

Zaawansowane Programowanie obiektowe

Projekt

2023/2024



**POLITECHNIKA
BYDGOSKA**

im. Jana i Jędrzeja Śniadeckich



Łukasz Rydz 118849, Informatyka Stosowana, semestr V

Panel administracyjny wypożyczalni samochodów

1. Informacje o projekcie:	3
1. Główne założenia:	3
2. Dostępność:	3
3. Technologie użyte przy tworzeniu aplikacji:	3
1. Backend:	3
2. Frontend:	3
2. Użyte technologie w projekcie po stronie serwera.	4
1. Python:	4
2. Firebase:	4
3. Flask:	5
2. Użyte technologie w projekcie po stronie klienta:	5
1. Biblioteka projektowa:	5
2. Stylowanie:	6
3. Komunikacja:	6
4. Struktura komponentów aplikacji ReactJS	6
2. Widoki w aplikacji:	7
1. Uwierzytelnianie:	7
2. Notatki:	8
3. Nawyki:	9
3. Aktywności:	10
4. Ustawienia:	11
3. Aplikacja serwerowa:	12
1. Konfiguracja aplikacji Flask oraz konfiguracja firebase:	12
2. Przykładowe „route’y”:	12
3. Klasy:	13
a. Klasa Activities:	13
b. Klasa Habits:	14
c. Klasa Notes:	15

1. Informacje o projekcie:

1. Główne założenia:

Aplikacja HabitsHub ma za zadanie ułatwić użytkownikowi zarządzanie swoim wolnym czasem dzięki możliwości dodawania aktywności w podanym terminie oraz możliwości dodawania oraz śledzenia nawyków które chce wykonywać codziennie. Użytkownik dzięki funkcjonalności notatnika może zapisywać swoje osiągnięcia bądź po prostu notować jakieś ważne dla niego informacje. Aplikacja połączona jest z serwerem który przetwarza dane i zapisuje je w bazie danych „Firebase” dzięki czemu użytkownik nie musi zapisywać danych na swoim urządzeniu oraz nie musi się martwić o ich utratę.

2. Dostępność:

Aplikacja będzie dostępna na każdym urządzeniu podłączonym do sieci oraz posiadającej przeglądarkę internetową.

Dzięki w pełni funkcjonalnej responsywności aplikacja będzie czytelna na każdym urządzeniu użytkownika.

Do budowy aplikacji zostaną użyte funkcjonalności które są wspierane niemal przez wszystkie wersje przeglądarek.

3. Technologie użyte przy tworzeniu aplikacji:

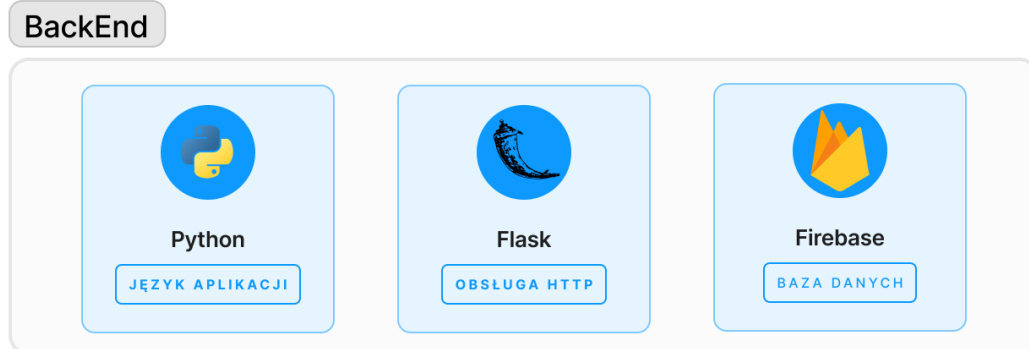
1. Backend:

- Python
- Flask
- Firebase
- Pyrebase
- Inne podstawowe biblioteki dostępne dla języka Python.

2. Frontend:

- JavaScript
- Sass
- HTML
- ReactJS

2. Użyte technologie w projekcie po stronie serwera.



1. Python:



Rysunek 1 - Python

1. Jako język programowania aplikacji serwerowej wybrałem Python'a. Wybrałem ten język z następujących powodów:
 - a. **Prostota i Czytelność Kodu:** Python jest znany ze swojej prostoty składniowej i czytelności kodu, co sprawia, że jest łatwy do zrozumienia.
 - b. **Integracja z Firebase:** Python oferuje biblioteki i narzędzia umożliwiające łatwą integrację z Firebase. To pozwala na efektywne korzystanie z usług Firebase w backendzie Twojej aplikacji.
 - c. **Dobra Wydajność:** Python jest językiem o dobrej wydajności, a dla wielu zastosowań w dziedzinie web developmentu jego osiągi są wystarczające.

2. Firebase:



Rysunek 2 – Firebase

2. Jako system do zarządzania i przechowywania danych wybrałem Firebase ponieważ:
 - a. **Szybki Deployment:** Firebase dostarcza proste narzędzia do szybkiego wdrażania aplikacji. Możesz łatwo hostować swoje aplikacje, zarządzać bazą danych i korzystać z innych funkcji bez konieczności zarządzania infrastrukturą.

- b. **Autentykacja i Zarządzanie Użytkownikami:** Firebase ma wbudowane narzędzia do autentykacji użytkowników, co pozwala łatwo integrować system uwierzytelniania w swojej aplikacji.

