A képen növény, zöld, kültéri, levél látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

*| Janos Orsos| Flora Mixtaj| Tamas Bito|*

*GitHub:*

*https://github.com/Jani1125/PlantCare.git*

*PlantCare App*

*Documentation*

Contents

[Android Assignment Requirements 3](#_Toc197891668)

[Implementation 3](#_Toc197891669)

[Application Implementation 3](#_Toc197891670)

[Overview 4](#_Toc197891671)

[Development Environment – Android Studio 5](#_Toc197891672)

[User Interface 6](#_Toc197891673)

[Fragment 10](#_Toc197891674)

[FavPlantFragment 10](#_Toc197891675)

[GoogleSignFragment 10](#_Toc197891676)

[MessagesFragment 11](#_Toc197891677)

[OwnPlantFragment 11](#_Toc197891678)

[PlantScannerFragment 12](#_Toc197891679)

[SearchFragment 13](#_Toc197891680)

[SettingsFragment 13](#_Toc197891681)

[MenuManager 14](#_Toc197891682)

[Database 15](#_Toc197891683)

[NotificationDao 15](#_Toc197891684)

[OwnPlantDao 15](#_Toc197891685)

[PlantDao 16](#_Toc197891686)

[PlantDatabase 16](#_Toc197891687)

[Watering 17](#_Toc197891688)

[Entity 18](#_Toc197891689)

[BasicPlant 18](#_Toc197891690)

[Notification 18](#_Toc197891691)

[OwnPlant 19](#_Toc197891692)

[Plant 20](#_Toc197891693)

[Adapter 22](#_Toc197891694)

[BasicPlantAdapter 22](#_Toc197891695)

[NewPlantAdapter 22](#_Toc197891696)

[NotificatonAdapter 23](#_Toc197891697)

[OwnPlantAdapter 24](#_Toc197891698)

[OwnPlantDetailsAdapter 25](#_Toc197891699)

[PlantDetailsAdapter 26](#_Toc197891700)

[MenuAdapter 27](#_Toc197891701)

[API 28](#_Toc197891702)

[ApiService 28](#_Toc197891703)

[DetailsResultCallBack 28](#_Toc197891704)

[SearchResultCallBack 28](#_Toc197891705)

[Util 29](#_Toc197891706)

[NotificationScheduler 29](#_Toc197891707)

[PlantWateringReceiver 30](#_Toc197891708)

[Activity 31](#_Toc197891709)

[MainActivity 31](#_Toc197891710)

[SplashActivity 32](#_Toc197891711)

[Manifest 32](#_Toc197891712)

[Key Elements 32](#_Toc197891713)

[Package Declaration 32](#_Toc197891714)

[Permissions 32](#_Toc197891715)

[Application Tag 33](#_Toc197891716)

[Activities 33](#_Toc197891717)

[Target API 33](#_Toc197891718)

A képen szöveg, Grafika, képernyőkép, kör látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

# Android Assignment Requirements

The task selected by the students must be discussed with the instructor beforehand to ensure that the application does not have too many or too few features. The project is to be completed in groups, with each group consisting of 2-4 students.

## Implementation

* **Android Studio** development environment
* **Android SDK**
* **Java programming language**
* **Git version control** (using Git Bash, Sourcetree, or other tools is recommended)
  + Continuous commits from all students
  + Only commits found on the server are counted (merge commits are not counted)
  + The project must be stored on **GitLab** online
  + The instructor must be invited with the **Reporter** role
* The **readme.md** file on GitLab must contain the goal of the application and a description of the features available to the user
* The application should be optimized for one resolution and one device
* A separate landscape view is only necessary in justified cases (e.g., for games where only landscape mode is used)

## Application Implementation

* The application must be implemented with **one Activity** and multiple **Fragments**
  + In justified cases, more Activities may be used, but this must be discussed with the instructor beforehand
* Interfaces should be created using **ConstraintLayout**
* Must include list and detailed views
  + Depending on the task, functionalities such as creating new items/objects, modifying existing objects, and deleting from a list
  + Depending on the task, search and sorting functionalities
* Lists should be implemented using **RecyclerView**
* Some screens should be accessible through **Tab**, **Action Bar**, or **Navigation Drawer** solutions
* If using a local database (on the device), the **Architecture Components Room** solution is mandatory
* For external server-side application interaction (REST API), the use of **Retrofit** or **Volley** is mandatory
* High-resolution image resources should be loaded using the **Glide** library

# Overview

**PlantCare** is a modern Android application designed to assist users in taking better care of their houseplants through smart reminders, detailed plant information, and a user-friendly experience. The main goal of the app is to make plant care simpler and more enjoyable for both beginners and experienced plant lovers. With PlantCare, users can explore a built-in plant database, search for specific plants, and access comprehensive care details such as watering frequency, sunlight preferences, temperature needs, and more.

The application allows users to build and manage a personal plant collection, where each plant can be given a custom name and tracked over time. It automatically generates tailored watering schedules and sends notification reminders to help users stay on top of their plant care routines. Additionally, users can mark plants as favorites, making it easier to revisit the ones they care about most.

One of the key features of PlantCare is its offline functionality. All user data and plant information are stored locally using Room Database, ensuring the app remains fully usable even without an internet connection. For discovering unknown plants, the app includes a plant scanner feature: users can take a picture of a plant, and the app will identify it using an integrated plant recognition API, offering immediate suggestions and care guidance.

With its intuitive interface, compatibility with dark and light modes, and focus on smooth usability, PlantCare combines technology and nature into one seamless experience — turning everyday plant care into a guided and rewarding habit.

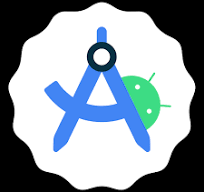
# Development Environment – Android Studio

The application was developed using **Android Studio**, which is the official integrated development environment (IDE) for Android application development, provided by Google. Android Studio is based on the **JetBrains IntelliJ IDEA** platform and is specifically optimized for Android development.

The environment offers numerous features that support efficient and structured development, including:

* **Built-in Emulators**: During development, Android emulators can be used to test the app on different devices and screen resolutions.
* **Layout Editor**: A visual interface editor that supports ConstraintLayout for designing responsive user interfaces.
* **Gradle Build System**: A flexible and customizable build system that manages configurations and dependencies.
* **Real-time Debugging**: Integrated debugging tools allow developers to trace and fix errors effectively.
* **Android Virtual Device (AVD) Support**: Enables development and testing using virtual devices.
* **Git Version Control Integration**: Supports version tracking and collaboration directly from within the IDE.

