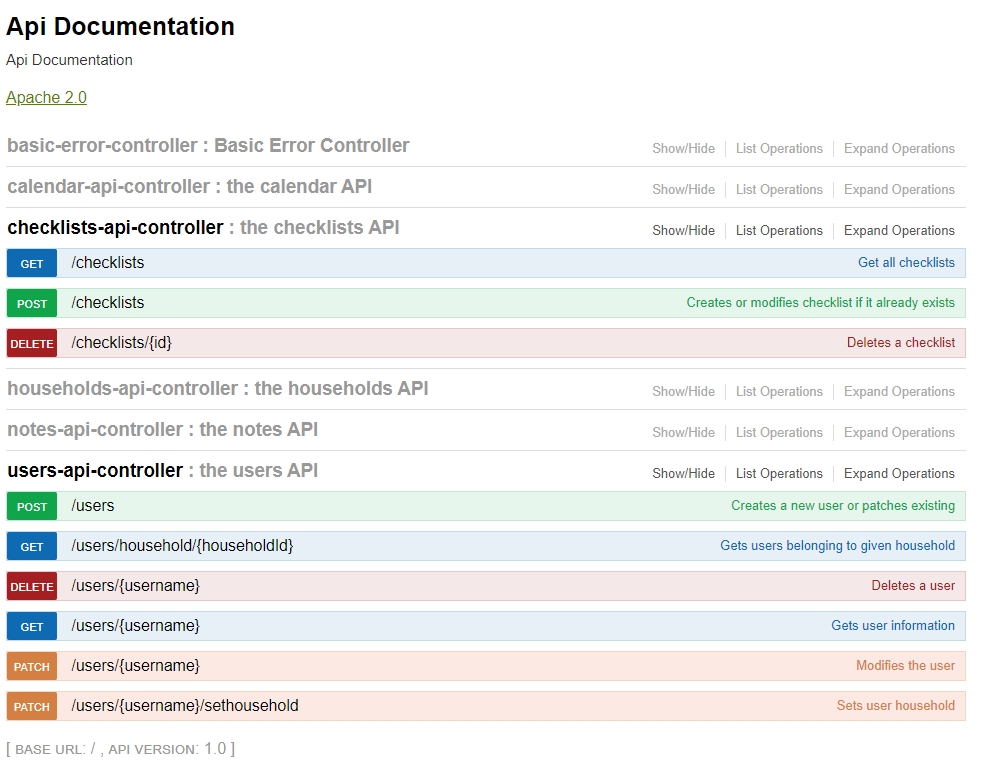
11 }

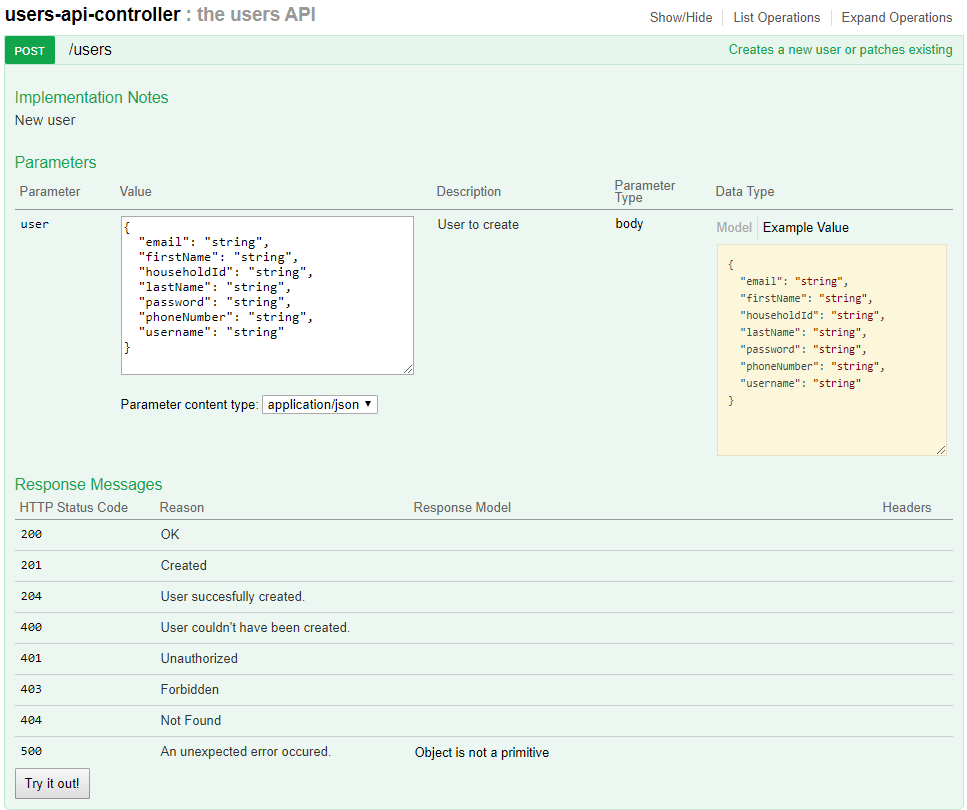
Ważnym punktem aplikacji jest interceptor autentykacji. Służy on do zmiany każdego wychodzącego zapytania w taki sposób, by do jego nagłówka dokleić token. Bez jego poprawnej konfiguracji, każdy request wymagałby manualnego jego doklejenia, co znacząco zwiększyłoby objętość kodu, jak   
i naraziło na niepotrzebne błędy.