3. Flask:

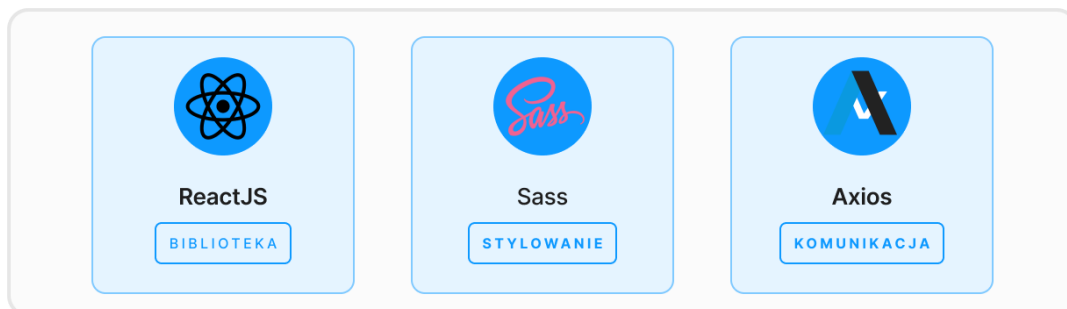


Rysunek 3 – Flask

- 3. Do obsługi żądań HTTP użyłem mikroframework'a Flask, ponieważ:
 - a. **Lekkość i Prostota:** Flask jest mikroframeworkiem, co oznacza, że dostarcza podstawowe funkcje niezbędne do budowy aplikacji webowych, ale jednocześnie jest prosty i nie narzuca zbyt wielu struktur i zależności.
 - b. **Dobra Dokumentacja:** Flask ma dobrą dokumentację, co ułatwia zarówno naukę, jak i rozwijanie aplikacji. Społeczność Flask jest aktywna, co oznacza, że możesz łatwo znaleźć wsparcie online.

2. Użyte technologie w projekcie po stronie klienta:

FrontEnd



1. Biblioteka projektowa:

Do tworzenia aplikacji webowej użyłem biblioteki ReactJS, ponieważ:

- a. **Virtual DOM:** ReactJS używa wirtualnego DOM (Document Object Model), co pozwala na efektywne aktualizacje interfejsu użytkownika bez konieczności manipulowania rzeczywistym DOM. To przyspiesza rendering i poprawia wydajność aplikacji.
- b. **Obsługa Stanu Aplikacji:** ReactJS posiada mechanizm do zarządzania stanem komponentów, co ułatwia śledzenie i aktualizację danych w odpowiedzi na interakcje użytkownika.
- c. **Duża Społeczność i Ekosystem Narzędzi:** ReactJS cieszy się ogromną społecznością programistyczną, co oznacza dostęp do wielu bibliotek, narzędzi i rozszerzeń. To ułatwia rozwój aplikacji przy wykorzystaniu istniejących rozwiązań.

2. Stylowanie:

Do stylowania użyłem procesowego języka skryptowego Sass, ponieważ:

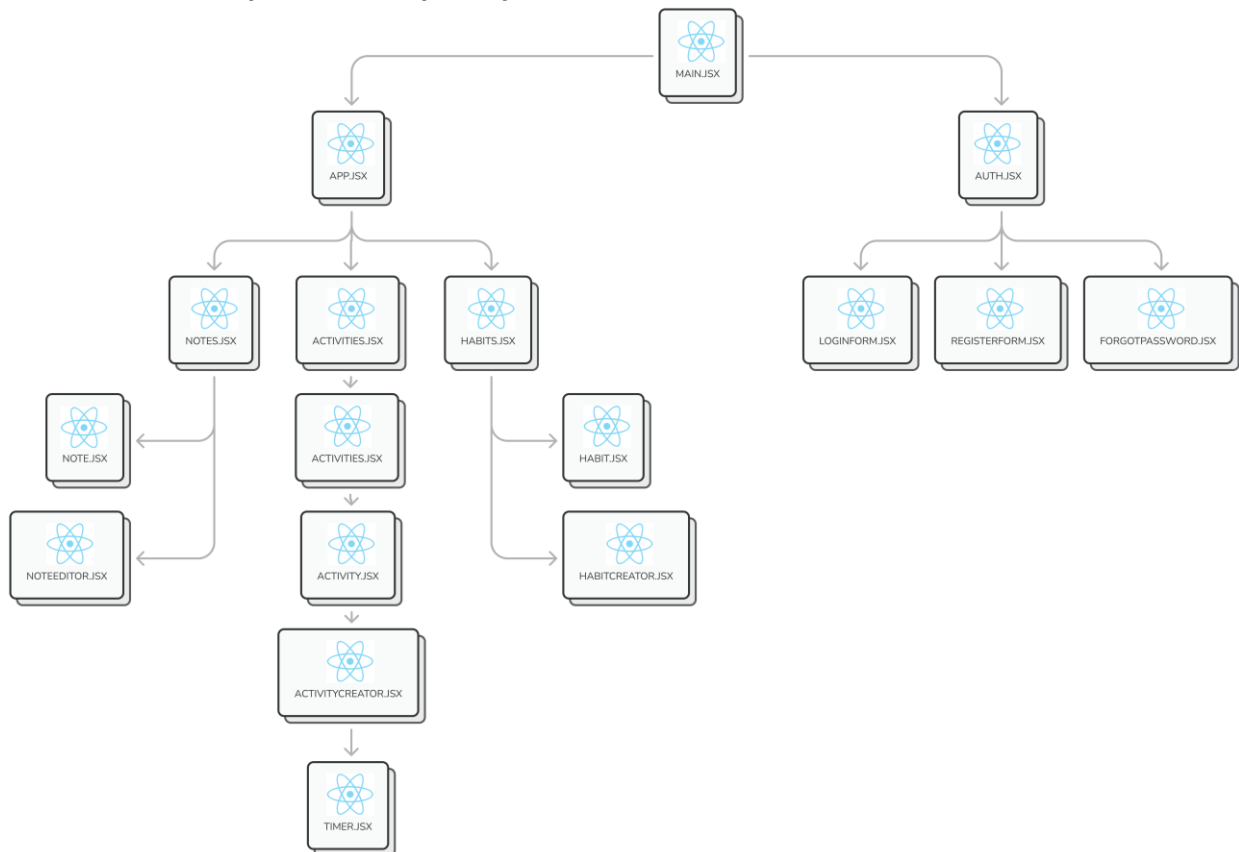
- a. **Rozszerzenia CSS:** Sass dostarcza dodatkowe funkcje, takie jak zmienne, zagnieżdżanie, mixins, funkcje matematyczne, co ułatwia i usprawnia pisanie arkuszy stylów CSS.
- b. **Zmienne:** Sass umożliwia definiowanie zmiennych, co pozwala na przechowywanie wartości i ponowne ich użycie w różnych miejscach w kodzie.
- c. **Zagnieżdżanie:** Sass pozwala na zagnieżdżanie reguł CSS, co sprawia, że kod staje się bardziej hierarchiczny i czytelny.

