Dokumentacja do aplikacji CRM

Spis treści:

Wymagania funkcjonalne

1. **Rejestracja użytkowników**
   - Formularz rejestracji z polami: nazwa użytkownika, imię, nazwisko, email, hasło.
   - Walidacja hasła zgodnie z wymaganiami bezpieczeństwa.
   - Automatyczne logowanie po pomyślnej rejestracji.

2. **Logowanie użytkowników**
   - Formularz logowania z polami: nazwa użytkownika, hasło.
   - Walidacja danych logowania i autoryzacja użytkownika.

3. **Wylogowanie użytkowników**
   - Możliwość wylogowania użytkownika.

4. **Zarządzanie rekordami klientów**
   - Dodawanie nowych rekordów klientów z polami: imię, nazwisko, email, numer telefonu, adres, miasto, województwo, kod pocztowy.
   - Wyświetlanie listy wszystkich rekordów klientów.
   - Wyświetlanie szczegółów konkretnego rekordu klienta.
   - Aktualizacja istniejących rekordów klientów.
   - Usuwanie istniejących rekordów klientów.

5. **Wyświetlanie pogody**
   - Pobieranie i wyświetlanie aktualnej pogody dla miasta Warszawa.
   - Informacje o pogodzie: temperatura, opis, nazwa miasta.

6. **Wyświetlanie świąt**
   - Pobieranie i wyświetlanie aktualnych świąt dla Polski.
   - Informacje o świętach: nazwa.

7. **Wyświetlanie nowości**
   - Pobieranie i wyświetlanie aktualnych nowości świata.

Wymagania niefunkcjonalne

1. **Bezpieczeństwo**
   - Stosowanie walidacji haseł przy rejestracji użytkowników.

2. **Wydajność**
   - Szybki czas odpowiedzi dla operacji CRUD (tworzenie, odczyt, aktualizacja, usuwanie) na rekordach klientów.
   - Optymalizacja zapytań do API w celu minimalizacji czasu odpowiedzi.

3. **Skalowalność**
   - Możliwość łatwego dodawania nowych funkcjonalności, takich jak obsługa dodatkowych miast w module pogodowym czy obsługa innych krajów w

module świąt.

4. **Dostępność**
   - Aplikacja powinna być dostępna dla użytkowników 24/7 z minimalnymi przerwami serwisowymi.

5. **Interfejs użytkownika**
   - Intuicyjny i łatwy w nawigacji interfejs użytkownika.
   - Minimalistyczny design.

6. **Zarządzanie błędami**
   - Logowanie błędów i wyjątków w celu łatwiejszego diagnozowania problemów.
   - Przyjazne komunikaty o błędach dla użytkowników.

Przykładowe scenariusze użytkowania

1. **Rejestracja nowego użytkownika**
   - Użytkownik wchodzi na stronę rejestracji, wypełnia formularz i przesyła go.
   - System waliduje dane, zapisuje nowego użytkownika i loguje go automatycznie.

2. **Logowanie użytkownika**
   - Użytkownik wchodzi na stronę logowania, wprowadza swoją nazwę użytkownika i hasło, a następnie przesyła formularz.
   - System weryfikuje dane i, jeśli są poprawne, loguje użytkownika.

3. **Dodawanie nowego rekordu klienta**
   - Zalogowany użytkownik wchodzi na stronę dodawania rekordu, wypełnia formularz i przesyła go.
   - System zapisuje nowy rekord klienta i wyświetla go na liście rekordów.

4. **Wyświetlanie listy rekordów klientów**
   - Zalogowany użytkownik wchodzi na stronę główną, gdzie widzi listę wszystkich zapisanych rekordów klientów.

