DOKUMENTACJA TECHNICZNA

Aplikacja pogodowa.

Autorzy:

Grzegorz Swajda

Karol Wykrota

Jakub Sadza

Opis zastosowanych technologii

W ramach tego projektu wykorzystano różne technologie, które pomogły w jego realizacji. Poniżej znajduje się lista głównych technologii użytych w projekcie:

Języki programowania:

- Dart
- Python

Frameworki:

- Flutter
- FastAPI

Biblioteki/Pakiety:

Backend	
Nazwa biblioteki	**Wersja**
OSMPythonTools	0.3.5
cryptography	41.0.4
fastapi	0.103.2
uvicorn	0.23.2
pydantic	2.4.2
request	2.31.0

Frontend	
Nazwa pakietu	**Wersja**
Go router	^11.1.3
provider	^6.0.5
location	^5.0.3
http	^1.1.0
Flutter svg	^2.0.7
encrypt	^5.0.3
Shared preferences	^2.2.2

Opis implementacji z fragmentami kodu źródłowego

Backend

Backend projektu został zaimplementowany przy użyciu framework FastAPI oraz pakietów potrzebnych do realizacji funkcjonowania serwera zgodnie z jej przeznaczeniem. Do najważniejszych fragmentów implementacji kodu stanowią:

Szyfrowanie/deszyfrowanie danych:

Szyfrowanie oraz deszyfrowanie danych przez serwer opiera się udzieleniu aplikacji, publicznego klucza serwera, którym wiadomości będą szyfrowane przez aplikację i dekodowane przez serwer swoim kluczem prywatnym. Wszystkie potrzebne funkcje, realizujące szyfrowanie oraz deszyfrowanie danych są zawarte w pliku crypto.py, znajdującym się w głównym katalogu.

Funkcja generowania pary kluczy:

```
def key_generate():
    private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048, backend=default_backend())
    public_key = private_key.public_key()
    private_key_pem =
private_key.private_bytes(encoding=serialization.Encoding.PEM,format=serial
ization.PrivateFormat.PKCS8,encryption_algorithm=serialization.NoEncryption
())
    with open("private_key.pem", "wb") as file:
        file.write(private_key_pem)
    public_key_pem =
public_key.public_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo)
    with open("public_key.pem", "wb") as file:
        file.write(public_key_pem)
```

Funkcja key_generate generuje klucz prywatny i klucz publiczny dla algorytmu RSA o długości klucza 2048 bitów i współczynniku publicznym (public_exponent) równym 65537. Wykorzystuje bibliotekę cryptography, a dokładniej moduł rsa oraz serialization. Funkcja generuje i zapisuje zarówno klucz prywatny, jak i publiczny do odpowiednich plików w formacie PEM, co umożliwia ich późniejsze wykorzystanie w operacjach szyfrowania oraz deszyfrowania.

Funkcja deszyfrowania danych:

```
def decode_data(encrypted_message: bytes):
    with open("private_key.pem", "rb") as file:
        private_key = serialization.load_pem_private_key(file.read(),
password=None, backend=default_backend())
    decode_message = base64.b64decode(encrypted_message)
    encrypted_message = private_key.decrypt(decode_message,
padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),algorithm=hashes.S
HA256(), label=None))
```

```
    print(json.loads(encrypted_message.decode("utf-8")))
    return json.loads(encrypted_message.decode("utf-8"))
```

Funkcja wczytuje klucz prywatny z pliku, dekoduje zaszyfrowaną wiadomość przy użyciu tego klucza, a następnie zwraca odszyfrowany obiekt JSON.