3. Komunikacja:

Do komunikacji z serwerem użyłem biblioteki Axios, ponieważ:

- a. **Prosta Składnia i Użycie:** Axios oferuje prostą i intuicyjną składnię, co sprawia, że jest łatwy do nauki i używania. Żądania HTTP można definiować w zwarty sposób, co przyspiesza proces implementacji komunikacji między frontendem a backendem.
- b. **Wsparcie dla Promises:** Axios opiera się na Promises, co ułatwia zarządzanie asynchronicznym kodem. To pozwala na bardziej czytelne i zorganizowane obsługiwanie żądań HTTP, zwłaszcza w kontekście asynchronicznych operacji.
- c. **Obsługa Błędów:** Axios oferuje prostą obsługę błędów, co ułatwia identyfikację i zarządzanie sytuacjami, w których żądanie HTTP nie powiedzie się. Możesz łatwo dostosować obsługę błędów na poziomie globalnym lub dla konkretnego żądania.

4. Struktura komponentów aplikacji ReactJS



Rysunek 4 - Rozkład komponentów

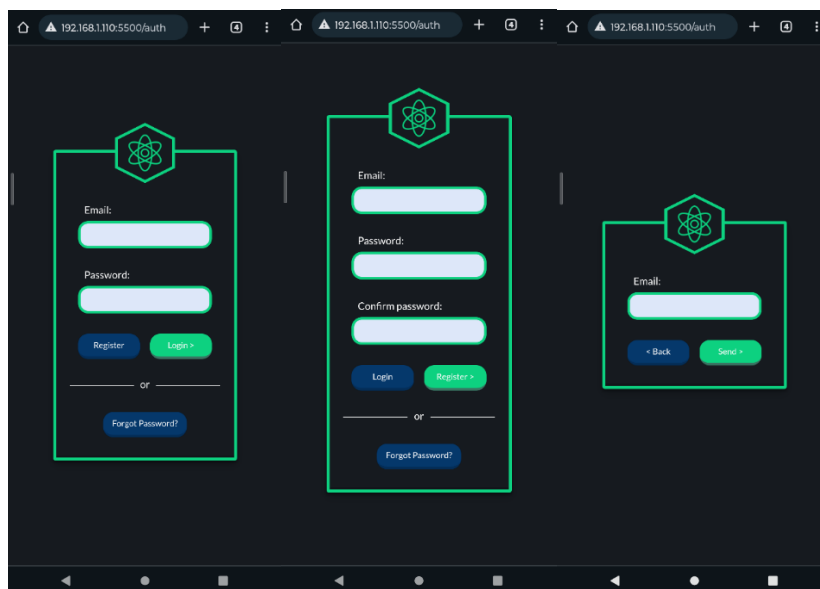
2. Widoki w aplikacji:

Główny komponent aplikacji składa się z czterech widoków przełączanych za pomocą przycisków nawigacyjnych. Zmiana widoków odbywa się w sposób dynamiczny i nie powoduje przeładowywania strony. Oto widoki aplikacji:

- Nawyki – widok w którym użytkownik dodaje swoje nawyki które chce wykonywać codziennie. Aplikacja zapisuje postępy w formie combo które się resetuje gdy użytkownik zapomni o wykonaniu nawyku.
- Notatki – widok notatek pozwala użytkownikowi na dodawanie notatek, edycję oraz usuwanie ich.
- Aktywności – widok ten pozwala użytkownikowi na dodawanie aktywności które chce wykonać w podany dzień i o podanej porze. Jeżeli użytkownik nie wykona przydzielonego zadania serwer zapisuje je jako expired i zadanie te znika z listy użytkownika.
- Ustawienia – widok ten pozwala użytkownikowi na usuwanie konta, zmianę hasła, wylogowanie oraz na zmianę wersji językowej aplikacji.