Wymagania techniczne

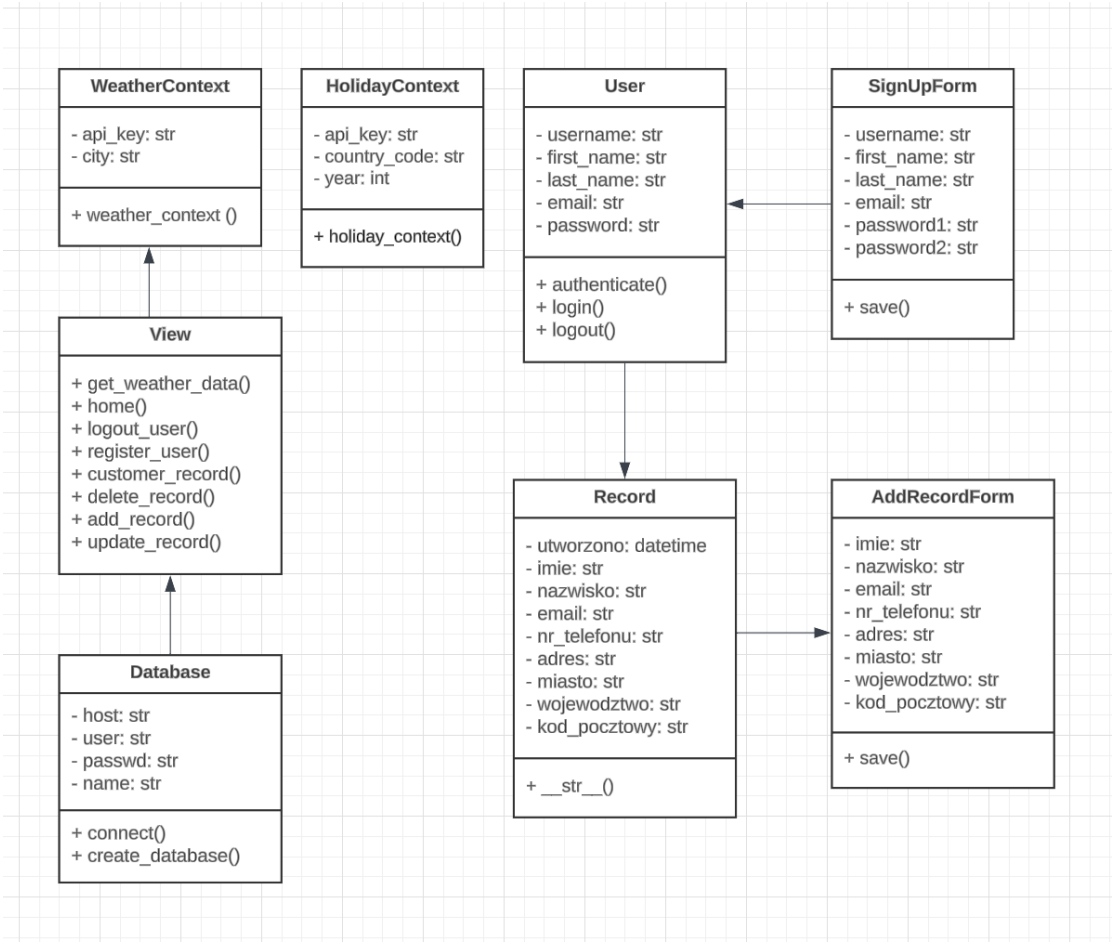
1. **Środowisko serwera**
   - Serwer lokalny z obsługą Python i Django w wirtualnym środowisku.
   - Baza danych oparta o chmurowe rozwiązanie Aiven Platform.

2. **Technologie**
   - Python 3.x
   - Django 3.x

3. **API zewnętrzne**
   - OpenWeatherMap API do pobierania danych pogodowych.
   - Holiday API do pobierania danych o świętach.
   - News Api do pobierania nowości.

4. **Testowanie**
   - Jednostkowe testy dla kluczowych funkcji i modele aplikacji.

Diagramy klas UML dla CRM



Relacje:

User "1" ----------> "0..*" Record
User "1" ----------> "1" SignUpForm
Record "1" ----------> "1" AddRecordForm
View "1" ----------> "1" WeatherContext
View "1" ----------> "1" HolidayContext
Database "1" ----------> "0..*" View

Opis diagramu Klas

1. **User:**
   - Reprezentuje użytkownika systemu.
   - Atrybuty: `username`, `first_name`, `last_name`, `email`, `password`.
   - Metody: `authenticate()`, `login()`, `logout()`.

2. **SignUpForm:**
   - Formularz rejestracyjny dla nowych użytkowników.
   - Atrybuty: `username`, `first_name`, `last_name`, `email`, `password1`, `password2`.
   - Metody: `save()`.