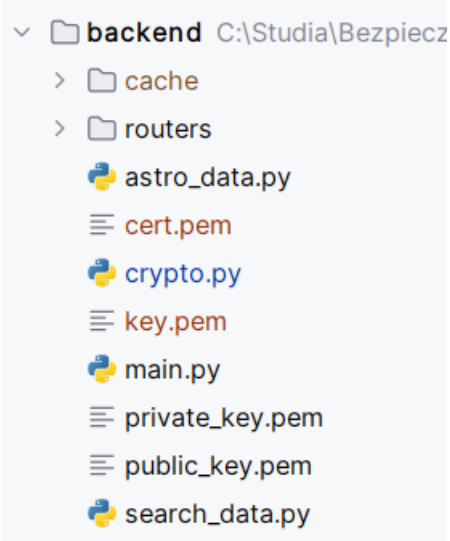
Funkcja wysyłająca klucz publiczny:

```
def public_key_send():
    with open("public_key.pem", "rb") as file:
        public_key = serialization.load_pem_public_key(file.read(),
backend=default_backend())
    public_key_pem =
public_key.public_bytes(encoding=serialization.Encoding.PEM,format=serializ
ation.PublicFormat.SubjectPublicKeyInfo)
    return public_key_pem
```

Funkcja public_key_send służy do wczytania klucza publicznego z pliku w formacie PEM i zwrócenia klucza publicznego jako ciągu bajtów w formie PEM.

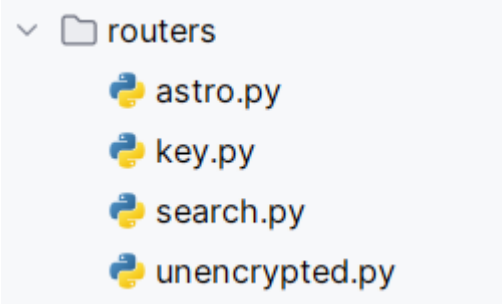
Struktura serwera:

Struktura katalogów oraz plików serwera odpowiedzialna za jej funkcjonowanie oraz wygląd jest umieszczona w katalogu głównym o nazwie „backend". (Kody źródłowe plików z poszczególnych katalogów znajdują się w załączniku).



Opis poszczególnych podkatalogów i plików:

1. Katalog routers - zawiera kody źródłowe odpowiedzialne za endpointy wystawione przez serwer.



2. Plik astro_data.py - zawiera kod źródłowy odpowiedzialny, za pobieranie danych o pogodzie, miastach. Plik zawiera trzy funkcje odpowiadające za pobieranie danych o pogodzie: weather(), air_quality(), get_city(). Przykładowy kod:

3.
```
def air_quality(lon: str, lat: str):
    params = {
        "latitude": lat,
        "longitude": lon,
        "current":"european_aqi",
        "forecast_days":"1",
        "timeformat":"iso8601",
        "timezone":"GMT"
      }
    url = "https://air-quality-api.open-meteo.com/v1/air-quality"

    data = requests.get(url, params=params).json()
    del data['latitude']
    del data['longitude']
    return data
```

   Funkcja air_quality służy do pobierania informacji o jakości powietrza dla określonych współrzędnych geograficznych (długość geograficzna i szerokość geograficzna).

4. Plik main.py - główny plik startowy aplikacji serwera napisanej w pythonie. Zawiera kod, która jest odpowiedzialny za uruchomienie serwera https.

Frontend

Aplikacja została zaimplementowana przy użyciu framework flutter oraz pakietów, potrzebnych do realizacji funkcjonowania aplikacji zgodnie z jej przeznaczeniem. Do najważniejszych fragmentów implementacji kodu, które nie stanowią wyłącznie implementacji graficznego interfejsu użytkownika, stanowią:

Pobieranie lokalizacji:

Pobieranie lokalizacji opiera się na wykorzystaniu wbudowanego w system android modułu GPS, do którego aplikacja ma dostęp poprzez uprawnienie oraz dzięki pakietowi „location". Wszystkie potrzebne funkcjonalności związane z lokalizacją są zaimplementowane w klasie „LocationService".

```
class LocationService{
  late Location location;
  late PermissionStatus _permissionStatus;
  late LocationData _locationData;
  late bool _serviceEnabled;

  LocationService(){
    location = Location();
  }

  Future<LocationData> getLocation() async{

    _serviceEnabled = await location.serviceEnabled();
    if(!_serviceEnabled){
      _serviceEnabled = await location.requestService();
    }

    _permissionStatus = await location.hasPermission();
    if(_permissionStatus == PermissionStatus.denied){
      _permissionStatus = await location.requestPermission();
    }

    _locationData = await location.getLocation();
    return _locationData;
  }
```