1. Uwierzytelnianie:

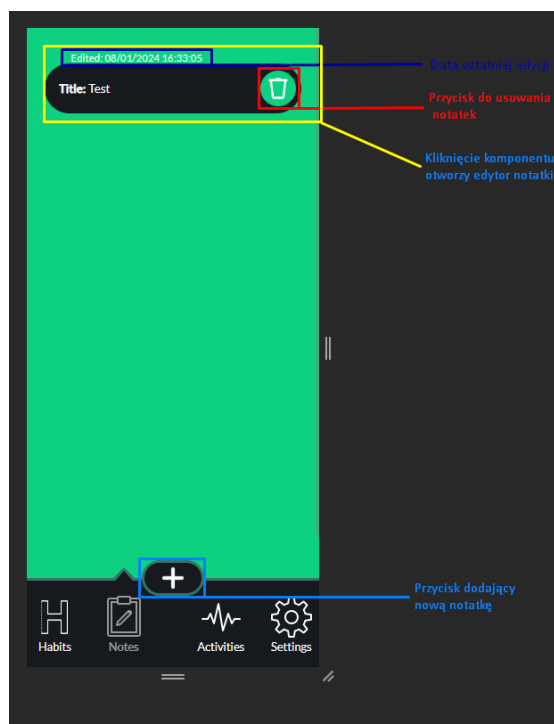
Uwierzytelnianie jest przeprowadzane w komponencie Auth.jsx składającego się z zagnieżdżonych komponentów LoginForm, RegisterForm, ForgotPassword. W tym komponencie użytkownik jest w stanie założyć konto, zalogować się lub w łatwy sposób wysłać email-a z kodem resetującym hasło do istniejącego konta.



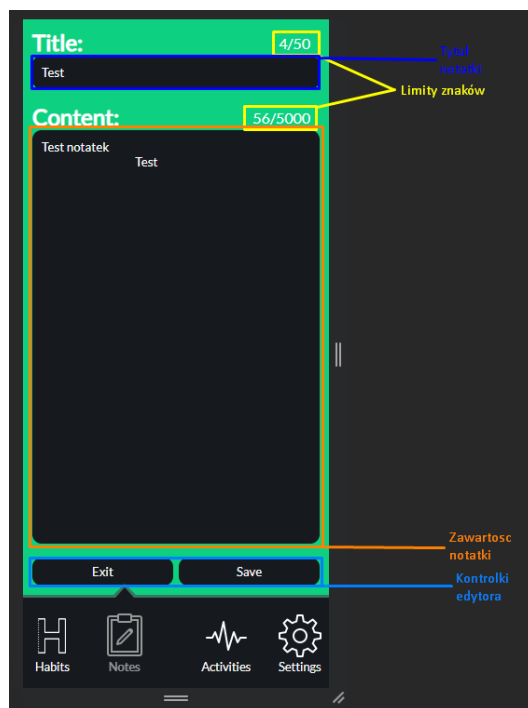
Rysunek 5 - Przedstawia formularz logowania, rejestracji, przypomnienia hasła.

2. Notatki:

Funkcjonalność notatek zbudowany jest w komponencie Notes.jsx który wyświetla listę komponentów Note.jsx. W tym widoku użytkownik ma możliwość dodawania nowych notatek, edytowania starych notatek oraz usuwania notatek z konta.



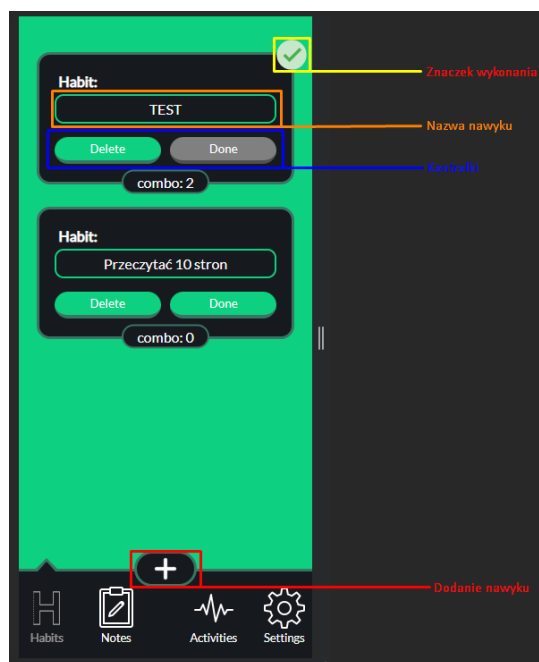
Rysunek 6 – Widok listy notatek.



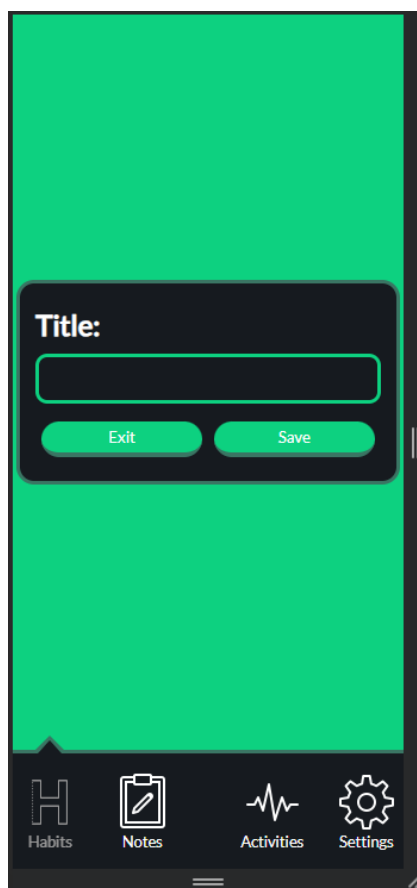
Rysunek 7 - Widok notatki (edytor)

3. Nawyki:

Widok ten umożliwia użytkownikowi tworzenie codziennych aktywności które musi wykonywać. Jeżeli wykona daną czynność zaznacza Done dzięki czemu aplikacja dodaje do combo +1 dzień. Jeżeli użytkownik nie wykona jakiejś czynności licznik combo się zresetuje



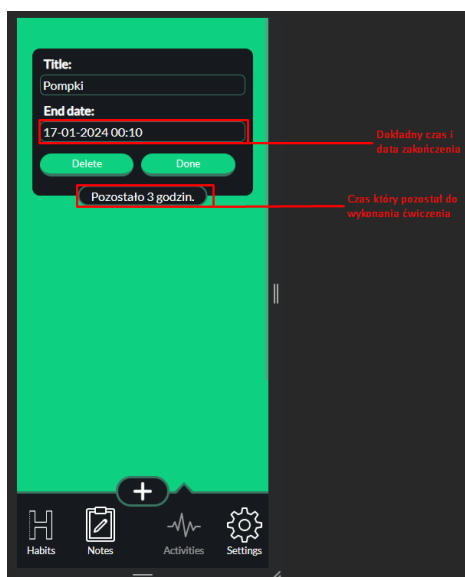
Rysunek 8 - Widok Nawyków.