For this project, we used the **Java programming language**, the **Android SDK**, and the **Android Architecture Components**



# User Interface



The **user interface**  of the **PlantCare** application was designed using **Figma.**

**These are the following screens:**

A képen szöveg, képernyőkép, embléma, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**A képen szöveg, képernyőkép, Betűtípus, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.A képen képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.activity\_splash.xml fragment\_google\_sign\_in.xml activity\_main.xml**

**A képen szöveg, képernyőkép, Betűtípus, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.menu\_item.xml**

**profile\_menu.xml**

A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**A képen szöveg, képernyőkép, Grafikus tervezés, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.fragment\_search.xml plant\_items\_details.xml**

**A képen szöveg, képernyőkép, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**fragment\_fav\_plant.xml**

**A képen szöveg, képernyőkép, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**fragment\_own\_plant.xml form\_field\_item.xml own\_plant\_details.xml**

**A képen szöveg, képernyőkép, növény, szobanövény látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.A képen szöveg, növény, szobanövény, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**A képen szöveg, képernyőkép, növény, szobanövény látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**fragment\_plant\_scanner.xml fragment\_settings.xml**

**A képen szöveg, képernyőkép, multimédia, válaszfal látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**A képen szöveg, képernyőkép, zöld látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**fragment\_messages.xml popup\_message.xml**

**A képen szöveg, Betűtípus, képernyőkép, fehér látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**A képen szöveg, képernyőkép, Téglalap, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

# Fragment

## FavPlantFragment

The FavPlantFragment is a part of an Android application that allows users to view a list of plants, see their basic details, and manage their favorites. This fragment interacts with a local database (PlantDatabase) to load plant data and handle changes to the user's favorite plants list.

The main features of this fragment are the display of basic plant information, the ability to view detailed plant information, and the functionality to add or remove plants from the favorites list. The fragment uses a RecyclerView to display the plants in a list. Each item in the list is represented by a BasicPlant object that contains basic details about the plant, such as its common name, scientific name, and image URL.

The fragment interacts with the database through the PlantDao, which is responsible for fetching the plant data. It runs background tasks to load the data so that the UI remains responsive. When the fragment is created, the method getItemsFromDb() is called, which retrieves all the plants from the database and updates the RecyclerView with the list.

Each plant in the list can be clicked on, and when a user selects a plant, the fragment fetches detailed information about the selected plant using the getDetailsFromDb() method. This detailed information includes aspects like the plant's watering needs, type, cycle, and a description. This detailed view is displayed in the RecyclerView using a different adapter, the PlantDetailAdapter.

One of the key features of the FavPlantFragment is the ability for users to mark a plant as a favorite. In the detailed view of each plant, there is a button that toggles the plant's favorite status. If the plant is not yet marked as a favorite, clicking the button will add it to the list of favorites in the database. If the plant is already a favorite, clicking the button will remove it from the favorites list. These updates to the favorite status are handled in a background thread to ensure that the UI remains smooth and responsive.

Additionally, the fragment includes a side menu, which is set up using the MenuManager. This menu allows users to navigate easily between different sections of the app, such as the Home screen, Search, and Settings.

When the fragment is, the onResume() method reloads the plant data to ensure that the list of plants is up to date, reflecting any changes made while the fragment was not visible.

## GoogleSignFragment

The GoogleSignInFragment handles the Google sign-in process within the PlantCare app. It facilitates users signing in using their Google accounts, which is a standard authentication method for Android apps.

The fragment initializes Google Sign-In using GoogleSignInOptions. The options specify that the app requires the user’s email and the ID token for authentication. The requestIdToken() method uses a Web Client ID defined in the app's strings.xml.

The GoogleSignInClient is then configured to manage the sign-in process.

The SignInButton from the Google Play services library is used to provide a standard Google Sign-In button. When the button is clicked, the signIn() method is called, which triggers the sign-in process by calling getSignInIntent() on the GoogleSignInClient.

The signIn() method starts an activity to launch the Google Sign-In process. This opens the Google Sign-In UI where the user can select their Google account or sign in. The result of this sign-in attempt is handled in the onActivityResult() method, which checks if the result corresponds to the sign-in request.

The result of the sign-in attempt is passed as a Task<GoogleSignInAccount> to the handleSignInResult() method. If the sign-in is successful (no exceptions are thrown), the GoogleSignInAccount object is passed to the updateUI() method to update the UI with user details. If the sign-in fails (due to network issues, wrong credentials, or other reasons), the exception is caught in the catch block, and the error is handled appropriately.

Different types of errors are handled using the handleSignInError() method. The method checks the error code returned by Google Sign-In and shows relevant messages:

* If the sign-in is canceled, a "Sign-in canceled" message is displayed.
* If the sign-in fails, a "Sign-in failed" message is shown, prompting the user to try again.

If the sign-in is successful, the updateUI() method displays a welcome message with the user's display name and proceeds to the main activity (MainActivity). The current activity (the Google Sign-In fragment) is then finished using requireActivity().finish(), ensuring the user does not return to the sign-in fragment once signed in.

## MessagesFragment

The MessagesFragment is responsible for displaying user notifications within the app. It uses a RecyclerView to show a list of notifications, which are retrieved from a local database (PlantDatabase). The fragment fetches the notifications using a background thread to avoid blocking the main UI, and once the data is fetched, it updates the UI on the main thread.

The fragment includes a NotificationAdapter that binds the list of notifications to the RecyclerView. When the fragment is created or resumed, the notifications are loaded and displayed to the user. The fragment also integrates a navigation menu using MenuManager, allowing users to easily switch between different sections of the app, such as Home, Search, and Favorite Plants.

## OwnPlantFragment

The OwnPlantFragment is a part of an Android application that allows users to manage their own plants, including adding new plants, viewing details, and deleting plants. It interacts with a local database (PlantDatabase) to store and retrieve plant data, and it displays the plants using a RecyclerView. The fragment also includes the ability to add new plants by selecting an image and filling out a form, and it supports the scheduling of watering notifications.

The OwnPlantFragment interacts with the PlantDatabase and its OwnPlantDao to fetch and manage the user's plant data. It retrieves the list of owned plants and updates the UI accordingly. It also handles adding, updating, and deleting plants in the database.

The fragment uses a RecyclerView to display the list of owned plants. It provides functionality to view a list of owned plants, view detailed information about a selected plant, and delete plants.

