Dokumentacja

RoomMates Manager

Opracowali:

Artur Przystaś,

Bartłomiej Nawój,

Mateusz Nehrebecki

Spis treści

1. Wstępny opis projektu

Ogólne założenia projektu

Celem projektu było stworzenie aplikacji internetowej, której zadaniem było ułatwienie zarządzaniem zasobami dla współlokatorów. Aplikacja umożliwia dodawanie oraz przechowywanie informacji o rachunkach, dodawanie i organizowanie obowiązków, zarządzanie wspólnymi wydatkami oraz obliczanie zużycia mediów za pomocą dedykowanego kalkulatora. Aplikacja została napisana wg technik SPA oraz RWD.

Technologia

Python

Interpretowany język wysokiego poziomu, udostępniający wiele wymaganych przy tworzeniu rozbudowanych aplikacji funkcjonalności. Dodatkowymi zaletami jest jedna z najbogatszych baz zewnętrznych bibliotek, możliwość łatwego łączenia ich ze sobą oraz przystępna, pomocna składnia.

Flask

Jest to mikro-framework aplikacji webowych napisany w języku Python. Upraszcza on projektowanie zapewniając przejrzysty schemat łączenia adresów URL, źródła danych, widoków i szablonów. Domyślnie dostajemy również deweloperski serwer WWW, nie musimy instalować żadnych dodatkowych narzędzi typu LAMP (WAMP).

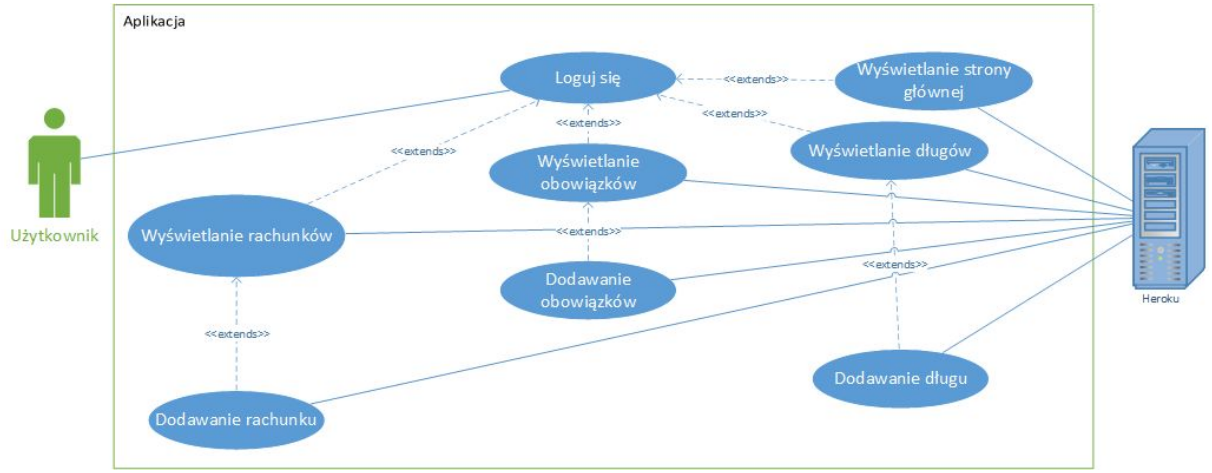
Vue.js

Vue.js to obecnie trzeci pod względem popularności framework JavaScript, służącym do budowania interfejsów użytkownika. Pozwala na tworzenie zarówno prostych komponentów, jak i zaawansowanych i skalowalnych aplikacji typu SPA (Single-Page Application).

Bootstrap

Bootstrap to biblioteka CSS wydawana na licencji MIT. Zawiera zestaw przydatnych narzędzi ułatwiających tworzenie interfejsu graficznego stron oraz aplikacji internetowych. Bazuje głównie na gotowych rozwiązaniach HTML oraz CSS (kompilowanych z plików Less) i może być stosowany m.in. do stylizacji takich elementów jak teksty, formularze, przyciski, wykresy, nawigacje i innych komponentów wyświetlanych na stronie. Biblioteka korzysta także z języka JavaScript. Biblioteka wspiera tworzenie witryn RWD.