Klasa LocationService umożliwia zarządzanie lokalizacją w aplikacji Flutter, zapewniając dostęp do danych lokalizacyjnych oraz zarządzając uprawnieniami i usługą lokalizacji. Metoda „getLocation" jest głównym punktem dostępu do danych lokalizacyjnych i zapewnia, że aplikacja ma odpowiednie uprawnienia i włączoną usługę lokalizacji przed pobraniem lokalizacji. Metoda ta sprawdza, czy usługa lokalizacji jest włączona i czy aplikacja ma odpowiednie uprawnienia. Jeśli usługa lokalizacji nie jest włączona, próbuje ją włączyć. Jeśli brak uprawnień, próbuje uzyskać je od użytkownika. Następnie pobiera dane lokalizacyjne i je zwraca.

Zapytania do serwera:

Zapytania do serwera w celu uzyskania potrzebnych danych do funkcjonowania aplikacji, opiera się na standardowym tworzeniu zapytań kierowanych na endpointy serwera, przy użyciu protokołu https, która funkcjonalność została zaimplementowana w pliku o nazwie http_service.dart.

```
import 'package:http/http.dart' as http;

Future<http.Response?> responseBody(String urlAddress,[String unencodedPath =
'',Map<String,dynamic>? queryParams]) async {
  var url = Uri.https(urlAddress,unencodedPath);
  if(queryParams != null){
    var response = await http.post(
      url,
      headers: {"Content-Type": "application/json", "charset":"utf-8"},
      body: json.encode(queryParams)
    );
    if(response.statusCode == 200){
      return response;
    }
  }else{
    var response = await http.get(url);
    if(response.statusCode == 200){
      return response;
    }
  }
  return null;
}
```

Funkcja „responseBody” umożliwia pobieranie odpowiedzi HTTP z określonego adresu URL. Parametry funkcji:

- urlAddress: główny adres URL pod którym znajduje się serwer.
- unencodedPath: Ścieżka z której ma być pobrana odpowiedź HTTP.
- queryParams: Parametry zapytania w postaci mapy, które można przekazać razem z zapytaniem.

Funkcja ta tworzy URI na podstawie urlAddress i unencodedPath, a następnie wykonuje zapytanie HTTP GET lub POST, w zależności od dostępności queryParams. Jeśli queryParams są dostępne, zostanie użyte zapytanie POST z danymi przekazanymi jako JSON w ciele zapytania. W przeciwnym razie zostanie użyte zapytanie GET. Jeśli odpowiedź jest dostępna i ma kod stanu 200 (OK), zostanie ona zwrócona. W przeciwnym razie funkcja zwraca null.

Pobieranie informacji o pogodzie:

Pobieranie informacji o pogodzie jest realizowane za pomocą odpowiednich endpointów udostępnianych przez serwer oraz przy wykorzystaniu funkcji, realizującej zapytania do serwera. Wszystkie funkcjonalności odpowiadające za pobieranie pogody, są zaimplementowane w klasie „WeatherService". W tej klasie znajdują się funkcje:

```
  Future<Map<dynamic,dynamic>?> weatherInfo( String url, [String unencodePath
= '', Map<String, dynamic>? queryParams] ) async{

    if (queryParams != null && queryParams.containsKey("daily")) {
      String daily = queryParams["daily"] as String;
      queryParams["daily"] = daily;
    }

    if (queryParams != null && queryParams.containsKey("current")) {
      String current = queryParams["current"] as String;
      queryParams["current"] = current;
    }

    if (queryParams != null && queryParams.containsKey("hourly")) {
      String hourly = queryParams["hourly"] as String;
      queryParams["hourly"] = hourly;
    }

    var response = await responseBody(url,unencodePath,queryParams);
    var responseByte = utf8.decode(response!.bodyBytes);
    var modifyResponseByte = responseByte.replaceAll(r'\"','"');
    var convertToMap =
modifyResponseByte.substring(1,modifyResponseByte.length - 1);
    var decodedResponse = jsonDecode(convertToMap) as Map;
    return decodedResponse;
  }
```

