# Flutter Product Feed Project Documentation

## Overview

This Flutter project demonstrates a structured and maintainable approach to implementing a product feed application.   
The application fetches data from a remote API and displays products with functionalities like sorting and filtering based on categories.

## Project Structure

The project is organized into the following directories and files:  
- \*\*/lib/screens/home\_screen.dart\*\*: Contains the main UI logic for the home screen.  
- \*\*/lib/services/api\_services.dart\*\*: Handles API interactions for fetching data from the remote server.  
- \*\*/lib/widgets/category\_item.dart\*\*: A reusable widget for rendering category items.  
- \*\*/lib/widgets/product\_card.dart\*\*: A reusable widget for rendering product details in a card layout.  
- \*\*/lib/widgets/glass\_loading\_effect.dart\*\*: A reusable widget for displaying a glass effect loading animation.  
- \*\*main.dart\*\*: Entry point of the application.

## Features

1. Fetch and display categories dynamically from the API.  
2. Fetch and display products with sorting and filtering capabilities.  
3. Global reusable widgets for consistent UI/UX.  
4. Proper separation of concerns with distinct files for services, widgets, and screens.

## Implementation Details

1. \*\*API Integration\*\*:  
 - The `ApiService` class is responsible for interacting with the API endpoints.  
 - Fetches categories and products dynamically.  
   
2. \*\*Category Selection\*\*:  
 - Categories are fetched and displayed in a horizontal scrollable row.  
 - Selecting a category fetches and displays products related to that category.  
   
3. \*\*Product Feed\*\*:  
 - Products are displayed in a grid layout.  
 - Users can sort products by price (Lower Price First, Higher Price First).  
 - Each product card includes an image, title, description, and price.  
  
4. \*\*Global Loading Effect\*\*:  
 - The `GlassLoadingEffect` widget is used globally to show a consistent loading animation during API calls.

## How to Extend

1. \*\*Add New Features\*\*:  
 - Create new services in the `services` folder for additional API endpoints.  
 - Add new widgets for UI components in the `widgets` folder.  
   
2. \*\*Change API\*\*:  
 - Update the `ApiService` class to interact with a new API or data source.  
   
3. \*\*Add Additional Screens\*\*:  
 - Create a new Dart file in the `screens` folder.  
 - Use the existing widgets for a consistent design language.

## Conclusion

This project demonstrates a clean, modular, and maintainable approach to building a Flutter application.   
With reusable components, organized code structure, and clear documentation, the project is scalable for future enhancements.