The user can add a new plant by clicking the "Add Plant" button. When clicked, a new adapter (NewPlantAdapter) is used to show a form where the user can fill out details about the plant, such as selecting an image. The user can also go back to the list of owned plants by clicking the "Back" button, and finally, the "Create" button saves the new plant to the database.

To add a new plant, the user must select an image for the plant. The OwnPlantFragment uses the ActivityResultLauncher to launch an image picker, allowing the user to select an image from their device. The selected image's URI is stored and passed to the adapter to display the image path.

When adding a new plant, the fragment checks if notifications are enabled in the app's preferences. If notifications are enabled, it schedules a watering notification for the new plant using the NotificationScheduler class.

Each owned plant in the list has an option to delete it. When the user deletes a plant, the fragment cancels any scheduled watering notifications and removes the plant from the database. The list is then updated.

When a plant is selected from the list, the fragment retrieves the plant's detailed information from the database. A new adapter (OwnPlantDetailsAdapter) is used to display the plant's details, including the plant's common name, watering needs, and other information. The user can go back to the plant list view by clicking the "Back" button.

The createButton's OnClickListener inserts the new plant into the database in a background thread. It also handles scheduling notifications if they are enabled. The OnDeleteClickListener deletes the plant from the database and updates the plant list in the UI. The GetAllOwnPlantFromDb() method fetches all owned plants from the database, and the GetOwnPlantDetailsFromDb() method fetches detailed information about a selected plant.

## PlantScannerFragment

The **PlantScannerFragment** enables users to identify plants by capturing their images with the camera. The fragment uses Google's **ML Kit** to label the detected plants and displays the results in real-time. It also handles camera permissions and sets up the necessary components for plant detection.

When the fragment is created, it checks if the app has the required camera permissions using ContextCompat.checkSelfPermission(). If permission is granted, the camera starts using the startCamera() method. If not, the app requests the permission using ActivityCompat.requestPermissions().

The camera preview is displayed in a **PreviewView**, and the detected plant name is shown in a **TextView**. The camera is initialized with the ProcessCameraProvider class, which provides a lifecycle-aware camera implementation.

The **Preview** object is used to display a real-time feed from the camera on the screen. The **ImageAnalysis** object is configured to analyze each frame from the camera feed. The frames are processed using an analyzer set with the cameraExecutor.

For each frame captured by the camera, the analyzer processes the image using Google **ML Kit's ImageLabeler** with ImageLabelerOptions.DEFAULT\_OPTIONS. This is used to identify the plant in the image. When the image is processed successfully, it returns labels (plant names) with confidence scores. The first label is displayed in the **TextView** as the detected plant.

## SearchFragment

The **SearchFragment** enables users to search for plants based on keywords and explore their details. It integrates with an external plant API to fetch data and uses a local database to store favorite plants.

The fragment includes a **SearchView** where users can input a search query (a plant name or keyword). Upon submitting the query, the searchWithApi() method sends a request to an external API (perenual.com) to fetch plants that match the search keyword. The results are then displayed in a **RecyclerView** using the **BasicPlantAdapter**.

The results from the API (matching plants) are shown in the RecyclerView. The **BasicPlantAdapter** binds the data, displaying basic information like the plant’s common name, scientific name, and image.

When a user clicks on a plant in the search results, the getDetailsWithApi() method is triggered to fetch detailed information about the selected plant. First, it checks whether the plant’s details are already stored in the local database (for favorite plants). If found, it loads the data from there, avoiding unnecessary API calls. If the plant is not found in the database, an API call is made to fetch the details and display them to the user using the **PlantDetailAdapter**.

In the plant detail view, there is a button (like a switch) to mark the plant as a favorite. When clicked, the OnFavoriteClick() method toggles the plant’s favorite status. If the plant is not already marked as a favorite, it will be added to the database as a favorite. If it is already a favorite, it will be removed from the database.

The user types a keyword into the **SearchView**, triggering an API call. Matching plants are displayed in a list. Clicking on a plant displays detailed information, either loaded from the local database or fetched via the API. The user can toggle the plant’s favorite status, which is updated in the local database.

## SettingsFragment

The SettingsFragment is used to manage user preferences in the application, where users can toggle features like dark mode and watering reminders.

The fragment includes a switch that allows users to toggle between light and dark themes. Upon view creation, it reads the current theme preference from SharedPreferences and updates the switch accordingly. When the switch is changed by the user, the preference is saved, and the app's theme is immediately updated using AppCompatDelegate.setDefaultNightMode().

Another switch controls whether watering reminder notifications are enabled. This setting is also retrieved from SharedPreferences. If the user disables the notifications, the app launches a background thread to cancel all scheduled notifications using plant IDs fetched from the local database (OwnPlantDao). If the user enables notifications, the app reschedules them for all owned plants using NotificationScheduler.scheduleWatering().

In the onResume() method, both switches are re-evaluated when the fragment becomes visible again. This ensures that their states reflect the actual values stored in SharedPreferences, providing a consistent and responsive user experience even after navigating away from or returning to the fragment.

## MenuManager

The MenuManager class provides a reusable utility method to set up a navigation menu across different fragments or activities within the application. It supports switching between core sections of the app like Home, Search, Favorite Plants, Own Plants, Scanner, Settings, and Notifications. It also handles user profile image display and sign-out functionality using Firebase and Google Sign-In.

The setupMenu method takes in several parameters including the activity, a list of menu items, and UI elements (RecyclerView, menu icon, and profile image). It initializes a MenuAdapter with click listeners that navigate to the corresponding fragment or activity based on the clicked menu item. For example, selecting "Search" replaces the current fragment with SearchFragment.

The method uses the FragmentManager to replace the currently displayed fragment with the selected one. Each fragment transaction is added to the back stack so users can navigate back. The menu is hidden after any selection to ensure a clean UI transition.

The hamMenuIcon controls the visibility of the navigation drawer (menuRecyclerView). When clicked, it toggles between showing and hiding the menu using View.GONE and View.VISIBLE.

If the user is signed in with a Google account and has a profile photo, the Glide library loads and displays it into the profileImageView using a circular crop for better aesthetics.

Clicking the profile image shows a popup menu (profile\_menu) with options like "Sign Out." If the user selects "Sign Out", the signOut() method is triggered.

The signOut() method signs the user out of Firebase and Google using FirebaseAuth.getInstance().signOut() and GoogleSignInClient.signOut(). After a successful sign-out, it launches the SplashActivity and clears the current task stack to prevent the user from navigating back.