Diagram przypadków użycia



Wymagania funkcjonalne

id	Wymaganie	Priorytet	Spełnione
FUN01	Dodawanie rachunków	Bardzo wysoki	TAK
FUN02	Wyświetlanie rachunków	Bardzo wysoki	TAK
FUN03	Dodawanie nowych zadań	Bardzo wysoki	CZĘŚCIOWO
FUN04	Wyświetlanie dodanych zadań	Bardzo wysoki	CZĘŚCIOWO
FUN05	Stworzenie dashboardu	Wysoki	NIE
FUN06	Stworzenie strony „about”	Niski	TAK
FUN07	Stworzenie kalkulatora rachunków	Wysoki	NIE
FUN08	Stworzenie funkcjonalności „długi”	Wysoki	NIE

Wymagania niefunkcjonalne

id	Wymaganie	Priorytet	Spełnione
NFUN01	Kompatybilność z najpopularniejszymi przeglądarkami	Wysoki	TAK
NFUN02	Nowoczesny wygląd strony	Średni	TAK
NFUN03	Szybki czas reakcji na zapytania	Wysoki	TAK
NFUN04	Zapewnienie przenoszalności aplikacji	Średni	TAK

Reguły biznesowe

id	Reguła
BIZ01	Aplikacja przeznaczona jest dla współlokatorów
BIZ02	Aplikacja dostępna jest przez przeglądarkę internetową
BIZ03	Serwerowa część aplikacji jest całkowicie oddzielona od części klienckiej

Podział pracy

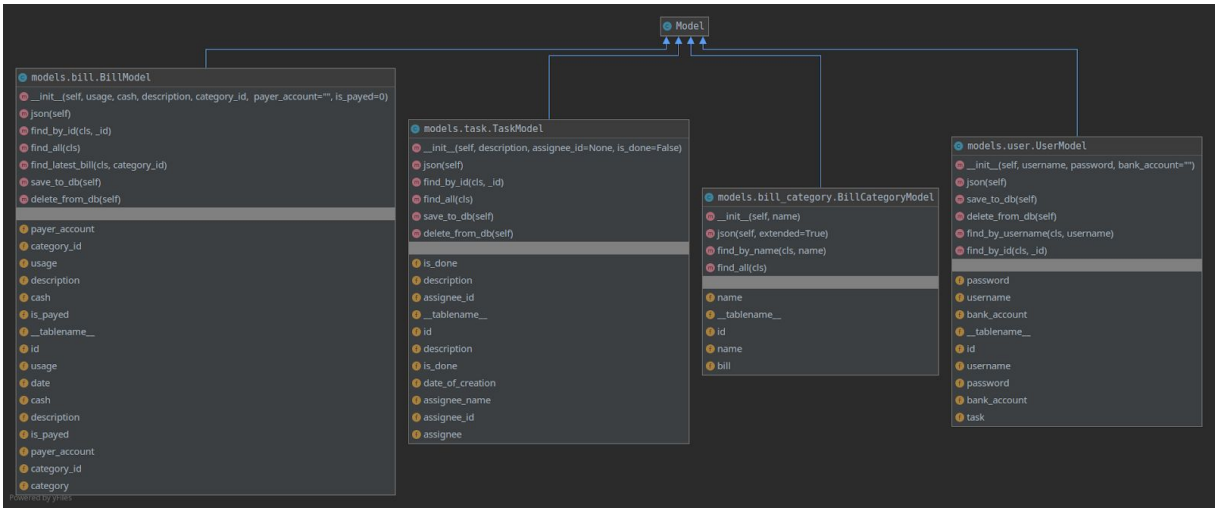
BARTŁOMIEJ NAWÓJ – zaprogramowanie części serwerowej aplikacji