Funkcja „weatherInfo" umożliwia pobieranie informacji pogodowych z serwera za pomocą zapytania HTTP. Parametry:

- url: Adres URL, z którego ma być pobrana informacja pogodowa.
- unencodedPath: Opcjonalna ścieżka URL, która może być niekodowana.
- queryParams: Opcjonalne parametry zapytania w postaci mapy, które można przekazać razem z zapytaniem.

Funkcja ta wykorzystuje funkcję responseBody do pobrania odpowiedzi HTTP z podanego url, unencodedPath i queryParams. Następnie dekoduje odpowiedź i konwertuje ją na mapę. Funkcja jest przystosowana do obsługi odpowiedzi, które zawierają dane pogodowe w formacie JSON. Zwraca mapę zawierająca informacje pogodowe lub null, jeśli wystąpił błąd w zapytaniu lub odpowiedź nie jest dostępna.

```
Future<dynamic> getPublicKey(String url,[String unencodePath = '']) async{
  var response= await responseBody(url,unencodePath);
  var decodedResponse = jsonDecode(response!.body) as Map;
  return decodedResponse;
}
```

Funkcja „gerPublicKey” służy do uzyskania publicznego klucza serwera w celu szyfrowania danych dla innych zapytań.

```
  Future<Map<dynamic,dynamic>?> weatherInfoUnencrypted( String url, [String
unencodePath = '', Map<String, dynamic>? queryParams] ) async{
    if (queryParams != null && queryParams.containsKey("daily")) {
      String daily = queryParams["daily"] as String;
      queryParams["daily"] = daily;
    }

    if (queryParams != null && queryParams.containsKey("current")) {
      String current = queryParams["current"] as String;
      queryParams["current"] = current;
    }

    if (queryParams != null && queryParams.containsKey("hourly")) {
      String hourly = queryParams["hourly"] as String;
      queryParams["hourly"] = hourly;
    }

    var response = await responseBody(url,unencodePath,queryParams);
    var responseByte = utf8.decode(response!.bodyBytes);
    var modifyResponseByte = responseByte.replaceAll(r'\"','"');
    var decodedResponse = jsonDecode(modifyResponseByte) as Map;
    return decodedResponse;
  }
```

Funkcja „weatherInfoUnecrypred” jest tak naprawdę przeciążeniem funkcji „weatherInfo”, która realizuje podobną funkcjonalność, z różnicą, że dane, które są przekazywane do serwera za pośrednictwem tej funkcji nie są w żadnym stopniu szyfrowane.

Szyfrowanie danych:

Szyfrowanie danych lokalizacyjnych, które są przekazywane do serwera, używa algorytmu RSA z kodowaniem OAEP i funkcją skrótu SHA-256 do zabezpieczenia tekstu. Wszystkie zadania realizujące szyfrowanie są zaimplementowane w funkcji „encryption", która wykorzystuje bezpieczne funkcję pakietu „encrypt".

```
crypto.Encrypted encryption(String publickKey, String plainText){

  final publicKey = crypto.RSAKeyParser().parse(publickKey) as RSAPublicKey;
  final encrypter = crypto.Encrypter(
                      crypto.RSA(
                        publicKey: publicKey,
                        encoding: crypto.RSAEncoding.OAEP,
                        digest: crypto.RSADigest.SHA256
                      )
                    );
  var encrypted = encrypter.encrypt(plainText);
  return encrypted;
}
```

Funkcja encryption umożliwia szyfrowanie tekstu przy użyciu klucza publicznego RSA. Parametry:

- publicKey: Klucz publiczny RSA w formie tekstowej, który będzie używany do szyfrowania.
- plainText: Tekst, który ma zostać zaszyfrowany.