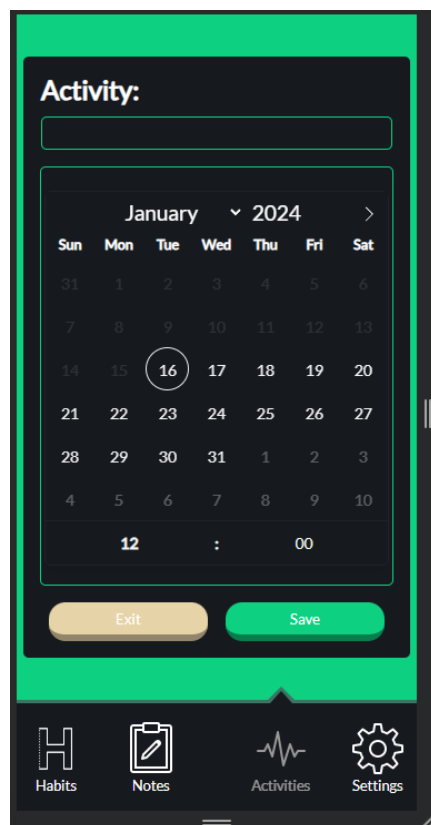
Rysunek 9 - Kreator nawyków.

3. Aktywności:

Funkcjonalność ta pozwala użytkownikowi dodawać aktywności do wykonania w podanym dniu i godzinie. Jeżeli aktywność nie zostanie wykonana zmieni swój status na „expired” i przestanie się wyświetlać.



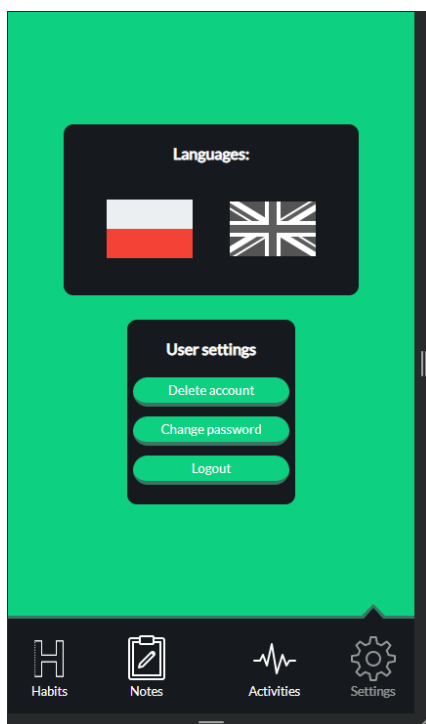
Rysunek 10 - Widok aktywności.



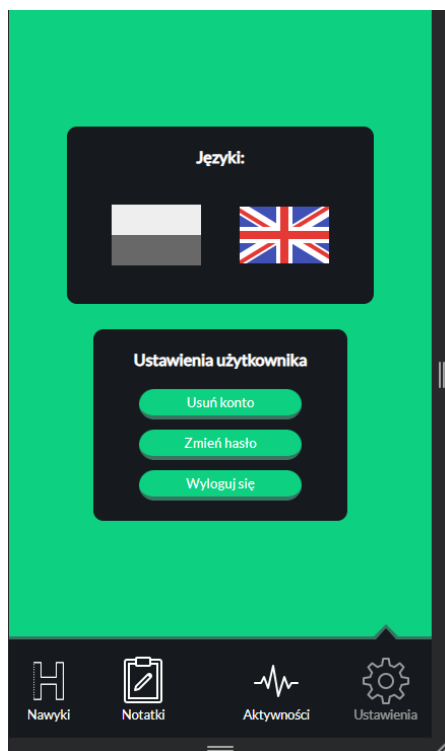
Rysunek 11 - Kreator aktywności.

4. Ustawienia:

Widok ustawień pozwala użytkownikowi na usunięcie swojego konta, zmienienie hasła do konta, wylogowanie się z konta oraz zmiana języku między Polskim a Angielskim. Zmiana języka wykonuje się w sposób dynamiczny i nie wymaga przeładowywania strony.



Rysunek 12 - Angielska wersja językowa.



Rysunek 13 - Polska wersja językowa.

3. Aplikacja serwerowa:

Aplikacja serwerowa została napisana w języku Python z wykorzystaniem bibliotek: Flask, Firebase, Pyrebase. Aplikacja serwerowa składa się:

- Skrypt główny: App.py
- Moduły/Klasy:
 - Activities.py – wykonuje operacje na aktywnościach.
 - Habits.py – wykonuje operacje na nawykach.
 - Notes.py – wykonuje operacje na notatkach.
 - User.py – wykonuje operacje na kontach użytkowników.
 - Helpers.py – moduł z funkcjami pomagającymi.

1. Konfiguracja aplikacji Flask oraz konfiguracja firebase:

Wszystkie klucze oraz inne dane prywatne przechowywane są w pliku .env w celach bezpieczeństwa.

