1. Introduction

Welcome to the developer documentation for Touri application, a robust platform dedicated to facilitating the purchase of cleaning products. This application serves as a comprehensive marketplace designed to showcase an extensive range of cleaning items, highlight various brands, and efficiently display delivery addresses for seamless transactions.
Purpose and Focus
At its core, Touri aims to streamline the purchase process for cleaning products. It focuses on three primary components:

- Product Catalog: The platform hosts an extensive catalog featuring a diverse array of cleaning products, ensuring users have access to a wide selection suitable for various needs.

- Brand Showcase: Touri showcases multiple brands known for their quality cleaning supplies.

- Address Management: With a user-friendly interface, the application provides a convenient means for users to manage and select delivery addresses, enhancing the overall shopping experience.

Developer Documentation Focus
This documentation is crafted to assist developers in understanding the intricacies of Touri. It primarily focuses on elucidating how to run the application, interact with its main features (products, brands, addresses), and effectively utilize the provided functionalities within the development environment.

2. Prerequisites

Before you proceed with running Touri, ensure you have the following prerequisites installed on your development environment:

Development Tools:

- Integrated Development Environment (IDE):
  Android Studio or Visual Studio Code with Flutter and Dart plugins/extensions installed.

Configurable Flutter and Dart:

- Flutter SDK:Download and install the stable version 2 of Flutter by following the official Flutter installation guide. (attention the current code doesn't support flutter 3)
- Dart SDK:Dart comes bundled with Flutter. Ensure your Flutter installation includes the Dart SDK.

Internet Connection: A stable internet connection is required for installing dependencies and fetching project resources.
Installation Guides:
Follow the respective IDE's documentation to install Flutter and Dart plugins/extensions if they are not already installed.

3. Installation Guide

1. Download and Unzip Application Files:

Download the Application:
Obtain the ZIP file containing Touri.
Unzip the Application:
Extract the contents of the ZIP file to a preferred directory on your system.

2. Open in Integrated Development Environment (IDE):

- **Using Android Studio:**
  Open Android Studio.
  Choose "Open an existing Android Studio project."
  Navigate to the directory where you extracted the application files and select the main project folder. Click "OK" to open the project.

- **Using Visual Studio Code:**
  Open Visual Studio Code.
  Choose "File" > "Open Folder" and navigate to the directory where you extracted the application files. Select the main project folder to open it in VS Code.

3. Configure Flutter and Dart:

- **Install Dependencies:**
- Ensure a stable internet connection and, if required, a VPN to access specific resources.
- Open a terminal or command prompt within the IDE or navigate to the project folder using the terminal/command prompt.
- **Run Command:**
  Execute the command flutter pub get in the terminal/command prompt.
  `flutter pub get`
  This fetches and installs the necessary dependencies and packages for the application.

4. Run the Application:

Use the IDE's run/debug options (usually found in the top toolbar) to launch the application on an emulator or a connected device.

Additional Considerations:

Internet Connection: Ensure a stable internet connection throughout the installation process for dependency downloads.
VPN: If specific resources require a VPN, ensure it is active before running flutter pub get.
Troubleshooting: If encountering issues, refer to the official Flutter documentation or community forums for assistance.

4. Running the Application

Upon successful setup and completion of library downloads, the initial run of Touri may require a few minutes to configure and prepare the environment.

First Run Experience:

- Select Language Screen:
  Upon launching the application for the first time, users will be greeted with a "Select Language" screen.|
  This screen provides options for users to choose their preferred language for app navigation and content.

- Onboarding Screen:
  Following the language selection, users will be guided through an "Onboarding" process.
  The onboarding screens introduce the application's key features, functionalities, or benefits using engaging visuals and brief descriptions.

- Home Screen:
  After completing the onboarding sequence, users will be directed to the "Home Screen."
  The Home Screen serves as the central hub of the application, featuring essential sections or categories of cleaning products, brand highlights, and navigation elements.

Application Navigation:

- Language Selection: Users can select between available language options and select their preferred language for the application.

- Onboarding Sequence: Users can swipe through the onboarding screens to grasp the application's core features.

- Home Screen Navigation: The Home Screen provides intuitive navigation options, enabling users to explore various sections or categories seamlessly.

Note:
First Run Setup: The initial setup may take a few minutes to configure and download essential resources. Subsequent launches will be faster as the required resources will already be cached.

5. Main Functions or Features

- Home Screen:
  Carousel Slider: Displays a rotating list of advertisements highlighting various products or promotional offers.
- Featured Products:
  Lists recommended or highlighted products.
- Featured Categories:
  Highlights specific categories to enhance product discovery.