# Database

## NotificationDao

NotificationDao defines a **Data Access Object (DAO)** interface for interacting with a **Room database** in an Android application. Specifically, it handles operations related to **notifications** for a plant care application.

**The NotificationDao interface includes the following methods:**

1. **insertNotification(Notification notification)**: Inserts a new notification into the database.
2. **getAllNotifications()**: Retrieves all notifications from the database, ordered by the time of the notification (from the most recent).
3. **getNotificationsForPlant(int plantId)**: Fetches all notifications related to a specific plant, identified by its plantId.
4. **getNotificationById(long notificationId)**: Retrieves a specific notification by its unique identifier (notificationId).
5. **updateNotification(Notification notification)**: Updates an existing notification in the database.
6. **deleteNotification(long notificationId)**: Deletes a specific notification based on its notificationId.
7. **deleteNotificationsForPlant(int plantId)**: Deletes all notifications associated with a particular plant, identified by its plantId.

## OwnPlantDao

OwnPlantDao defines a **Data Access Object (DAO)** interface for managing **own plant data** in the Room database of an Android application. The OwnPlantDao interface includes several methods for performing various database operations related to the plants that a user owns.

The interface contains the following methods:

1. **insertOwnPlant(OwnPlant ownPlant)**: Inserts a new plant into the database. The method returns a long value, which is the ID of the newly inserted plant.
2. **getAllOwnPlants()**: Retrieves all plants stored in the database.
3. **getOwnPlant(int selectedPlantId)**: Fetches a specific plant from the database based on its unique selectedPlantId.
4. **deleteAllOwnPlants()**: Deletes all plants from the database.
5. **deletePlant(int id)**: Deletes a specific plant from the database based on its unique id.
6. **getAllOwnPlantIds()**: Retrieves a list of all plant IDs from the database.

## PlantDao

The PlantDao interface, located in the hu.nje.plantcare.database package, defines the data access operations related to the Plant entity in the Room database. Annotated with @Dao, this interface provides abstract methods that Room uses to generate the necessary SQLite code at compile time.

It includes the following operations:

1. insert(Plant plant): Adds a new plant record to the plants table. This method is annotated with @Insert, enabling Room to handle the SQL insert statement automatically.
2. getAllPlants(): Returns a list of all Plant entries in the database by executing a SELECT \* FROM plants query.
3. deleteAll(): Deletes all records from the plants table using a bulk DELETE query.
4. deleteOne(int plantId): Deletes a specific plant from the table based on the plantId provided.
5. getPlant(int plantId): Retrieves a single Plant object whose plantId matches the given parameter.

This DAO serves as a clean and modular interface between the application logic and the SQLite database, allowing for easy testing and maintainability.

## PlantDatabase

The PlantDatabase class is the central component of the app’s local data storage, implemented using Android’s Room Persistence Library. Located in the hu.nje.plantcare.database package, it defines an abstract database that includes three entities: Plant, OwnPlant, and Notification. These correspond to the app’s core data models, representing general plant information, user-owned plants, and plant care reminders, respectively. The database is annotated with @Database, where the current schema version is set to 6. For development purposes, it uses fallbackToDestructiveMigration(), allowing Room to drop and recreate the database if a migration path is not provided — a convenient but potentially data-destructive approach.

The PlantDatabase class follows the Singleton pattern to ensure that only one database instance exists during the app’s runtime. This is implemented via a volatile static INSTANCE variable and a thread-safe, double-checked locking mechanism inside the getDatabase(Context context) method. This method initializes the Room database using Room.databaseBuilder() with the application context and assigns it the name "plant\_database". The database is built lazily and cached for future access. The class also defines three abstract DAO access methods — plantDao(), ownPlantDao(), and notificationDao() — which are used by the rest of the app to perform queries and data operations.

## Watering

Watering defines an **enum** called Watering, which is used to represent different types of watering frequencies for plants. The enum helps categorize the watering schedule into distinct groups, with each group having a corresponding name, minimum days, and maximum days between watering.

**The Watering enum has the following constants:**

1. **FREQUENT**: Watering frequency that requires watering every 1 to 3 days.
2. **AVERAGE**: Watering frequency that requires watering every 4 to 6 days.
3. **MINIMUM**: Watering frequency that requires watering every 7 to 14 days.
4. **NONE**: A watering frequency where the plant does not need watering for 15 to 25 days.

**Each enum constant has three associated properties:**

* **apiValue**: A string value used for API communication (e.g., "frequent", "average").
* **minDays**: The minimum number of days between waterings.
* **maxDays**: The maximum number of days between waterings.

The constructor Watering(String apiValue, int minDays, int maxDays) initializes these properties for each enum constant.

**Methods:**

1. **getApiValue()**: Returns the apiValue string (e.g., "frequent", "average") for the watering frequency.
2. **getMinDays()**: Returns the minimum number of days between waterings for the given frequency.
3. **getMaxDays()**: Returns the maximum number of days between waterings for the given frequency.
4. **getWateringInDays(String apiValue)**: A static method that takes an apiValue string and returns the corresponding Watering enum constant. If the provided apiValue doesn't match any of the defined enum values, it throws an IllegalArgumentException with a message indicating an unknown watering value.

## Entity

### BasicPlant

The BasicPlant class represents a plant entity in the application, encapsulating essential plant information, such as its **ID**, **common name**, **scientific name**, and **image URL**. This class is designed to store basic data about a plant that can be used throughout the application.

**It has four main attributes:**

1. **id**: A unique identifier for each plant, which is set during the object's creation and cannot be modified afterward.
2. **common\_name**: The commonly recognized name of the plant (e.g., "Rose," "Cactus").
3. **scientific\_name**: The scientific or Latin name of the plant (e.g., "Rosa," "Cactaceae").
4. **imgUrl**: A URL linking to an image of the plant, which is used for displaying a visual representation in the application.

The class provides a constructor to initialize these attributes. The constructor accepts the id, common\_name, scientific\_name, and imgUrl as parameters, ensuring that each plant object is initialized with the correct data.

Additionally, the class includes getter and setter methods for each attribute. The **getId()** method allows access to the plant's unique identifier, while the **getCommon\_name()**, **getScientific\_name()**, and **getImgUrl()** methods provide access to the corresponding plant details. The setter methods **setCommon\_name()**, **setScientific\_name()**, and **setImgUrl()** allow modifications to the plant's common name, scientific name, and image URL, respectively.