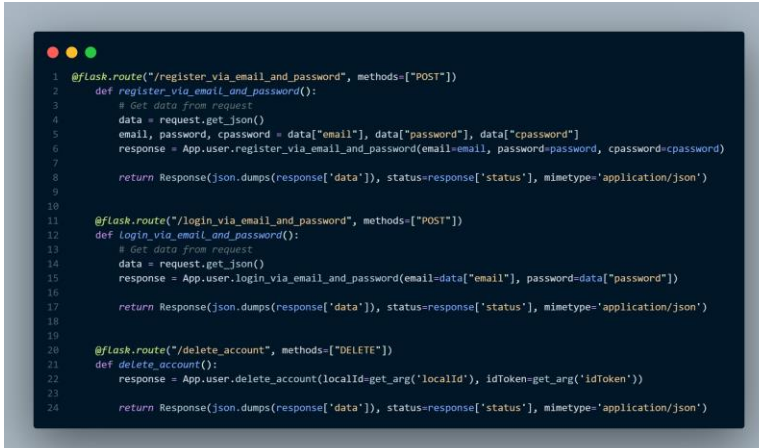


```
1 class App:
2
3     # Flask config
4     flask = Flask(__name__)
5     CORS(flask)
6
7     # Firebase config
8     firebase = pyrebase.initialize_app({
9         "apiKey": env["apiKey"],
10        "authDomain": env["authDomain"],
11        "databaseURL": env["databaseURL"],
12        "projectId": env["projectId"],
13        "storageBucket": env["storageBucket"],
14        "messagingSenderId": env["messagingSenderId"],
15        "appId": env["appId"],
16        "measurementId": env["measurementId"]
17    })
18    firebase_auth = firebase.auth()
19    firebase_db = firebase.database()
20
21    # Classes
22    user = User(firebase_auth=firebase_auth, firebase_db=firebase_db)
23    notes = Notes(firebase_db=firebase_db)
24    habits = Habits(firebase_db=firebase_db)
25    activities = Activities(firebase_db=firebase_db)
```

Rysunek 14 - Konfiguracja aplikacji Flask oraz firebase.

2. Przykładowe „route’y”:

Aplikacja flask obsługuje żądania http poprzez flask.route. Pobierane są tutaj dane z zapytania oraz przeprowadzane są operacje poprzez wywoływanie odpowiednich metod z innych klas.



```
1 @flask.route("/register_via_email_and_password", methods=["POST"])
2 def register_via_email_and_password():
3     # Get data from request
4     data = request.get_json()
5     email, password, cpassword = data["email"], data["password"], data["cpassword"]
6     response = App.user.register_via_email_and_password(email=email, password=password, cpassword=cpassword)
7
8     return Response(json.dumps(response["data"]), status=response["status"], mimetype='application/json')
9
10
11 @flask.route("/login_via_email_and_password", methods=["POST"])
12 def login_via_email_and_password():
13     # Get data from request
14     data = request.get_json()
15     response = App.user.login_via_email_and_password(email=data["email"], password=data["password"])
16
17     return Response(json.dumps(response["data"]), status=response["status"], mimetype='application/json')
18
19
20 @flask.route("/delete_account", methods=["DELETE"])
21 def delete_account():
22     response = App.user.delete_account(localId=get_arg('localId'), idToken=get_arg('idToken'))
23
24     return Response(json.dumps(response["data"]), status=response["status"], mimetype='application/json')
```

Rysunek 15 - Przykładowe route'y apk Flask.

3. Klasy:

a. Klasa Activities:

```
1 class Activities:
2     def __init__(self, firebase_db):
3         self.db = firebase_db
4
5
6     # ++++++ HELPERS ++++++
7     def user_path(self, localId): return self.db.child("users").child(localId)
8     def get_activity_by_id(self, localId, id): return self.user_path(localId).child("activities").child(id)
9     # ++++++ HELPERS ++++++
10
11     def get_activities(self, localId):
12         if not self.user_path(localId).child("activities").get().val():
13             return {'data': f'You have no notes yet.', 'status': 204}
14
15         try:
16             response = self.user_path(localId).child("activities").get()
17             if response:
18                 return {'data': move_keys_inside_values(response.val()), 'status': 200}
19             else:
20                 return {'data': 'No activities found.', 'status': 404}
21         except Exception as e:
22             print(e)
23             return {'data': f'Getting activities failed. {e}', 'status': 520}
24
25     def add_activity(self, localId, title, date, time):
26         try:
27             response = self.user_path(localId).child("activities").push({
28                 "title": title,
29                 "date": date,
30                 "time": time,
31                 "state": "pending",
32                 "added": dt.now().strftime("%d/%m/%Y %H:%M:%S")
33             })
34
35             if response:
36                 return {'data': f'Activity added successfully.', 'status': 201}
37
38         except Exception as e:
39             print(e)
40             return {'data': f'Adding activity failed. {e}', 'status': 520}
41
42
43     def change_activity_state(self, localId, id, state):
44         try:
45             response = self.user_path(localId).child("activities").child(id).update({
46                 "state": state
47             })
48
49             if response:
50                 return {'data': f'Activity state changed successfully.', 'status': 200}
51
52         except Exception as e:
53             print(e)
54             return {'data': f'Changing activity state failed. {e}', 'status': 520}
55
56
57     def delete_activity(self, localId, id):
58         try:
59             response = self.get_activity_by_id(localId, id).remove()
60             return {'data': f'Activity deleted successfully.', 'status': 200}
61
62         except Exception as e:
63             print(e)
64             return {'data': f'Deleting activity failed. {e}', 'status': 520}
```

Rysunek 16 - kod klasy activities.