# Weryfikacja i walidacja

Program testowany był po każdym etapie implementacji. Po stworzeniu API, wszystkie jego metody były testowane za pomocą narzędzia Postman [9]. Jest to bardzo popularne narzędzie służące do symulowania klienta API. Umożliwia ono wysyłanie zapytań, analizowanie odpowiedzi, a nawet debuggowanie. Za jego pomocą można w łatwy i szybki sposób sprawdzić poprawność działania wszystkich punktów końcowych. Umożliwia on także autoryzację, co okazało się szczególnie przydatne przy implementacji zabezpieczeń.

Narzędziem, które najlepiej spełniało funkcję testowania była dokumentacja wygenerowana przez Swaggera (Rys. 6.1). Umożliwia ona na przegląd całego napisanego API, w tym ścieżki do poszczególnych endpointów, wymaganych parametrów oraz możliwych odpowiedzi. Jednak najistotniejszą funkcją z punktu widzenia testowania, jest tworzenie przez dokumentacje przykładowego ciało wysyłanego obiektu np. przy tworzeniu użytkownika. Przykład takiego modelu przedstawiony jest na rysunku 6.2. Zawiera on wszystkie pola encji, wraz z ich przykładowymi wartościami. Przyciskiem **Try it out!** znajdującym w lewym dolnym rogu można wykonać zapytanie z podanym ciałem pod wybrany punkt końcowy oraz otrzymać jego odpowiedź. Jest to niezwykle przydatne narzędzie, ponieważ pozwala w bardzo szybki sposób przetestować działanie API losowymi danami. Przy bardziej skomplikowanych obiektach, stworzenie ciała zapytania w formacie JSON może być bardzo czasochłonne, dlatego możliwość, że Swagger zrobi to sam, jest bardzo pomocna.

Rys.6.1. *Dokumentacja wygenerowana przez narzędzie Swagger*

Rys.6.2. *Przykładowy model w dokumentacji Swagger*

Wykryto w ten sposób wiele błędów dotyczących API oraz encji. Częstym błędem była wadliwa generacja na skutek niepoprawności składni plików yaml. Pliki te nie mają tak zaawansowanej kontroli pisowni, przez co wiele nieprawidłowości jest akceptowanych przez kompilator i powodują one błędy dopiero po generacji. Testowana była także napisana w serwisach logika. Pozwoliło to wyeliminować takie potknięcia jak błędne zapytania do bazy danych, czy źle zaimplementowana paginacja.