Funkcja ta przyjmuje klucz publiczny RSA w formie tekstowej oraz tekst, który ma zostać zaszyfrowany. Następnie używa klucza publicznego do szyfrowania tekstu przy użyciu algorytmu RSA. Funkcja zwraca zaszyfrowany tekst.

Przechowywanie danych lokalnie:

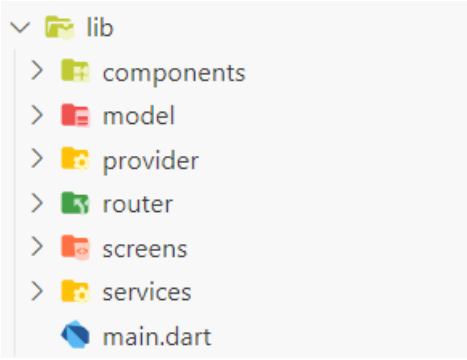
Ustawienia aplikacji takie jak na przykład jednostka temperatury, prędkości wiatru itp. Są lokalnie przechowywane w systemie w „shared preferences", który jest lokalnym magazynem danych dla aplikacji na androida. Wszystkie potrzebne funkcjonalności związane z przechowywaniem danych są zaimplementowane w klasie „LocalStorageService". Klasa wykorzystuje pakiet „shared_preferences", do zarządzania danymi lokalnymi.

```
class LocalStorageService{
  final Map<String,dynamic> _defaultSettings = {};
  late bool _securityDisabled;


  Map<String,dynamic> get getSettings => _defaultSettings;
  bool get getSecurityStatus => _securityDisabled;


  Future<void> loadlLocalStorage() async {
    SharedPreferences _prefs = await SharedPreferences.getInstance();
    String? temperature = _prefs.getString("temperatureUnit");
    String? utniOfWindSpeed = _prefs.getString("utniOfWindSpeed");
    String? unitOfAtmosphericPressure =
_prefs.getString("unitOfAtmosphericPressure");
    bool? autoUpdate = _prefs.getBool("autoUpdate");
    bool? soundEffect = _prefs.getBool("soundEffect");
    bool? security = _prefs.getBool("security");


    _defaultSettings["temperatureUnit"] = temperature ?? "ºC";
    _defaultSettings["utniOfWindSpeed"] = utniOfWindSpeed ?? "Kilometrów na
godzinę (km/h)";
    _defaultSettings["unitOfAtmosphericPressure"] = unitOfAtmosphericPressure
?? "Milibar (mbar)";
    _defaultSettings["autoUpdate"] = autoUpdate ?? false;
    _defaultSettings["soundEffect"] = soundEffect ?? false;
    _defaultSettings["security"] = security ?? false;
    _securityDisabled = security ?? false;

  }
}
```

Funkcja „loadLocalStorage" jest odpowiedzialna za wczytywanie ustawień aplikacji z lokalnego magazynu SharedPreferences. Następnie ustawia te wartości jako domyślne ustawienia aplikacji. Wartość zabezpieczeń jest również odczytywana i przypisywana do odpowiedniego pola.