### Notification

The Notification class represents a notification entity in the application, specifically designed for notifying users about their plants' care routines. This class is marked with the @Entity annotation, which indicates that it is a Room database entity, and it is associated with the "notifications" table in the database.

**The class has several fields:**

1. **id**: A unique identifier for each notification. This field is marked with @PrimaryKey(autoGenerate = true), meaning the ID will be automatically generated by the database when a new notification is inserted.
2. **plantId**: The identifier of the plant associated with the notification. It links the notification to a specific plant in the database.
3. **plantName**: The common name of the plant associated with the notification (e.g., "Rose").
4. **notificationTime**: A long value representing the timestamp of when the notification is set to trigger, stored as the number of milliseconds since the Unix epoch.
5. **isDelivered**: A boolean flag indicating whether the notification has been delivered to the user.
6. **plantImageUrl**: A new field that stores the URL of the plant’s image, providing a visual reference along with the notification.

The class includes a no-argument constructor (public Notification()) for Room to use when creating objects of this type.

Additionally, the class provides getter and setter methods for each field:

* **getId()** and **setId()** allow access to and modification of the notification's unique identifier.
* **getPlantId()** and **setPlantId()** provide access to the plant ID related to the notification.
* **getPlantName()** and **setPlantName()** let you retrieve and set the plant’s common name.
* **getNotificationTime()** and **setNotificationTime()** allow you to get and set the time when the notification is scheduled.
* **isDelivered()** and **setDelivered()** are used to check and update the delivery status of the notification.
* **getPlantImageUrl()** and **setPlantImageUrl()** handle the URL of the plant's image.

### OwnPlant

The OwnPlant class represents a plant that is owned by the user in the application. This class is marked with the @Entity annotation, indicating that it is a Room database entity, and it is associated with the "own\_plants" table in the database.

**Fields:**

1. **id**: A unique identifier for each plant. This field is marked with @PrimaryKey(autoGenerate = true), meaning that the database will automatically generate a unique ID for each new plant entry.
2. **commonName**: The common name of the plant (e.g., "Rose," "Cactus").
3. **type**: The type or category of the plant (e.g., "Flower," "Succulent").
4. **watering**: A string that represents the watering frequency of the plant (e.g., "frequent," "average").
5. **imgUrl**: A URL pointing to an image of the plant, allowing for the visual representation of the plant in the app.
6. **description**: A textual description of the plant, providing additional details or care instructions for the user.

**Constructor:**

The class provides a no-argument constructor (public OwnPlant()) for Room to use when creating objects of this type.

**Getter and Setter Methods:**

* **getId()** and **setId()**: Accessors for the plant's unique identifier.
* **getCommonName()** and **setCommonName()**: Accessors for the plant's common name.
* **getType()** and **setType()**: Accessors for the plant's type or category.
* **getWatering()** and **setWatering()**: Accessors for the plant's watering frequency.
* **getImgUrl()** and **setImgUrl()**: Accessors for the URL of the plant's image.
* **getDescription()** and **setDescription()**: Accessors for the plant's description.

### Plant

The Plant class represents a plant entity in the application and is designed to store detailed information about a specific plant. It is marked with the @Entity annotation, indicating that it is a Room database entity, and is associated with the "plants" table in the database.

**Fields:**

1. **id**: A unique identifier for each plant, marked with @PrimaryKey(autoGenerate = true), meaning it will be automatically generated by the database when a new plant is inserted.
2. **plantId**: A specific ID for the plant, typically used for referencing the plant across different parts of the application.
3. **commonName**: The common name of the plant (e.g., "Rose," "Cactus").
4. **scientificName**: The scientific or Latin name of the plant (e.g., "Rosa," "Cactaceae").
5. **type**: The type or category of the plant (e.g., "Flower," "Succulent").
6. **cycle**: The plant's growth cycle or flowering cycle (e.g., "Annual," "Perennial").
7. **watering**: The watering frequency for the plant, typically a string value (e.g., "frequent," "average").
8. **imgUrl**: A URL pointing to an image of the plant, enabling the visual display of the plant in the app.
9. **description**: A textual description of the plant, offering additional details or care instructions.
10. **isFavorite**: A boolean flag that indicates whether the plant is marked as a favorite by the user.

**Constructor:**

The class provides a constructor to initialize the plant object with specific details. The constructor takes the following parameters:

plantId,commonName,scientificName,type,cycle,watering,imgUrl,description,isFavorite

**Getter and Setter Methods:**

* **getId()** and **setId()**: Accessor methods for the plant’s unique identifier.
* **getPlantId()** and **setPlantId()**: Accessor methods for the plant’s specific ID.
* **getCommonName()** and **setCommonName()**: Accessor methods for the plant’s common name.
* **getScientificName()** and **setScientificName()**: Accessor methods for the plant’s scientific name.
* **getType()** and **setType()**: Accessor methods for the plant’s type.
* **getCycle()** and **setCycle()**: Accessor methods for the plant’s growth or flowering cycle.
* **getWatering()** and **setWatering()**: Accessor methods for the plant’s watering schedule.
* **getImgUrl()** and **setImgUrl()**: Accessor methods for the URL of the plant’s image.
* **getDescription()** and **setDescription()**: Accessor methods for the plant’s description.
* **isFavorite()** and **setFavorite()**: Accessor methods for the plant’s favorite status.

# Adapter

## BasicPlantAdapter

The BasicPlantAdapter is a custom RecyclerView.Adapter that displays a list of basic plant information in a scrollable list. It binds plant data to a layout (plant\_item.xml) and allows the user to interact with individual plant items. Each item shows the common name, scientific name, and an image of the plant. The adapter takes in a list of BasicPlant objects and an OnPlantClickListener as parameters. The plants list holds the data to be displayed, while the listener handles user interactions .

* **setPlants Method**  
  This method allows updating the adapter’s data with a new list of plants and automatically refreshes the displayed items using notifyDataSetChanged().
* **onCreateViewHolder Method**  
  This method inflates the layout for each plant item in the list. It uses the plant\_item.xml layout file and returns a PlantViewHolder that holds references to the views inside the item layout.
* **onBindViewHolder Method**  
  This method binds each plant's data to the corresponding view in the RecyclerView. It sets the plant’s common and scientific names in the respective TextViews and uses the Glide library to load the plant’s image into the ImageView. Additionally, it sets a click listener on each item, triggering the onPlantClick callback with the plant’s ID when clicked.
* **getItemCount Method**  
  Returns the total number of items in the list. If the plant list is null, it returns 0 to avoid crashes.
* **PlantViewHolder Inner Class**  
  This static inner class holds references to the views for each item in the list (common name, scientific name, and image). It helps in recycling views efficiently for smooth scrolling.
* **OnPlantClickListener Interface**  
  This interface defines a single method onPlantClick(int plantId) that external classes (like fragments or activities) must implement if they want to handle click events on individual plant items.