Ostatnią metodą sprawdzania poprawności napisanego kodu było pisanie testów. Były to testy integracyjne, ponieważ potrzebowały niektórych części środowiska np. bazy danych. Sprawdzały one podstawowe funkcje serwisów, które najczęściej opierały się na operacjach związanych z Mongo. Przykładowy plik z testami przedstawiony jest na listingu 6.1. Są to testy dotyczące notatek  
i sprawdzają zapis, odczyt oraz usuwanie z bazy danych. Testy były pisane tak, aby były jak najmniej zależne od siebie nawzajem. Skutkuje to jednak często zduplikowanymi fragmentami kodu (zapis notatki do bazy w każdym teście). Uruchomienie testów tego typu po wprowadzeniu zmian pozwala na szybką weryfikację, czy aktualizacja nie zepsuła podstawowych funkcji aplikacji.

Listing 6.1 *Przykładowe testy integracyjne w aplikacji*

1 private static final String TEST\_USER = "Test";

2 private static final String TEST\_TITLE = "Test title";

3 private static final String TEST\_MESSAGE = "Test msg";

4

5 @Autowired

6 private NotesService service;

7

8 @Test

9 public void shouldSaveNoteToDBAndReturnId() {

10 Note note = getNote();

11 Assertions.assertNotNull(service.saveNote(note));

12 }

13

14 @Test

15 public void shouldReturnListOfNotesFromDB() {

16 service.saveNote(getNote());

17 NotesPaging notes = service.getFilteredNotesPaging(TEST\_USER, null, null, null, null, null);

18 Assertions.assertTrue(notes.getMaxItems() > 0, "Notes maxItems wrongly evaluated!");

19 Assertions.assertTrue(notes.getArray().size() > 0, "Notes list is empty!");

20 }

21

22 @Test

23 public void shouldDeleteNoteFromDBAndReturnTrue(){

24 String id = service.saveNote(getNote());

25 Assertions.assertTrue(service.deleteNote(id));

26 }

27

28 private Note getNote() {

29 Note note = new Note();

30 note.setTitle(TEST\_TITLE);

31 note.setMessage(TEST\_MESSAGE);

32 note.setCreator(TEST\_USER);

33 return note;

34 }

# Podsumowanie i wnioski

Jako rezultat wielotygodniowej pracy postał program spełniający wszystkie zdefiniowane wcześniej cele oraz wymagania funkcjonalne i niefunkcjonalne opisane w rozdziale 3.1 oraz 3.2. Aplikacja jest w pełni działająca i realizuje wszystkie funkcjonalności, które były w niej przewidziane. Efektywniejsze organizowanie wydarzeń w domu przyczyniać się może większej ilości czasu oraz lepszej organizacji. Ograniczy to również ilość zużywanego papieru. Dodatkowym celem projektu był samorozwój autora i nauka nieznanych wcześniej technologii, co zostało również osiągnięte. Projekt pokazał jak ważna jest analiza i sam proces projektowy. Uzyskane doświadczenie pozwoli na efektywniejsze realizowanie tych obszarów oraz pisanie kodu na wyższym poziomie.

## Kierunki dalszych prac i możliwości poprawy

Serwer aplikacji został stworzony w sposób umożliwiający łatwe rozszerzanie i modyfikacje. Dodawanie kolejnych funkcjonalności opiera się na dodaniu kolejnych punktów końcowych oraz stworzenie nowej zakładki po stronie klienta. Z tego powodu, różne elementy mogą być tworzone niezależnie, z możliwością pracy grupowej. Kolejnym krokiem przy rozwijaniu aplikacji byłoby stworzenie systemu zarządzania budżetem domowym, wraz   
z jego analizą. Jest to temat dobrze wpasowujący się w założenia projektowe, jak i potrzebny w wielu gospodarstwach domowych. W późniejszym etapie prac koniecznym byłoby stworzenie aplikacji mobilnej wykorzystującej aktualnie napisane API. Byłaby ona zsynchronizowana z aplikacją webową i umożliwiała wykonywać te same operacje. Dałoby to możliwość zarządzania domostwem z każdego miejsca, nie tylko przy ekranie komputera.