ARTUR PRZYSTAŚ – zaprogramowanie części klienckiej aplikacji

MATEUSZ NEHREBECKI – zaprojektowanie wyglądu aplikacji

2. Dokumentacja projektu

Diagramy klas



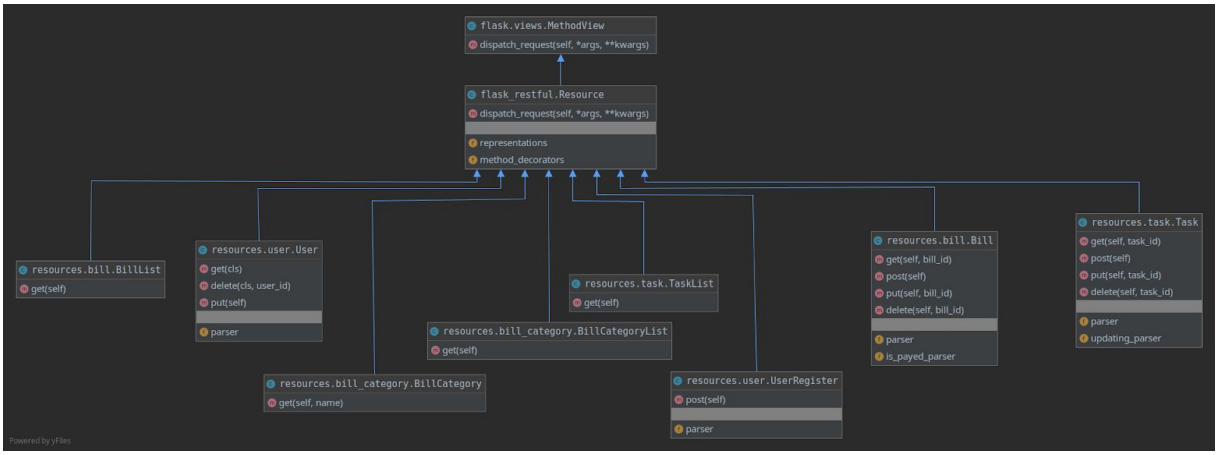
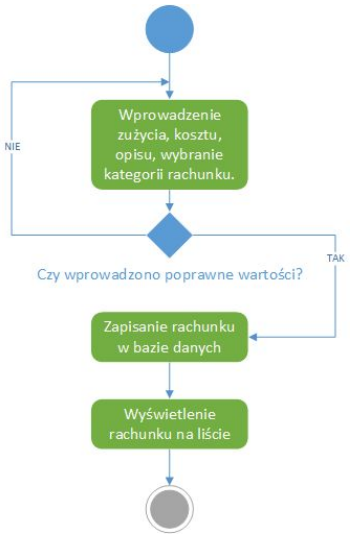


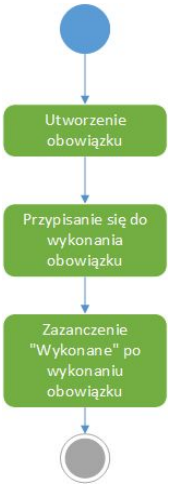
DIAGRAM AKTYWNOŚCI DLA FUNKCJONALNOŚCI

FUN01 – Dodawanie rachunku



Użytkownik aplikacji ma możliwość przejścia do ekranu rachunków poprzez naciśnięcie przycisku w panelu nawigacyjnym. Aby dodać rachunek należy wybrać kategorię, wpisać zużycie, koszt oraz opis rachunku. Po kliknięciu przycisku *Dodaj* rachunek zapisywany jest w bazie danych a następnie wyświetlany na liście rachunków.

FUN03 - Dodanie nowych zadań

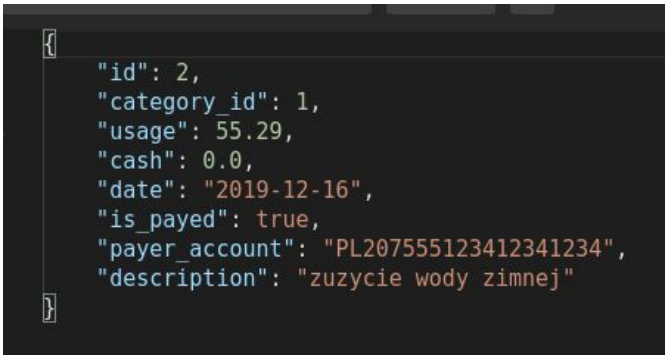


Użytkownik ma możliwość tworzenia obowiązków, które ma wykonać w mieszkaniu np. wynoszenie śmieci. Po utworzeniu obowiązku zalogowana osoba ma możliwość przypisania się do tego obowiązku. Po wykonaniu zadania należy zaznaczyć opcję „Wykonane”.