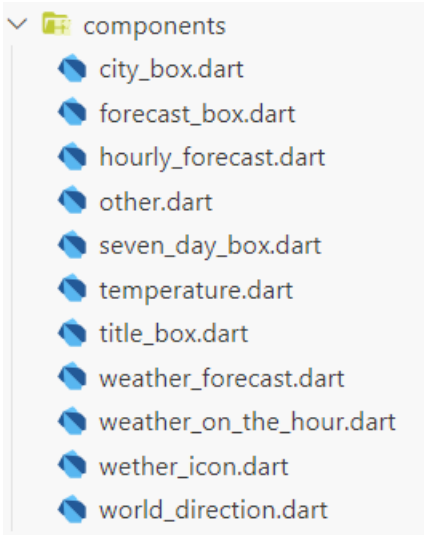
Struktura aplikacji:

Struktura katalogów oraz plików aplikacji odpowiedzialna za jej funkcjonowanie oraz wygląd jest umieszczona w katalogu głównym o nazwie „Lib”. (Kody źródłowe plików z poszczególnych katalogów znajdują się w załączniku)



Opis poszczególnych podkatalogów i plików:

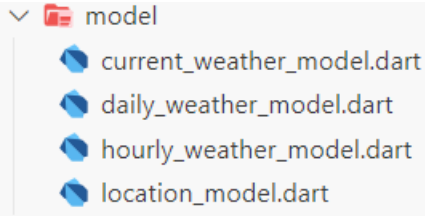
1. Katalog components - zawiera komponenty, części interfejsu użytkownika, które są wielokrotnie używane w różnych częściach aplikacji. Te komponenty są niezależnymi fragmentami kodu źródłowego, które można łatwo dostosowywać i ponownie wykorzystywać w różnych ekranach aplikacji. Katalog ten zawiera kody źródłowe następujących komponentów:



2. Katalog model – zawiera kody źródłowe poszczególnych klas, które są odpowiedzialne za reprezentację poszczególnych (własnych) struktur danych, stworzonych na potrzeby aplikacji. Przykładowa implementacja modelu wygląda następująco:

```
class HourlyWeatherModel{

  final String time;
  final double temperature;
  final int weathercode;
  final double windspeed;

  HourlyWeatherModel({
    required this.time,
    required this.temperature,
    required this.weathercode,
    required this.windspeed
  });

  String get getTime => time;
  int get getWeathercode => weathercode;
  double get getTemperature => temperature;
  double get getWindspeed => windspeed;
}
```

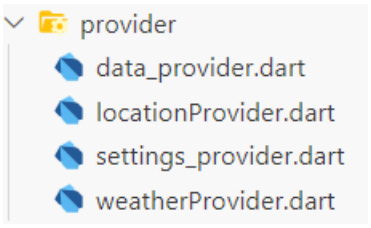
3. Klasy odpowiadające za model struktury danych zawierają wyłącznie prywatne pola, określające typ oraz nazwę przechowywanych danych, konstruktor oraz metody służące do pobierania danych. Pełna zawartość katalogu model jest następująca:



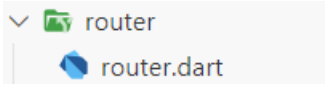
4. Katalog provider – zawiera implementacje tak zwanych „dostawców danych”. Pliki w tym folderze są odpowiedzialne za przechowywanie oraz udostępnianie danych w wyznaczonym obrębie aplikacji. Przykładowa implementacja providera wygląda następująco:

```
class WeatherProvider extends ChangeNotifier{

  late List<DailyWeatherModel> _dailyWeatherList ;
  late CurrentWeatherModel _currentWeather;
  late List<HourlyWeatherModel> _hourlyWeatherList;
  DailyWeatherModel getDailyweather(int index){ return
_dailyWeatherList[index]; }
  CurrentWeatherModel getCurrentWeather(){ return _currentWeather; }
  HourlyWeatherModel getHourlyWeather(int index){ return
_hourlyWeatherList[index]; }
  List<HourlyWeatherModel> getHourlyWeatherList() { return
_hourlyWeatherList;}
  void addDailyweather(List<DailyWeatherModel> dailyWeatherModel)
async{
    _dailyWeatherList = dailyWeatherModel;
    notifyListeners();
  }

  void addCurrentWeather(CurrentWeatherModel currentWeatherModel)
async{
    _currentWeather = currentWeatherModel;
    notifyListeners();
  }

  void addHourlyWeather(List<HourlyWeatherModel> hourlyWeatherModel)
async{
    _hourlyWeatherList = hourlyWeatherModel;
    notifyListeners();
  }

}
```