Jedną z rzeczy, które byłyby niezbędne przy wydaniu gotowego produktu, byłoby zapewnienie bezpiecznej współbieżności dla wielu użytkowników korzystających z systemu. Można to uzyskać np. przez zastosowanie optymistycznej lub pesymistycznej kontroli współbieżności. Zapewni ona to, że jeśli więcej niż jedna osoba w tej samej chwili będzie próbowała zmodyfikować ten sam zasób, zwrócony zostanie błąd. Kontrola pesymistyczna blokuje zasób za każdym razem, gdy jest używany, co mogłoby być niedopuszczalne dla tego typu aplikacji. Lepszym rozwiązaniem jest skorzystanie z kontroli optymistycznej, która dopiero przy zapisie sprawdza, czy nie został on wcześniej zmodyfikowany przez innego użytkownika. Zapewnia ona też wyższą wydajność kosztem okazjonalnego odrzucenia zmian [11]. System logowania błędów byłby również konieczny, by wychwycić wszystkie luki powstałe w systemie.

Optymalizacja jest sektorem, w którym szczególnie jest jeszcze miejsce na udoskonalenia. Dane nie są w żaden sposób przechowywane w pamięci podręcznej (ang. Cache), lecz za każdym razem pobierane z bazy danych. Sama baza w przypadku większej liczby użytkowników również wymagałaby poprawy. Dużą różnice w czasach zwracania wyników można osiągnąć tworząc indeksy, które poprawiają szybkość wyszukiwania dokumentów. W przypadku, gdy liczba użytkowników byłaby już duża i stale rosła, dobrym rozwiązaniem byłoby podzielenie bazy na mniejsze jednostki (shardy), które niezależnie obsługują żądania. Mimo że w aktualnym stadium projektu jest to niepotrzebne, warto znać sposoby poprawy wydajności, ponieważ to baza danych jest trzonem całej aplikacji.

## Problemy napotkane w trakcie pracy

Podczas implementacji oprogramowania, problemy napotyka się na porządku dziennym. Są one nieodłączną częścią programowania i to one sprawiają, że jest ono ciekawe i nieprzewidywalne. W trakcie tworzenia projektu, większość problemów w części klienckiej spowodowanych było słabą znajomością używanych tam języków. Podczas programowania serwera, głównym wyzwaniem była poprawna konfiguracja zabezpieczeń oraz wtyczki do generacji API.

Spring Security jest projektem z bogatą dokumentacją, lecz jego konfiguracja jest długa i wymaga stworzenia wielu klas konfiguracyjnych. Wiele poradników opiera się na wybudowanym przez framework mechanizmie zabezpieczenia, który domyślnie przygotowuje panel logowania, zarządza sesją, autoryzacją i autentykacją. W przypadku tego programu, widoki stworzone są w oddzielnej aplikacji, więc wszystkie wymienione elementy należy stworzyć samemu. Spring Security odpowiada wtedy tylko za autoryzacje przez zwrócenie tokenu oraz jego weryfikację przy każdym zapytaniu, co jest innym typem uwierzytelniania niż domyślna konfiguracja. Wymagane jest więc sprecyzowanie każdego etapu autoryzacji i autentykacji przez ustalenie formatu przychodzących danych, sposobu ich przetwarzania (np. porównanie nazwy użytkownika i hasła z danymi dostępnymi na bazie) oraz formatu zwracanego nagłówka. Cały proces nie jest zbyt dobrze opisany w dokumentacji, co znacząco utrudniało implementacje. Dodatkowo, odwołując się do API z innej aplikacji (to jest, korzystającej z innego protokołu, hosta lub portu), wymagane jest odpowiednie ustawienie mechanizmu   
CORS (Cross-Origin Resource Sharing), by serwer akceptował zapytania pochodzące z innego źródła (origina).

Kolejnym przeszkodą było poprawne skonfigurowanie narzędzia Swagger Codegen. W programie, było ono używane za pomocą wtyczki do Mavena   
o nazwie swagger-codegen-maven-plugin. Umożliwia ona generowanie API   
z poziomu aplikacji, jednak do poprawnej pracy, wymaga dużą ilość konfiguracji. Największą wadą, jest jednak bardzo skromna dokumentacja, która nie obejmuje wielu ważnych parametrów. Wtyczka również zawiera błąd powodujący, że w pliku pom.xml wygenerowanego API tworzony jest zbędny plugin oraz brakuje w nim zależności jaxb-api. Po generacji, wymagane jest ręczne naprawienie tych błędów, co częściowo zmniejsza wygodę używania narzędzia. Biorąc pod uwagę ilość wysiłku wymaganego do rozpoczęcia używania generatora, używanie go w małych projektach nie będzie dawało optymalizacji czasowej przy pisaniu API.

