Home.dart  
  
The code imports necessary packages and files related to Flutter and weather data.

* The **HomeScreen** class is a stateful widget that represents the home screen of the application.
* Inside the **HomeScreen** class, there is a private state class **\_HomeScreenState** that extends the **State** class.
* The **\_HomeScreenState** class includes a **searchController** to handle the text input for city names, a **permission** variable to track device location permission, and a **cityName** variable of type **ValueNotifier<String>** to store the current city name.
* The **devicePermission** method uses the **DeviceLocation** class to check if the device has location permission.
* The **initWeather** method retrieves the current device location using the **DeviceLocation** class and then calls the **getWeather** method from **WeatherServices** to fetch the weather data based on the latitude and longitude.
* The **searchWeatherByPlace** method takes a place (city name) as input, uses the **DeviceLocation** class to retrieve the latitude and longitude of the place, and then calls the **getWeather** method from **WeatherServices** to fetch the weather data.
* The **weatherFuture** variable is declared as a **Future<Weather>** to store the future result of fetching the weather data.
* In the **initState** method, the **devicePermission** method is called to check for location permission, and then the **initWeather** method is called to initialize the **weatherFuture** variable with the weather data.
* The **dispose** method is responsible for disposing of the **searchController** and **cityName** objects.
* The **build** method constructs the UI of the home screen using a **Scaffold** widget with an **AppBar** and a **SafeArea** container for the main content.
* Inside the **SafeArea**, there is a **Container** widget with a background image and padding.
* The UI consists of a **TextField** for entering city names and a search button. The **ValueListenableBuilder** widget is used to listen to changes in the **cityName** value and display the weather information accordingly.
* If a city name is entered, the **WeatherDisplay** widget is instantiated with the **searchWeatherByPlace** future as the **future** parameter. Otherwise, the **WeatherDisplay** widget is instantiated with the **weatherFuture** as the **future** parameter.

Location.dart

* The **DeviceLocation** class is a singleton class that provides access to location services.
* The **\_instance** variable is a private static instance of the **DeviceLocation** class, which ensures that only a single instance of the class can exist.
* The **get instance** getter returns the **\_instance** variable, allowing other parts of the code to access the singleton instance.
* The **DeviceLocation** class has a private constructor **\_internal()** to prevent direct instantiation of the class.
* The **checkPermission** method is used to check if the location services are enabled and if the required permissions are granted.
* It first checks if location services are enabled using **Geolocator.isLocationServiceEnabled()**.
* If location services are disabled, it throws an error indicating that location services are disabled.
* It then checks the location permission using **Geolocator.checkPermission()**.
* If the permission is denied, it requests permission using **Geolocator.requestPermission()**.
* If the permission is still denied, it throws an error indicating that location permissions are denied.
* If the permission is permanently denied, it throws an error indicating that location permissions are permanently denied and cannot be requested.
* Finally, if the permission is either **LocationPermission.always** or **LocationPermission.whileInUse**, it returns **true** indicating that the permission is granted.
* The **determinePosition** method is used to fetch the current device position using **Geolocator.getCurrentPosition()**.
* The **getPlacePosition** method takes a **cityName** as input and returns the position (latitude and longitude) of that place.
* It uses the **locationFromAddress** function from the **geocoding** package to convert the **cityName** to a list of **Location** objects.
* The **localeIdentifier** parameter is set to **"en\_US"** to ensure consistent geocoding results.
* It returns the first location from the list.

Weather.dart/services

* The **WeatherServices** class is a singleton class that provides weather-related functionality.
* The **\_instance** variable is a private static instance of the **WeatherServices** class, which ensures that only a single instance of the class can exist.
* The **get instance** getter returns the **\_instance** variable, allowing other parts of the code to access the singleton instance.
* The **WeatherServices** class has a private constructor **\_internal()** to prevent direct instantiation of the class.
* The **getWeather** method is used to fetch weather data based on the provided latitude and longitude.
* It takes **lat** (latitude) and **long** (longitude) as input parameters.
* It constructs the URL for the weather API using the provided latitude, longitude, and an API key.
* It sends an HTTP GET request to the constructed URL using the **http** package.
* If the response status code is 200 (indicating a successful response), it parses the JSON response body using **json.decode** and converts it into a **Map**.
* It then uses the **Weather.fromJson** method to create a **Weather** object from the parsed JSON map.
* Finally, it returns the **Weather** object.
* If the response status code is not 200, it throws an exception indicating that the weather data failed to load.