## NewPlantAdapter

The NewPlantAdapter class is a custom adapter used in the app to allow users to input and submit a new plant. Although it uses a RecyclerView, it only renders one item: a form with editable fields. This setup maintains consistency with other parts of the UI that also use RecyclerView-based adapters.

This adapter takes an OnImageActionListener interface as a parameter, allowing the host activity or fragment to handle image selection externally. It holds references to the selected image path and the currently displayed ViewHolder

**The nested ViewHolder class references all the form elements:**

* EditText fields for the plant’s common name, type, and description.
* A Spinner for selecting watering frequency (from preset options).
* A Button to trigger image selection.
* An ImageView to show the selected image preview.

These views are linked to their corresponding UI components in the form\_field\_item.xml layout.

In the onCreateViewHolder() method, the layout is inflated, and a new ViewHolder instance is created using form\_field\_item.xml. This is the only layout used by the adapter.

**The onBindViewHolder() method initializes the form:**

* It clears all text fields.
* Sets up the watering spinner with four watering options: Frequent, Average, Minimum, and None.
* Loads the selected image into the preview if available, or loads a default image if not.
* It also connects the "Set Image" button to the external listener's onSetImageClick() method.

**Two methods support form handling:**

* isFormValid() checks if all input fields are non-empty, ensuring the form is ready for submission.
* setPlantData() creates a new OwnPlant object using the entered form data, including the appropriate watering type string (e.g., "frequent", "average"), and the selected image path.

## NotificatonAdapter

The NotificationAdapter is a custom RecyclerView.Adapter used to display a list of plant-related notifications in the app. Each notification item shows relevant details such as the plant name, scheduled or sent time, notification status, and optionally an image of the plant. The data is provided via a list of Notification objects.

The adapter is initialized with a list of Notification objects. There’s also a setNotifications() method that allows the list to be updated dynamically, ensuring the view reflects any changes using notifyDataSetChanged().

**The inner NotificationViewHolder class defines the UI components for each list item:**

* ImageView notificationPlantImage: displays the plant's image.
* TextView notificationPlantName: shows the name of the plant.
* TextView notificationTime: displays the notification’s timestamp.
* TextView notificationStatus: indicates whether the notification has been sent or is still scheduled.

These views are initialized from the messages\_item.xml layout.

In onCreateViewHolder(), the messages\_item.xml layout is inflated and returned as a NotificationViewHolder.

**In onBindViewHolder(), the adapter binds the data from a Notification object to the appropriate views:**

* The plant name and formatted timestamp are set.
* The status text is updated to either “Status: Sent” or “Status: Scheduled”, based on the isDelivered() flag.
* If an image URL is present and not empty, it is loaded using Glide with a placeholder and error fallback image (plant\_placeholder).
* If no image URL is available, a default image (plantfavpic) is shown.

A SimpleDateFormat is used to format the notification time in the yyyy.MM.dd HH:mm format according to the device’s locale.

The getItemCount() method ensures safe handling of a null notification list and returns 0 in that case; otherwise, it returns the actual size of the list.

## OwnPlantAdapter

The OwnPlantAdapter is a custom RecyclerView.Adapter designed to display a list of user-added (own) plants within the application. Each item includes a plant name, its image, and a delete button. The adapter handles both click events for viewing a plant’s details and deleting a plant entry.

**The adapter takes:**

* A List<OwnPlant> that represents the user’s personal plants.
* An OnDeleteClickListener for handling delete button actions.
* An OnOwnPlantClickListener for handling general item clicks (like opening details).

There's also a setOwnPlants() method to update the dataset dynamically and refresh the view via notifyDataSetChanged().

**The OwnPlantViewHolder inner class defines and references three main UI components from the own\_plant\_item.xml layout:**

* TextView commonName: Displays the plant’s name.
* ImageView plantImage: Shows the image of the plant (from a URL).
* ImageButton deleteButton: Triggers deletion of the plant when pressed.

These are bound in the constructor using findViewById().

onCreateViewHolder() inflates the own\_plant\_item.xml layout and returns a new OwnPlantViewHolder.

**onBindViewHolder() binds plant data to the view. Specifically:**

* The plant's name is set in the TextView.
* The image is loaded with Glide using the image URL from OwnPlant. A default image (plantfavpic) is shown on error.
* The delete button calls the provided OnDeleteClickListener with the plant ID.
* Tapping the item triggers the OnOwnPlantClickListener, allowing the app to handle navigation or other actions.

**Two interfaces are defined to decouple click handling from the adapter logic:**

* OnDeleteClickListener: Called when the delete button is pressed.
* OnOwnPlantClickListener: Called when the entire item view is clicked.

The adapter safely returns 0 if the plant list is null, otherwise it returns the list's size via getItemCount().

## OwnPlantDetailsAdapter

OwnPlantDetailsAdapter is a custom RecyclerView.Adapter used to display detailed information about a single user-added (OwnPlant) plant. It presents the plant's image, name, type, watering frequency, and description. Additionally, it includes a "Back" button to return to the plant list view.

Accepts a single OwnPlant object to display. Accepts an OnBackClickListener interface implementation to handle "Back" button presses.The setOwnPlant() method allows updating the displayed plant and refreshing the view.

**The inner class OwnPlantDetailsViewHolder maps to own\_plant\_item\_details.xml and contains:**

* ImageView plantImage: Displays the plant's photo.
* TextView commonNameTextView: The plant’s common name.
* TextView typeTextView: The plant type or category.
* TextView wateringTextView: Human-readable watering frequency.
* TextView descriptionTextView: A short plant description.
* Button back2OwnList: Returns to the plant list when clicked.

**onBindViewHolder() populates all fields from the OwnPlant instance:**

Loads the plant image using **Glide**, with placeholders for loading or error.Formats the watering frequency using the Watering.getWateringInDays() utility and displays it as a range (e.g., "Every 4–6 days").Sets the rest of the data from OwnPlant fields.The back button uses the provided listener to trigger navigation.

OnBackClickListener: Defines OnBackClick() to respond when the user presses the back button.