b. Klasa Habits:

```
1 class Habits:
2     def __init__(self, firebase_db):
3         self.db = firebase_db
4
5     # ++++++ HELPERS ++++++
6     def user_path(self, localId): return self.db.child("users").child(localId)
7     def __get_habit_by_id(self, localId, noteId): return self.user_path(localId).child("habits").child(noteId)
8     # ----- HELPERS -----
9
10    def get_habits(self, localId):
11        if not self.user_path(localId).child("habits").get().val():
12            return {'data': f'You have no habits yet.', 'status': 204}
13
14        try:
15            data = self.user_path(localId).child('habits').get()
16            if data.val():
17
18                updated_data = {}
19                for habit_id, habit in data.val().items():
20                    if habit['date'] != dt.now().strftime("%d/%m/%Y"):
21                        updated_data[habit_id] = {
22                            "title": habit['title'],
23                            "date": dt.now().strftime("%d/%m/%Y"),
24                            "combo": 0 if not habit['done'] else habit['combo'],
25                            "done": False,
26                        }
27
28                if updated_data:
29                    self.user_path(localId).child('habits').update(updated_data)
30                    data = self.user_path(localId).child('habits').get()
31
32                data = move_keys_inside_values(data.val())
33                return {'data': data, 'status': 200}
34            else:
35                return {'data': f'Fetching habits failed.', 'status': 520}
36
37        except Exception as e:
38            return {'data': f'Fetching habits failed. {e}', 'status': 520}
39
40
41    def add_habit(self, localId, title):
42        try:
43            response = self.user_path(localId).child("habits").push({
44                "title": title,
45                "date": dt.now().strftime("%d/%m/%Y"),
46                "done": False,
47                "combo": 0,
48            })
49
50            if response:
51                return {'data': f'Habit added successfully.', 'status': 201}
52
53        except Exception as e:
54            return {'data': f'Adding habit failed. {e}', 'status': 520}
55
56
57    def done_habit(self, localId, habitId):
58        habit_path = self.__get_habit_by_id(localId, habitId).get().val()
59        try:
60            self.user_path(localId).child(f'habits/{habitId}').update({
61                "done": True,
62                "combo": habit_path['combo'] + 1,
63            })
64            return {'data': f'Habit done successfully.', 'status': 200}
65
66        except Exception as e:
67            print(e)
68            return {'data': f'Doing habit failed. {e}', 'status': 520}
69
70
71    def delete_habit(self, localId, habitId):
72        try:
73            self.user_path(localId).child(f'habits/{habitId}').remove()
74            return {'data': f'Habit deleted successfully.', 'status': 200}
75
76        except Exception as e:
77            return {'data': f'Deleting habit failed. {e}', 'status': 520}
```

c. Klasa Notes:

```
1 class Notes:
2     def __init__(self, firebase_db):
3         self.db = firebase_db
4
5
6     # ++++++ HELPERS ++++++
7     def user_path(self, localId): return self.db.child("users").child(localId)
8     def __get_note_by_id(self, localId, noteId): return self.user_path(localId).child("notes").child(noteId)
9     # ----- HELPERS -----
10
11
12     def get_notes(self, localId):
13         if not self.user_path(localId).child("notes").get().val():
14             return {'data': f'You have no notes yet.', 'status': 204}
15
16         try:
17             data = self.user_path(localId).child('notes').get()
18             if data.val():
19                 data = move_keys_inside_values(data.val())
20                 return {'data': data, 'status': 200}
21             else:
22                 return {'data': f'Fetching notes failed.', 'status': 520}
23
24         except Exception as e:
25             return {'data': f'Fetching notes failed. {e}', 'status': 520}
26
27
28     def add_note(self, localId, title, content):
29         try:
30             response = self.user_path(localId).child("notes").push({
31                 "title": title,
32                 "content": content,
33                 "date": dt.now().strftime("%d/%m/%Y %H:%M:%S")
34             })
35             if response:
36                 return {'data': f'Note added successfully.', 'status': 201}
37
38         except Exception as e:
39             return {'data': f'Adding note failed. {e}', 'status': 520}
40
41
42     def update_note(self, localId, noteId, title, content):
43         try:
44             self.user_path(localId).child(f'notes/{noteId}').update({
45                 "title": title,
46                 "content": content,
47                 "date": dt.now().strftime("%d/%m/%Y %H:%M:%S"),
48             })
49             return {'data': f'Note updated successfully.', 'status': 200}
50
51         except Exception as e:
52             return {'data': f'Updating note failed. {e}', 'status': 520}
53
54     def delete_note(self, localId, noteId):
55         try:
56             self.__get_note_by_id(localId, noteId).remove()
57             return {'data': f'Note deleted successfully.', 'status': 200}
58         except Exception as e:
59             return {'data': f'Deleting note failed. {e}', 'status': 520}
60
```