3. Analiza techniczna projektu

Część serwerowa projektu opiera się o zbudowane w mikro-frameworku Flask - REST api. Takie podejście umożliwia odseparowanie warstwy klienta od warstwy serwerowej. Zaletą takiego rozwiązania jest jego uniwersalność oraz łatwość testowania za pomocą aplikacji Postman.
Zastosowana baza danych to PostgreSQL. Do pracy z bazą i mapowania obiektowo-relacyjnego użyto Flask-SQLAlchemy (odpowiednik SQLAlchemy zmodyfikowany do pracy z frameworkiem Flask).

Klient, tj. aplikacja webowa w Vue.js, komunikuje się z API za pomocą protokołu HTTP, preparując zapytania (request) na odpowiedni adres (endpoint) np. roommates-resources-manager.herokuapp.com/bills
Następnie wysyła request do Api, po czym serwer zwraca odpowiedź. Taką odpowiedź powinna zawierać JSON z pożądanymi informacjami. Przykład zapytania typu GET w programie Postman wraz z odpowiedzią serwera:



Autentykacja

Api *"Roommates Resources Manager"* korzysta ze standardu JWT (JSON web tokens) w celu zapewnienia autentykacji. Aby otrzymać token należy wykonać POST request na endpoint {api_url}**/auth** z poprawnymi danymi logowania. Jeśli dane były poprawne, w odpowiedzi otrzymamy token dostępu (access_token). Jego ważność to 1 godzina.

Endpointy

W celu zwiększenia przejrzystości opisu pełny adres API zmieniono na {api_url}.

1. Logowanie
   Aby przejść proces logowania, należy wykonać POST request z poprawnymi danymi. Do tego celu służy endpoint {api_url}**/auth**. Ciało (ang. body) zapytania powinno być skonstruowane następująco:

```
{
        "username": "<username>",
        "password": "<password>"
}
```

   Jeśli dane logowania były poprawne, odpowiedź będzie wyglądała następująco:

```
{
        "access_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1Nzc3MjIwNDAsImlhdCI6MTU3NzcxODQ0MCwibmJmIj
oxNTc3NzE4NDQwLCJpZGVudGl0eSI6MX0.WKk13ZLXfjQqfJF5bBlrsaY7ZcK7ySQIgGWQWlce6HM"
}
```

   W przypadku podania błędnych danych serwer zwróci odpowiedź ze statusem 401 UNAUTHORIZED z następującym body:

```
{
    "description": "Invalid credentials",
    "error": "Bad Request",
    "status_code": 401
}
```

2. Rejestracja użytkownika
   Do rejestracji służy endpoint {api_url}**/register**. Aby zarejestrować nowego użytkownika należy wykonać POST request z nazwą użytkownika i hasłem. Format danych powinien być identyczny jak przy logowaniu. Jeśli dane zostały podane poprawnie, zostanie zwrócona odpowiedź:

```
{
    "message": "User created successfully."
}
```

   Jeśli istnieje użytkownik o podanej nazwie, body odpowiedzi będzie następujące:

```
{
    "message": "User with username 'admin' already exists"
}
```

3. Pobranie informacji o aktualnie zalogowanym użytkowniku
   Do tego celu należy wysłać request GET na endpoint {api_url}**/user**. W ciele odpowiedzi serwera zwracany jest następujący JSON:

```
{
    "id": 3,
    "username": "bob2",
    "bank_account": ""
}
```

   Pole *"bank_account"* aktualizuje się za pomocą PUT requestu na ten sam endpoint.