## PlantDetailsAdapter

PlantDetailAdapter is a RecyclerView.Adapter designed to display detailed information about a single plant. It provides an interface for managing favorite status (via a Switch) and navigating back to the plant list view. It includes the plant’s image, common and scientific names, type, lifecycle, watering frequency, and description.

**The inner class PlantViewHolder maps to plant\_item\_details.xml and contains:**

* Button back2list: Button to return to the plant list view.
* ImageView plantImage: Displays the plant's image.
* TextView commonNameTextView: Displays the plant’s common name.
* TextView scientificNameTextView: Displays the plant’s scientific name.
* TextView typeTextView: Displays the plant’s type (e.g., flower, herb).
* TextView cycleTextView: Displays the plant's lifecycle (e.g., annual, perennial).
* TextView wateringTextView: Displays the plant’s watering frequency.
* Switch favoriteSwitch: Toggle to mark the plant as a favorite.
* TextView descriptionTextView: Displays a description of the plant.

**onBindViewHolder() populates all fields with the plant data:**

* Loads the plant’s image using **Glide**, with optional placeholders for loading or errors.
* Sets the plant's common name, scientific name, type, lifecycle, and description.
* Calculates and formats the watering frequency into a range (e.g., "Every 3–5 days") using Watering.getWateringInDays().
* Sets the favoriteSwitch based on the plant’s isFavorite status and handles the click event to toggle the favorite status. When the switch is clicked, the plant's favorite status is updated, and the item is refreshed.

The back2list button triggers the OnBackClickListener to navigate back to the plant list when clicked.

OnFavoriteClickListener: Defines OnFavoriteClick() to respond when the user toggles the favorite switch.

OnBackClickListener: Defines OnBackClick() to respond when the user presses the back button.

## MenuAdapter

MenuAdapter is a RecyclerView.Adapter used for displaying a list of menu items. Each item in the menu is a string that is rendered as a TextView, and clicking on any item triggers a callback to a listener, enabling interaction with the menu.

**The adapter takes three parameters in its constructor:**

* Context context: Context for inflating layout resources.
* List<String> menuItems: List of menu items to be displayed.
* OnMenuItemClickListener listener: Listener interface to handle item clicks.

The getItemCount() method returns the number of menu items.

**The inner class MenuViewHolder maps to the layout file menu\_item.xml, and contains:**

* TextView menuItemText: Displays the text of each menu item.

onBindViewHolder() binds the menu item text to the TextView in each item view.

Sets an OnClickListener for each item in the list. When a menu item is clicked, the onMenuItemClick() method of the listener is called with the item text as an argument. Returns the size of the menuItems list, determining how many menu items to display.

OnMenuItemClickListener: Defines the method onMenuItemClick(String item) which is called when an item is clicked.

# API

## ApiService

The ApiService class in the Android-based PlantCare application is the component responsible for handling API calls. It retrieves data using the Perenual plant database API and relies on the Volley library to send HTTP requests. This class provides two main functions: retrieving detailed information about a specific plant (ApiRequest) and performing a keyword-based search (SearchApiRequest).

The ApiRequest method fetches detailed information about a plant using its unique identifier (id) and an API key. After constructing the URL, the method sends a GET request to the server and parses the JSON response to extract the plant’s main attributes: its ID, common name (common\_name), scientific name (scientific\_name), type, life cycle (cycle), watering needs (watering), a description, and a small-sized image. These values are used to create a Plant object, which is returned through a callback function, making the data available to be displayed on the user interface. The method includes error handling to catch any JSON parsing issues or network-related problems.

The SearchApiRequest method performs a plant search based on a keyword provided by the user. It constructs the request URL using the keyword, API key, and base URL. The method processes each entry in the "data" array from the returned JSON by extracting the ID, common name, first scientific name, and a small image. It filters out entries with unavailable images or those requiring paid API access. The valid results are converted into BasicPlant objects and passed back using a callback. This method also includes error handling to notify in case of issues during JSON processing.

In summary, the ApiService class enables dynamic data retrieval from the online plant database, supporting both plant search and detailed data display within the user interface. Its required parameters (API key, ID, keyword, etc.) allow it to adapt flexibly to different use cases within the application.

## DetailsResultCallBack

The DetailResultCallBack interface in the hu.nje.plantcare.api package is a simple callback mechanism used to return detailed plant data retrieved via an API call.

## SearchResultCallBack

The SearchResultCallBack interface in the hu.nje.plantcare.api package serves as a callback to handle the result of a plant search operation.

# Util

## NotificationScheduler

The NotificationScheduler class is a utility class responsible for managing scheduled plant watering notifications. It leverages Android’s AlarmManager to notify users when it is time to water their plants, based on each plant’s specified watering frequency. This contributes significantly to the app’s goal of promoting consistent plant care through timely reminders.

**Key Functionalities:**

* **Notification Scheduling**  
  The scheduleWatering() method sets up a repeating alarm for each plant, using its watering frequency to calculate the interval. If notifications are globally disabled in the app’s shared preferences, the method exits early without scheduling. Otherwise, it calculates the delay until the first notification using calculateTriggerTime() and sets an inexact repeating alarm through AlarmManager.

When scheduling:

* + A PendingIntent is created with the plant’s ID, name, and other data.
  + A new Notification entity is saved into the local Room database (PlantDatabase) for future reference, including the time and image URL.
  + The log output confirms scheduling details or warns if an invalid frequency is passed.
* **Notification Cancellation**  
  The cancelWatering() method cancels a previously scheduled notification for a specific plant by its unique ID. It recreates the corresponding PendingIntent and cancels it via the same AlarmManager. Additionally, it removes the associated Notification entry from the local database on a background thread.
* **Frequency Parsing**  
  The calculateTriggerTime() helper method interprets textual frequency descriptors:
  + "frequent" → every 2 days
  + "average" → every 5 days
  + "minimum" → every 10 days
  + "none" → disables scheduling

Any unknown values will return -1, indicating an invalid configuration.

**Preferences:**

* PREF\_NOTIFICATIONS\_ENABLED is a global user preference that determines whether notifications should be used at all.
* If disabled, no alarms will be scheduled, regardless of individual plant settings.

**Design Notes:**

This class provides modular and centralized control over watering notifications, helping maintain consistent logic across the app. It interacts closely with:

* AlarmManager and PendingIntent for time-based scheduling.
* PlantDatabase and the NotificationDao for persistence.
* PlantWateringReceiver, which is expected to handle the broadcast and trigger user-visible alerts.