Rysunek 18 - Klasa Notes.

```

1 class User:
2     def __init__(self, firebase_auth, firebase_db):
3         self.auth = firebase_auth
4         self.db = firebase_db
5
6     # ===== REGISTRATION =====
7     def __get_user_path(self, localid): return self.db.child("user").child(localid)
8     # =====
9
10    # ===== REGISTER =====
11    @staticmethod
12    def __register_create_user_template(email):
13        return {
14            "user_info": {
15                "email": email,
16                "password": email.split("@")[0],
17                "created_at": datetime.now().strftime("%d/%m/%Y %H:%M:%S"),
18                "disabled_at": "",
19                "deleted_at": "",
20                "role": "user",
21                "account_status": "non-verified",
22                "lang": {
23                    "login": {
24                        "password_changes": [],
25                        "email_changes": [],
26                    },
27                },
28                # additional fields (to be added later)
29                "profile_picture": "",
30                "gender": "",
31            },
32            "user_settings": {
33                "theme": "default",
34                "language": "en",
35            },
36        }
37
38    @staticmethod
39    def __register_validate_method("args, **kwargs"):
40        # returns status (if validation passes, else returns error message)
41
42        _EMAIL_REGEX = re.compile(r'^[a-z0-9]+(\.[a-z0-9]+)*@[a-z0-9]+(\.[a-z0-9]+)*$', re.I)
43        _PASSWORD_REGEX = re.compile(r'^[a-zA-Z0-9]{8,}$')
44
45        # method to "email/password":
46        if kwargs['password'] != kwargs['password']:
47            return {"data": "Passwords do not match", "status": 422}
48
49        if not _EMAIL_REGEX.match(kwargs['email']):
50            return {"data": "Invalid email", "status": 422}
51
52        if not _PASSWORD_REGEX.match(kwargs['password']):
53            print(kwargs['password'])
54            return {"data": "Minimum eight characters, at least one letter, one number and one special character.", "status": 422}
55
56        return False
57
58    def register_via_email_and_password(self, email, password, password2):
59        if (validation_res := self.__register_validate("email/password", email=email, password=password, password=password2)): return validation_res
60
61        user = None
62        try:
63            # create user
64            user = self.auth.create_user_with_email_and_password(email, password)
65            if user:
66                # add user to database with default values
67                self.__get_user_path(user['localid']).set(self.__register_create_user_template(email))
68                return {"data": "user created successfully", "status": 201}
69            return {"data": "user creation failed", "status": 400}
70        except Exception as e:
71            if user:
72                # delete user (if user database creation fails)
73                self.auth.delete_user_account(user['email'])
74            if 'EMAIL_EXISTS' in str(e):
75                return {"data": "Email already exists", "status": 400}
76            return {"data": "Registration failed: {e}", "status": 500}
77
78    # ===== LOGIN =====
79    # This method is implemented to reduce the number of requests sent to Firebase.
80    @staticmethod
81    def __login_validate_method("args, **kwargs"):
82        # returns status (if validation passes, else returns error message)
83
84        _EMAIL_REGEX = re.compile(r'^[a-z0-9]+(\.[a-z0-9]+)*@[a-z0-9]+(\.[a-z0-9]+)*$', re.I)
85        _PASSWORD_REGEX = re.compile(r'^[a-zA-Z0-9]{8,}$')
86
87        # method to "email/password":
88        if not _EMAIL_REGEX.match(kwargs['email']):
89            return {"data": "Invalid email", "status": 422}
90
91        if not _PASSWORD_REGEX.match(kwargs['password']):
92            return {"data": "Minimum eight characters, at least one letter, one number and one special character.", "status": 422}
93
94        return False
95
96    def login_via_email_and_password(self, email, password):
97        # validate email and password
98        if (validation_res := self.__login_validate("email/password", email=email, password=password)): return validation_res
99
100        user = self.auth.sign_in_with_email_and_password(email, password)
101        if user:
102            # check user status
103            account_status = self.__get_user_path(user['localid']).child("user_info/account_status").get().val()
104            if account_status == "suspended":
105                return {"data": "Account suspended", "status": 403}
106            elif account_status == "disabled":
107                # TODO: To not allow FCM push to the disabled account, we need to *** REMOVE ***
108                disabled_at = self.__get_user_path(user['localid']).child("user_info/disabled_at").get().val()
109                delete_at = datetime.strptime(disabled_at, "%d/%m/%Y") + timedelta(days=30)
110                return {"data": "Account disabled at {disabled_at} and will be deleted {delete_at}", "status": 403}
111            elif account_status == "deleted":
112                return {"data": "Account deleted", "status": 403}
113
114            # add language to response
115            user['language'] = self.__get_user_path(user['localid']).child("user_settings/language").get().val()
116            print(user['language'])
117            return {"data": "user", "status": 200}
118        else:
119            return {"data": "Unexpected error", "status": 500}
120
121    except Exception as e:
122        if 'INVALID_LOGIN_CREDENTIALS' in str(e):
123            return {"data": "Wrong email or password", "status": 403}
124        return {"data": "Login failed: {e}", "status": 500}
125
126    # ===== OTHER =====
127    def change_password(self, email):
128        # =====
129        try:
130            self.auth.send_password_reset_email(email)
131            return {"data": "Email sent successfully", "status": 200}
132        except Exception as e:
133            return {"data": "Password change failed: {e}", "status": 500}
134
135    def delete_account(self, localid, ifname):
136        try:
137            # delete user from database
138            user_info = self.__get_user_path(localid).child("user_info")
139            user_info.update({"account_status": "disabled", "disabled_at": datetime.now().strftime("%d/%m/%Y")})
140            return {"data": "Account deleted successfully", "status": 200}
141        except Exception as e:
142            return {"data": "Account deletion failed: {e}", "status": 500}
143
144    def change_user_lang(self, localid, lang):
145        try:
146            self.__get_user_path(localid).child("user_settings").update({"language": lang})
147            return {"data": "Language change successfully", "status": 200}
148        except Exception as e:
149            return {"data": "Language change failed: {e}", "status": 500}
150
151    def get_user(self, localid):
152        # =====
153        try:
154            # get user settings and info
155            user_settings = self.__get_user_path(localid).child("user_settings").get().val()
156            user_info = self.__get_user_path(localid).child("user_info").get().val()
157            user = {"user_settings": user_settings, "user_info": user_info}
158            return {"data": "user", "status": 200}
159        except Exception as e:
160            return {"data": "User get failed: {e}", "status": 500}
161
162    # ===== OTHER =====

```

Rysunek 19 - Klasa User.