4. Pobranie listy rachunków
   Aby pobrać listę rachunków należy wykonać GET request na endpoint {api_url}**/bills**. Jeśli do zapytania nie jest dołączony poprawny JWT access_token, API zwróci odpowiedź z następującym błędem:

```
{
    "description": "Invalid header padding",
    "error": "Invalid token",
    "status_code": 401
}
```

   Jeśli załączony token będzie poprawny, body odpowiedzi będzie zawierać wszystkie rachunki.

```
{
    "bills": [
        {
            "id": 2,
            "category_id": 1,
            "usage": 45.67,
            "cash": 0.0,
            "date": "2019-12-28",
            "is_payed": false,
            "payer_account": "",
            "description": "zużycie zimnej wody"
        },
        {
            "id": 3,
            "category_id": 2,
            "usage": 0.0,
            "cash": 50.0,
            "date": "2019-12-28",
            "is_payed": true,
            "payer_account": "PL207574672341234",
            "description": "internet za grudzien"
        }
    ]
}
```

   Jeśli ID rachunku jest już znane, istnieje możliwość pobrania informacji wyłącznie o danym rachunku. Aby ją otrzymać, należy wysłać zapytanie GET na {api_url}**/bill/<bill_id>**. Odpowiedź będzie w takiej samej formie jak w przypadku endpointu {api_url}**/bills** , lecz zwrócone będą informacje tylko o interesującym nas rachunku.

5. Dodanie nowego rachunku

   Aby utworzyć nowy rachunek należy wysłać zapytanie POST na endpoint {api_url}**/bill**. W ciele zapytania należy przesłać informacje o rachunku. Przykładowe ciało takiego zapytania:

```
{
        "category_id": 1,
        "usage": 45.67,
        "cash": 0.0,
        "description": "test",
        "payer_account": ""
}
```

   Parametry *usage, cash i payer_account* są opcjonalne i nie muszą być dołączane. W odpowiedzi na takie zapytanie API zwraca utworzony rachunek:

```
{
    "id": 5,
    "category_id": 1,
    "usage": 45.67,
    "cash": 0.0,
    "date": "2019-12-30",
    "is_payed": false,
    "payer_account": "PL207574672341234",
    "description": "test"
}
```

   API zostało zaprogramowane tak, aby automatycznie dołączać datę utworzenia rachunku. Parametr rachunku *is_payed* zostaje domyślnie ustawiony na nieprawdziwy - rachunek jest nieopłacony. Jeśli użytkownik tworzący rachunek nie podał numeru konta wraz z rachunkiem, a w opcjach swojego konta ma dodany swój numer konta, API automatycznie doda numer konta użytkownika do rachunku.

   Istnieje możliwość usunięcia rachunku ze znanym ID (jeszcze nie zaprogramowana po stronie klienta). Aby z niej skorzystać należy wysłać request DEL na endpoint {api_url}**/bill/<bill_id>**. Do każdego requestu należy umieścić JWT access_token. Ciało odpowiedzi na takie zapytanie wygląda następująco:

```
{
    "message": "Bill deleted"
}
```

6. Zapłacenie rachunku - zmiana pola *is_payed*

   Aby zaktualizować pole *is_payed* rachunku o znanym ID należy wysłać zapytanie PUT na endpoint {api_url}**/bill/<bill_id>**. Body zapytania powinno wyglądać następująco:

```
{
        "is_payed": true
}
```

   W odpowiedzi zwracany jest zaktualizowany rachunek.

7. Pobranie nazw kategorii
   Do tego celu stworzono endpoint {api_url}**/bill_categories**. Aby otrzymać listę kategorii należy wysłać zapytanie GET. Odpowiedź na poprawne zapytanie wygląda następująco:


```
{
    "bill_categories": [
        {
            "id": 0,
            "name": "Ciepła woda"
        },
        {
            "id": 1,
            "name": "Zimna woda"
        },
        {
            "id": 2,
            "name": "Internet"
        }
    ]
}
```


Istnieje możliwość pobrania rachunków tylko z określonej kategorii (np. aby je filtrować), której nazwa jest znana. W tym celu należy wysłać request GET na endpoint {api_url}**/bill_category/<name>**. W ciele odpowiedzi z serwera znajduje się lista rachunków:


```
{
    "id": 2,
    "name": "Internet",
    "bills": [
        {
            "id": 3,
            "category_id": 2,
            "usage": 0.0,
            "cash": 50.0,
            "date": "2019-12-28",
            "is_payed": false,
            "payer_account": "",
            "description": "internet za grudzień"
        },
        {
            "id": 4,
            "category_id": 2,
            "usage": 0.0,
            "cash": 50.0,
            "date": "2019-12-28",
            "is_payed": false,
            "payer_account": "PL207574672341234",
            "description": "zaległy rachunek"
        }
    ]
}
```