## PlantWateringReceiver

The PlantWateringReceiver class is a broadcast receiver that listens for scheduled alarms triggered by the NotificationScheduler. Its purpose is to display a notification to the user when it's time to water a specific plant. It also updates the app's local database to mark the notification as delivered.

**Main Functionalities:**

* **Receiving the Alarm**  
  The receiver listens for the ACTION\_WATERING\_REMINDER broadcast. When it receives this intent, it extracts the plant ID and name from the intent’s extras. If the global notifications setting is disabled (notifications\_enabled), it logs the event and aborts the process.
* **Displaying the Notification**  
  The showNotification() method builds and displays a system notification using NotificationCompat. It shows a small icon, the name of the plant, and a message reminding the user to water it. The notification is linked to MainActivity, so tapping it will launch the main screen of the app. The notification is also automatically dismissed when tapped (setAutoCancel(true)).
* **Creating a Notification Channel (API 26+)**  
  The method createNotificationChannel() ensures that a channel for plant watering reminders exists, which is required for displaying notifications on Android Oreo (API 26) and above. It defines a name, description, and default importance level.
* **Updating the Database**  
  Once the notification is shown, the receiver updates the local Room database entry for that plant's notification. It sets the delivered flag to true, indicating that the reminder was successfully sent. This update is executed in a background thread to avoid blocking the UI.

**Design Considerations:**

* **User Preferences**  
  The receiver respects the global user preference for notifications. If disabled, it logs the action and exits early.
* **Notification Uniqueness**  
  Each notification ID is offset by the plant ID, ensuring that multiple plants can have distinct, non-conflicting notifications.
* **Efficient and Lightweight**  
  This receiver does not reschedule alarms or store complex logic; instead, it acts as a bridge between the system alarm and the UI, ensuring modularity.

# Activity

## MainActivity

The MainActivity class serves as the central entry point of the PlantCare application after the splash screen. It is responsible for initializing the main layout, setting up the custom menu interface, handling theme preferences (light or dark mode), and integrating user profile display.

When the activity is created, it first checks the user’s preferred theme setting stored in shared preferences. If **dark mode** is enabled (PREF\_DARK\_MODE), the application applies the night mode theme using AppCompatDelegate.

The layout associated with this activity is activity\_main.xml, which contains the core UI components, including a **hamburger menu icon**, a **profile image**, and a **RecyclerView** used to display the custom sidebar menu.

The menu is initialized with the following options:

* **Home**
* **Search**
* **Favourite plants**
* **Own plants**
* **Plant scanner**
* **Settings**
* **Notifications**

These menu items are managed and displayed using a custom MenuAdapter and the MenuManager utility class. MenuManager.setupMenu(...) links the menu layout and handles navigation between fragments accordingly.

Additionally, the UI elements declared in MainActivity, such as TextView data, TextView infoText, and ImageView profileImageView, are set up to personalize the user interface and enhance user interaction, for example, by showing user details or profile images (potentially loaded using Glide).

Although some logic is handled externally (e.g., MenuManager manages fragment navigation), MainActivity plays a foundational role in connecting these components and providing a structured, responsive experience from the main navigation screen of the application.

## SplashActivity

The SplashActivity class is the initial activity that is displayed when the PlantCare application is launched. It acts as a transitional screen, presenting a splash screen with a short animation while performing a quick check on the user’s authentication status.

Upon creation, the activity inflates the activity\_splash.xml layout, which typically includes branding elements such as a logo or app icon. It then triggers a **fade-in animation** using the AnimationUtils class, applied to an ImageView (imageView3). This visual effect enhances the onboarding experience and provides a polished introduction to the app.

The activity introduces a **3-second delay** using a Handler, during which the app determines whether the user is already authenticated via Google Sign-In. This is done by checking the GoogleSignInAccount:

* If a user is **already signed in**, the activity redirects the user to the MainActivity, which serves as the home screen of the app.
* If no user is authenticated, the activity dynamically replaces its content with the GoogleSignInFragment using a FragmentTransaction. This fragment provides the interface and logic for user authentication through Google.

By handling navigation logic within the splash screen, SplashActivity helps ensure that users are smoothly directed to the appropriate next screen based on their sign-in status. It combines aesthetics and essential functionality to provide a seamless and responsive app launch experience.

# Manifest

The AndroidManifest.xml file defines essential details about the Android application, such as permissions, components, and configuration settings required by the app.

## Key Elements

### Package Declaration

* + The package attribute defines the unique identifier for the app (hu.nje.plantcare), which is used to distinguish the app from other apps on the device.

### Permissions

* + The app requests two permissions:
    - **INTERNET**: The <uses-permission android:name="android.permission.INTERNET"/> tag allows the app to access the internet, which is likely necessary for features such as retrieving data from a remote server.
    - **CAMERA**: The <uses-permission android:name="android.permission.CAMERA"/> permission allows the app to use the device's camera.
  + Additionally, the app declares the use of a hardware feature:
    - **android.hardware.camera.any**: The <uses-feature android:name="android.hardware.camera.any"/> tag indicates that the app uses the camera and will function on devices that have a camera.

### Application Tag

* + The <application> tag contains global settings for the app, such as:
    - **Backup Options**:
      * android:allowBackup="true": Allows the app's data to be backed up.
      * android:dataExtractionRules="@xml/data\_extraction\_rules": Refers to XML configuration for backup rules.
      * android:fullBackupContent="@xml/backup\_rules": Specifies which files should be included or excluded from the full backup.
    - **App Icon and Label**:
      * android:icon="@mipmap/ic\_launcher": Specifies the app icon.
      * android:label="@string/app\_name": The app's name.
    - **Theme and Orientation**:
      * android:theme="@style/Theme.PlantCare": Specifies the theme used for the app.
      * android:screenOrientation="portrait": Locks the screen orientation to portrait mode.

### Activities

* + Two activities are declared within the app:
    - **SplashActivity**:
      * android:name=".SplashActivity": This is the entry activity of the app, typically shown when the app is launched.
      * The intent filter inside specifies that it is the main entry point for the app (android.intent.action.MAIN) and should appear in the launcher (android.intent.category.LAUNCHER).
    - **MainActivity**:
      * android:name=".MainActivity": This is the main screen of the app after the splash screen.
      * Both activities are set to use a portrait screen orientation with android:screenOrientation="portrait".

### Target API

* + The tools:targetApi="31" tag is used for specifying the app's target API level (API level 31, corresponding to Android 12). This helps the system handle compatibility with newer Android versions.