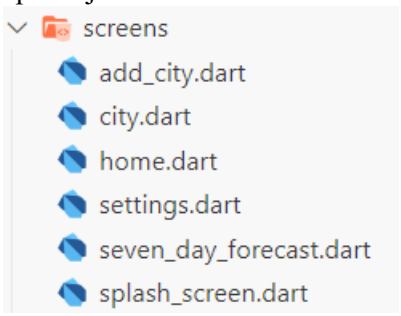
5. Każda klasa w katalogu provider dziedziczy klasę „ChangeNotifier”, która zmienia klasę na dostawcę danych. Dzięki temu wszystkie pola oraz metody tej klasy mogą być wykorzystane w każdym, wyznaczonym miejscu w kodzie aplikacji. Dane, które zostaną zmienione w jednym miejscu, dzięki dostawcy danych są również widziane w innych miejscach automatycznie. Pełna zawartość katalogu provider jest następująca:



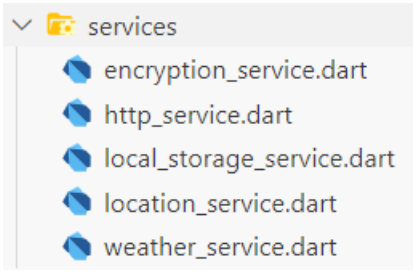
6. Katalog router – zawiera plik odpowiedzialny za nawigację po ekranach aplikacji.



7. Katalog screens – zawiera kody źródłowe interfejsu użytkownika dla poszczególnych ekranów aplikacji.



8. Katalog services – zawiera pliki, które są odpowiedzialne za implementację funkcjonalności aplikacji takich jak: pobieranie danych pogodowych, lokalizacji itd.



9. Plik main.dart – główny plik startowy aplikacji napisanej w Flutter. Zawiera funkcję main, która jest funkcja startową oraz zdefiniowanie listy dostawców danych, które będą widoczne w obrębie całej aplikacji i ekranu startowego aplikacji.

```
void main() {

  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider( create:(context)=> LocationProvider() ),
        ChangeNotifierProvider( create:(context)=> WeatherProvider() ),
        ChangeNotifierProvider( create:(context)=> SettingsProvider() ),
        ChangeNotifierProvider( create:(context)=> DataProvider() )
      ],
      child: const SplashScreen(),
    )
  );
}
```

Instrukcja obsługi „Dla Developera”

Narzędzia:

1. IDE:
    a. Visual Studio Code (wersja 1.84),
    b. PyCharm (wersja 2023.3.2),
    c. Android Studio (wersja 2022.3.1 Patch 2)
2. Zarządzanie kontrolą wersji – GitHub

Środowisko:

1. System Operacyjny – Windows 10 lub nowszy
2. Wersja języka programowania:
    a. Python – 3.12.0 lub nowszy,
    b. Flutter – 3.13.8,
    c. Dart – 3.1.4,
    d. Android API – 34

Kompilacja:

1. Sklonuj repozytorium projektu z systemu kontroli wersji,
2. Przejdź do katalogu z projektem,
3. W PyCharm otwórz katalog „Backend”,
4. Otwórz plik main.py i kliknij Run,
5. Uruchom android studio i przygotuj instancję emulatora z wersją android API minimum 34,
6. Jeżeli nie korzystasz z Visual Studio Code idź do kroku 10
7. W Visual Studio Code otwórz plik main.dart,
8. W prawym dolnym rogu kliknij na „No device” i z listy wybierz swój emulator
9. Kliknij w prawym górnym rogu przycisk „run without debbuging”
10. Opcjonalnie dla kroku 4: Uruchom emulator,
11. Opcjonalnie dla kroku 4: W terminalu uruchom komendę „Flutter run”