Dużym wyzwaniem podczas programowania widoków aplikacji okazało się edytowanie notatek oraz list. Zostało to wykonane przez nadawanie każdej edytowalnej części innych identyfikatorów, a następnie na podstawie nich pobranie odpowiednich elementów HTML oraz zmiana ich parametrów. Głównym zmienianym parametrem był atrybut *contentEditable*. Umożliwia on ze zwykłego bloku (div) z tekstem, uczynić element, który można edytować. W teorii wydaje się to bardzo przydatne i spełniające wymagane zadanie. W praktyce okazało się jednak, że atrybut ten posiada wiele wad i jest ciężki w obsłudze. ContentEditable ustawiony na wartość *true* tworzy z elementu edytor sformatowanego tekstu (rich text). Oznacza to, że chcąc przekopiować tekst z innego miejsca, zostanie on wklejony z jego oryginalnym formatowaniem np. rozmiarem czcionki i kolorem tła. Stanowi to problem dla estetycznego wyświetlania elementów i może w niektórych przypadkach (np. kopiując tekst z bardzo dużą czcionką) zupełnie go zepsuć. By temu zapobiec, należało w momencie pomiędzy wklejeniem, a wyświetleniem na ekran, zmienić go na zwykły tekst. Jest to tylko jeden z niewielu problemów jakie przysporzył ten atrybut, a jego często wadliwe działanie opisuje wiele artykułów, które odradzają jego używanie [12].

# Bibliografia

|  |  |
| --- | --- |
| [1] | Simon Kemp. *Digital 2019: Global Internet Use Accelerates – We Are Social*. https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates [data dostępu: 2019-11-20] |
| [2] | *Spring framework*. <https://spring.io/> [data dostępu: 2019-11-22] |
| [3] | *Angular* <https://angular.io/> [data dostępu: 2019-11-22] |
| [4] | *MongoDB* <https://www.mongodb.com/> [data dostępu: 2019-11-22] |
| [5] | *Swagger* <https://swagger.io/> [data dostępu: 2019-11-25] |
| [6]  [7] | *Robo 3T* <https://robomongo.org/> [data dostępu: 2019-11-23]  Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*, str.76-105 |
| [8] | Matt Lewis *angular-calendar documentation* <https://mattlewis92.github.io/angular-calendar/docs/> [data dostępu: 2019-11-26] |
| [9] | *Postman* <https://www.getpostman.com/> [data dostępu: 2019-11-26] |
| [10] | *Angular Material* https://material.angular.io/ [data dostępu: 2019-11-26] |
| [11] | *Pesymistyczna i optymistyczna kontrola współbieżności.* https://ninjaprogrammerblog.wordpress.com/ 2017/08/31/pesymistyczna-i-optymistyczna-kontrola-wspolbieznosci/ [data dostępu: 2019-11-27] |
| [12] | Nick Santos *Why ContentEditable is Terrible* <https://medium.engineering/why-contenteditable-is-terrible-122d8a40e480> [data dostępu: 2019-11-27] |
| [13] | *Spring initializr* <https://start.spring.io/> [data dostępu: 2019-11-27] |
| [14] | Robert C. Martin *Clean Code.* Prentice Hall, 2018 |

# Spis skrótów i symboli

|  |  |
| --- | --- |
| *DNA* | kwas deoksyrybonukleinowy (ang. *deoxyribonucleic acid*) |
| *MVC* | model – widok – kontroler (ang. *model–view–controller*) |
| *N* | Liczebność zbioru danych |

# Zawartość dołączonej płyty

Na płycie DVD dołączonej do dokumentacji znajdują się następujące materiały:

* praca w formacie pdf,
* źródła programu,
* zbiory danych użyte w eksperymentach.

# Spis rysunków

# Spis tabel