3. **Record:**
   - Reprezentuje rekord klienta.
   - Atrybuty: `utworzono`, `imie`, `nazwisko`, `email`, `nr_telefonu`, `adres`, `miasto`, `wojewodztwo`, `kod_pocztowy`.
   - Metody: `__str__()`.

4. **AddRecordForm:**
   - Formularz dodawania nowego rekordu klienta.
   - Atrybuty: `imie`, `nazwisko`, `email`, `nr_telefonu`, `adres`, `miasto`, `wojewodztwo`, `kod_pocztowy`.
   - Metody: `save()`.

5. **WeatherContext:**
   - Reprezentuje kontekst pogodowy.
   - Atrybuty: `api_key`, `city`.
   - Metody: `get_weather_data()`.

6. **HolidayContext:**
   - Reprezentuje kontekst świąt.
   - Atrybuty: `api_key`, `country_code`, `year`.
   - Metody: `get_holiday_data()`.

7. **View:**
   - Reprezentuje widoki aplikacji.
   - Metody: `home()`, `logout_user()`, `register_user()`, `customer_record()`, `delete_record()`, `add_record()`, `update_record()`.

8. **Database:**
   - Reprezentuje połączenie z bazą danych.
   - Atrybuty: `host`, `user`, `passwd`, `name`.
   - Metody: `connect()`, `create_database()`.

Relacje

- User może mieć wiele Record.
- SignUpForm jest powiązany z jednym User.
- AddRecordForm jest powiązany z jednym Record.
- View jest powiązany z jednym WeatherContext i jednym HolidayContext.
- Database jest powiązany z wieloma View

Opis sposobów i metod testowania

Testy jednostkowe

W ramach testów jednostkowych dla projektu CRM przeprowadzono testy dla formularzy, modeli oraz widoków.

1. Testy Formularzy (website/tests/test_forms.py)

Test SignUpFormTest.test_signup_form_valid_data:

- **Cel**: Sprawdzenie, czy formularz rejestracji (SignUpForm) jest prawidłowo wypełniony danymi.
- **Opis**: Tworzymy instancję formularza SignUpForm z prawidłowymi danymi, takimi jak nazwa użytkownika, imię, nazwisko, e-mail oraz hasła.
- **Sprawdzenie**: Upewniamy się, że formularz jest ważny (is_valid()), co oznacza, że wszystkie wymagane pola zostały poprawnie wypełnione.
- **Oczekiwany wynik**: Formularz powinien być prawidłowy.

Test SignUpFormTest.test_signup_form_invalid_data:

- **Cel**: Sprawdzenie, czy formularz rejestracji (SignUpForm) wykrywa brak danych.
- **Opis**: Tworzymy instancję formularza SignUpForm z pustymi danymi.
- **Sprawdzenie**: Upewniamy się, że formularz jest nieważny (is_valid()) oraz że liczba błędów w formularzu wynosi 6
  (pola username, first_name, last_name, email, password1, password2)
  .
- **Oczekiwany wynik**: Formularz powinien być nieważny z powodu brakujących danych.

```python
from django.test import TestCase
from website.forms import SignUpForm

class SignUpFormTest(TestCase):

    def test_signup_form_valid_data(self):
        form = SignUpForm(data={
            'username': 'jan_kowalski',
            'first_name': 'Jan',
            'last_name': 'Kowalski',
            'email': 'jan.kowalski@example.com',
            'password1': 'SuperSecret123',
            'password2': 'SuperSecret123'
        })
        print(form.errors)
        self.assertTrue(form.is_valid())

    def test_signup_form_invalid_data(self):
        form = SignUpForm(data={})
        self.assertFalse(form.is_valid())
        self.assertEqual(len(form.errors), 6)

```

2. Testy Modeli (website/tests/test_models.py)

Test RecordModelTest.test_record_creation:

- **Cel**: Sprawdzenie poprawności tworzenia rekordu w modelu Record.
- **Opis**: Tworzymy instancję modelu Record z przykładowymi danymi (imię, nazwisko, e-mail, nr telefonu, adres, miasto, województwo, kod pocztowy).
- **Sprawdzenie**: Upewniamy się, że utworzony rekord jest poprawny oraz że wszystkie dane zostały poprawnie zapisane.
- **Oczekiwany wynik**: Rekord powinien zostać poprawnie utworzony.

```python
from django.test import TestCase
from website.models import Record

class RecordModelTest(TestCase):

    def setUp(self):
        self.record = Record.objects.create(
            imie="Jan",
            nazwisko="Kowalski",
            email="jan.kowalski@example.com",
            nr_telefonu="123456789",
            adres="Ul. Przykładowa 1",
            miasto="Warszawa",
            wojewodztwo="Mazowieckie",
            kod_pocztowy="00-001"
        )

    def test_record_creation(self):
        self.assertEqual(self.record.imie, "Jan")
        self.assertEqual(self.record.nazwisko, "Kowalski")
        self.assertEqual(self.record.email, "jan.kowalski@example.com")
        self.assertEqual(self.record.nr_telefonu, "123456789")
        self.assertEqual(self.record.adres, "Ul. Przykładowa 1")
        self.assertEqual(self.record.miasto, "Warszawa")
        self.assertEqual(self.record.wojewodztwo, "Mazowieckie")
        self.assertEqual(self.record.kod_pocztowy, "00-001")

```

3. Testy Widoków (website/tests/test_views.py)

Test HomeViewTest.test_home_view_with_logged_in_user:

- **Cel**: Sprawdzenie widoku strony głównej dla zalogowanego użytkownika.
- **Opis**: Logujemy użytkownika i wykonujemy zapytanie do widoku strony głównej.
- **Sprawdzenie**: Upewniamy się, że odpowiedź zawiera kod statusu 200 oraz że w odpowiedzi są obecne dane rekordów.
- **Oczekiwany wynik**: Widok powinien poprawnie ładować się z danymi dla zalogowanego użytkownika.

Test HomeViewTest.test_home_view_with_logged_out_user:

- **Cel**: Sprawdzenie widoku strony głównej dla niezalogowanego użytkownika.
- **Opis**: Wykonujemy zapytanie do widoku strony głównej bez logowania użytkownika.

- **Sprawdzenie**: Upewniamy się, że odpowiedź zawiera kod statusu 200 oraz że dane pogodowe są dostępne w kontekście.
- **Oczekiwany wynik**: Widok powinien poprawnie ładować się z danymi pogodowymi dla niezalogowanego użytkownika.

```python
from django.test import TestCase
from django.urls import reverse
from django.contrib.auth.models import User

class HomeViewTest(TestCase):

    def test_home_view_status_code(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_home_view_template_used(self):
        response = self.client.get(reverse('home'))
        self.assertTemplateUsed(response, 'home.html')

```

Podsumowanie

Przeprowadzone testy obejmowały:

1. **Formularze**: Sprawdzano, czy formularze są poprawnie walidowane przy wypełnieniu prawidłowymi i nieprawidłowymi danymi.
2. **Modele**: Testowano, czy rekordy mogą być poprawnie tworzone i zapisywane w bazie danych.
3. **Widoki**: Weryfikowano, czy widoki odpowiednio obsługują zarówno zalogowanych, jak i niezalogowanych użytkowników oraz czy odpowiednie dane są przekazywane do szablonów.

```
/c/test_project/29415_52665_52668_51413_51414 (mai
n)
$ pytest
============================= test session starts =============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
django: version: 5.0.6, settings: project.settings (from ini)
rootdir: C:\test_project\29415_52665_52668_51413_51414
configfile: pytest.ini
plugins: django-4.8.0
collected 5 items

website\tests\test_forms.py ..                                           [ 40%]
website\tests\test_models.py .                                           [ 60%]
website\tests\test_views.py ..                                           [100%]

============================== 5 passed in 10.49s ==============================
(wirtual)
```

Testowanie ręczne

Testowanie ręczne polegało na ręcznym sprawdzaniu funkcji aplikacji przez uczestników grupy:
- Przeglądanie interfejsu użytkownika i sprawdzanie poprawności wyświetlania danych.
- Wykonywanie scenariuszy testowych przez testerów, aby upewnić się, że aplikacja działa zgodnie z oczekiwaniami.