Weather.dart/models

* The **Weather** class has several properties representing different weather attributes such as **weatherID**, **cityName**, **longitude**, **latitude**, **mainWeather**, **temperature**, **feelsLike**, **humidity**, **windSpeed**, **sunrise**, and **sunset**.
* **weatherID** is an optional integer representing the weather ID.
* **cityName** is a required string representing the name of the city.
* **longitude** and **latitude** are required doubles representing the coordinates of the location.
* **mainWeather** is a required string representing the main weather condition.
* **temperature** is a required double representing the temperature.
* **feelsLike** is a required double representing the perceived temperature.
* **humidity** is a required integer representing the humidity percentage.
* **windSpeed** is a required double representing the wind speed.
* **sunrise** and **sunset** are required **DateTime** objects representing the time of sunrise and sunset respectively.
* The **Weather** class includes a named constructor **Weather** that initializes the properties with the provided values.
* The **toJson** method converts the **Weather** object to a **Map** representation.
* It returns a **Map** containing key-value pairs representing each property of the **Weather** object.
* The **DateTime** objects (**sunrise** and **sunset**) are converted to ISO 8601 string representations using the **toIso8601String** method.
* The **fromJson** factory constructor creates a **Weather** object from a JSON map.
* It takes a **Map<String, dynamic>** as input, which represents the parsed JSON data.
* It retrieves the required values from the JSON map and constructs a **Weather** object.
* The **coord** variable is used to extract the **lon** (longitude) and **lat** (latitude) values from the **coord** field of the JSON map.
* The values are assigned to the corresponding properties of the **Weather** object.
* The sunrise and sunset values are converted from milliseconds since epoch to **DateTime** objects using **DateTime.fromMillisecondsSinceEpoch**.
* The constructed **Weather** object is then returned.

Weather\_display.dart

* The **WeatherDisplay** widget is a stateless widget that takes a **Future<dynamic>** as input in its constructor. This future represents the weather data to be displayed.
* Inside the **build** method, the **SingleChildScrollView** widget is used to enable scrolling if the weather display overflows the screen.
* The **FutureBuilder** widget is used to handle the asynchronous loading of the weather data.
* The **future** property of **FutureBuilder** is set to the provided **future** value.
* In the **builder** callback, the snapshot of the future is checked to determine the state of the asynchronous operation.
* If the snapshot does not have data yet, a circular progress indicator is displayed.
* If the snapshot has data, the weather information is extracted from the snapshot and assigned to the **Weather** object.
* Temperature values are converted from Kelvin to Celsius by subtracting 273.15.
* Sunrise and sunset times are formatted using the **\_formatTime** method.
* The weather information is displayed using a column layout and various **Container** widgets.
* The city name is displayed in a container with white text.
* The temperature is displayed in a colored container with white text.
* The "Feels Like" temperature and humidity are displayed in separate containers.
* The longitude and latitude values are displayed in separate containers.
* The sunrise and sunset times are displayed in separate containers.
* The **\_formatTime** method is used to format the time in the format "HH:MM AM/PM". It takes a **DateTime** object as input and returns the formatted string.
* The **\_formatTime** method is used to format a **DateTime** object into a string representation of the time in the format "HH:MM AM/PM". Let's break down how it works:
* The method takes a **DateTime** object called **dateTime** as input.
* It checks whether the hour of **dateTime** is greater than 12. If so, it subtracts 12 from the hour to convert it to a 12-hour format. Otherwise, it keeps the hour as is.
* This is done using a ternary operator: **dateTime.hour > 12 ? dateTime.hour - 12 : dateTime.hour**
* The minute component of **dateTime** is converted to a string and padded with a leading zero if necessary to ensure a two-digit representation. This is done using the **toString()** and **padLeft()** methods.
* **dateTime.minute.toString().padLeft(2, '0')**
* The period (AM/PM) is determined based on whether the hour of **dateTime** is greater than or equal to 12. If the hour is greater than or equal to 12, the period is set to 'PM'; otherwise, it is set to 'AM'.
* This is done using a ternary operator: **dateTime.hour >= 12 ? 'PM' : 'AM'**
* Finally, the formatted time string is constructed by concatenating the hour, minute, and period together with appropriate spacing and returning the result.
* The string is constructed using string interpolation: **'$hour:$minute $period'**