8. Utworzenie nowego obowiązku
   Aby utworzyć nowy obowiązek należy skonstruować i wysłać zapytanie POST na adres {api_url}**/task**. Oprócz poprawnego JWT token, zapytanie musi zawierać opis tworzonego obowiązku w następującym formacie:

```
{
        "description": "wyniesienie smieci"
}
```

   Ciało odpowiedzi na zapytanie to następujący JSON:

```
{
    "id": 5,
    "assignee_name": null,
    "description": "wyniesienie smieci",
    "is_done": false,
    "date_of_creation": "2019-12-30"
}
```

   API automatycznie ustawia pole *"assignee_name"* na *null*. Pole te mówi, kto jest przypisany do obowiązku. Z chwilą utworzenia obowiązku jest on wolny, oraz niewykonany - pole *"is_done"* ma wartość domyślną ***false***. Również automatycznie dodawana jest data utworzenia obowiązku.

9. Przypisanie się do obowiązku
   Przypisać się można tylko do wolnego obowiązku o znanym ID. W tym celu należy wysłać zapytanie PUT na endpoint {api_url}**/task/<task_id>** z pustym JSON i poprawnym JWT access_token. API na podstawie przesłanego tokenu ustala ID użytkownika, który chce przypisać się do zadania. Jeśli token jest poprawny, a obowiązek nie zajęty, serwer w odpowiedzi zwraca rachunek z zaktualizowanymi danymi:

```
{
    "id": 5,
    "assignee_name": "admin",
    "description": "wyniesienie smieci",
    "is_done": false,
    "date_of_creation": "2019-12-30"
}
```

   Jeśli obowiązek jest już zajęty, API zwróci następującą odpowiedź:

```
{
    "message": "You are not assigned to this task"
}
```

10. Wykonanie obowiązku - aktualizacja pola *is_done*
    Aby zasygnalizować wykonanie obowiązku, czyli zmienić wartość pola *"is_done"*, należy wysłać request PUT na endpoint {api_url}**/task/<task_id>** z body zawierającym nową wartość pola:

```
{
        "is_done": true
}
```

Jeśli użytkownik jest przypisany do tego obowiązku (ma do niego prawa), serwer zwróci odpowiedź z zaktualizowanym rachunkiem w ciele odpowiedzi:

```
{
    "id": 5,
    "assignee_name": "admin",
    "description": "wyniesienie smieci",
    "is_done": true,
    "date_of_creation": "2019-12-30"
}
```

W przypadku błędnego zapytania - gdy użytkownik nie jest przypisany do obowiązku - serwer zwróci:

```
{
    "message": "You are not assigned to this task"
}
```

Zalogowany użytkownik może usuwać obowiązki do których jest przypisany. W tym celu należy wysłać request DEL na endpoint {api_url}**/task/<task_id>**. Odpowiedź serwera przyjmie następującą postać:

```
{
    "message": "Task deleted"
}
```

11. Pobranie listy obowiązków
    Aby pobrać wszystkie utworzone obowiązki należy wysłać zapytanie GET na endpoint {api_url}**/tasks**. W ciele odpowiedzi serwera zwracany jest następujący JSON:

```
{
    "tasks": [
        {
            "id": 4,
            "assignee_name": "bob",
            "description": "posprzątać kuchnie",
            "is_done": true,
            "date_of_creation": "2019-12-28"
        },
        {
            "id": 5,
            "assignee_name": "admin",
            "description": "wyniesienie smieci",
            "is_done": true,
            "date_of_creation": "2019-12-30"
        }
    ]
}
```

Istnieje możliwość pobrania informacji o konkretnym rachunku - analogicznie jak w rachunkach. Należy wysłać request GET na adres {api_url}**/task/<task_id>**. Ciałem odpowiedzi jest JSON z pożądanymi informacjami.