- Brands Section:
  Showcases recognized brands associated with the offered products.
- Categories Navigation:
  Category Exploration: Allows users to navigate through hierarchical categories and subcategories, providing a comprehensive view of available products within each section.
- Product Details:
  Product Display: Provides a detailed view of individual products, including images, descriptions, and other relevant information.
  Variants Selection: Enables users to choose product variations (e.g., sizes, colors) and specify quantities for adding items to the cart.
- Cart, Wishlist, and Checkout:
  Cart Management: Allows users to add products to their cart, view cart items, and proceed to checkout.
  Wishlist Functionality: Users can add desired products to their wishlist for future reference.
  Checkout Process: Users can proceed to checkout, requiring an added address for completion.
- Offers and Discounts:
  Sales and Discounts: Showcases time-bound sales and products offering discounts or special promotions.
  Limited-Time Offers: Highlights time-sensitive deals and promotional offers.
- User Profile and Side Menu:
  Profile Overview: Displays user-specific information, including counters for orders, wishlist, cart products, and loyalty points earned from purchases.
- Address Management:
  Allows users to add, edit, or view their addresses for shipping.
- Order History:
  Provides access to a history of previous orders for reference.
- Points and Rewards:
  Shows the points accrued from purchases as part of a loyalty program.
- Notification Screen:
  Order Status Notifications: Provides real-time updates on the latest status of orders, ensuring users are informed about their purchase progress.

Note:
Login Requirement: Actions like adding to cart or wishlist may require users to log in or create an account for authentication and personalized experiences.
Enhanced User Experience: The application aims to provide a comprehensive shopping experience by offering categorized navigation, personalized recommendations, and efficient access to cart and order information.

6. Configuration

1. **Server URL Configuration:**
   Locate Base API Service:
   Navigate to the network folder in the remote directory within the data section of the

project structure.Access the base_api_service.dart file.
Update Server URL:
Inside base_api_service.dart, locate the variable defining the root URL used for server communication.
Note: Currently, the server might not be operational. Update the server URL when the server is available.

2. **Application State Configuration:**
   Access App BLoC:
   Locate the app_bloc.dart file within the blocs section of the project.
   This BLoC (Business Logic Component) manages the overall application state.
   Configuration Settings:
   Check for configuration settings or parameters within the app_bloc.dart file that impact the application's behavior, such as initial state, default values, or global settings.
   Make necessary adjustments according to project requirements or specific configurations.

Note:
Server Status: As mentioned, the server might not be operational currently. Ensure the server is up and running to facilitate communication between the application and the backend.
Configuration Review: Regularly review and update configuration files, URLs, or global settings as needed, especially when deploying to different environments or working with new servers or APIs.
Development Environment Setup: If working in different environments (e.g., development, staging, production), ensure proper configuration for each environment.
Include a subsection or a note titled "Theme Configuration" or "Theming Approach" where you can briefly describe the situation regarding the themes in the application:

Theming Approach

Previous Theme Configuration:
There were two themes previously implemented in the application. the customer requests to consolidate these themes into a single one.

Changes Made:

We commented on the function responsible for the change.
Note that despite the commented code, the remaining theme continued to function, which may have been due to the theme being applied elsewhere or due to theme inheritance.

Note for Developers:

Commented Function Behavior:
The previously commented theme function(s) within the codebase can be uncommented without affecting the functionality of the active theme.
Consolidated Themes - Light and Dark:
The application has transitioned from two distinct themes (light and dark) to a unified theme concept.
Despite this consolidation, the application continues to utilize the same color schemes for both light and dark themes.

Unified Color Scheme:
Color Usage Consistency:
Emphasize that any new color additions or modifications should accommodate both light and dark themes.
When incorporating a new color, ensure its compatibility and visibility across both themes by defining it appropriately for each.
Note for Adding Colors:
Developing with Light and Dark Themes:
Developers should consider both light and dark theme appearances when introducing new colors into the application.
When defining new colors, ensure they are adequately adjusted or incorporated for both themes to maintain visual consistency and user experience across the application.
Maintenance and Updates:
Consistency Check:
During code maintenance or updates, verify that newly introduced colors are appropriately applied to both the light and dark themes.
Conduct thorough testing to ensure the visibility and aesthetic appeal of new colors under both theme settings.

Developer Note: Localization Methodology

Localization Technique:

Context-based Localization:The application implements localization and translations by utilizing the context in Flutter for string translations.
Strings for translations are directly incorporated within Dart files, allowing direct access using the context object.

*Absence of JSON File:*Non-Usage of JSON for Translations:
the application does not employ a JSON file for storing translations; instead, translations are directly embedded within Dart files.

Accessing Static Context:
Static Context Reference:
Developers can find a static context object accessible across the application where localization is required.
this static context reference when implementing or retrieving translations in the codebase.

Further Considerations:
Context-based Workflow:
Developers are encouraged to use the available static context for seamless localization implementation and avoid direct hardcoding of strings throughout the codebase.
Consideration should be given to adopting dedicated localization packages or methodologies for future scalability and ease of translation management.

7.Versioning Configuration

1. Versioning Setup:

YAML File Configuration:

Locate the YAML file responsible for managing version information within the project directory.

The version structure follows the format major.minor.patch+build, where:

major.minor.patch: Denotes the version numbers for different components of the application (e.g., major feature updates, minor changes, bug fixes).

+build: Includes platform-specific build numbers (iOS and Android).

2. iOS and Android Versioning:

iOS Versioning:

The section before the + sign represents the iOS version number.

Any changes made to this section should correspond to updates on the server to ensure synchronization.

Android Versioning:

The number after the + sign denotes the Android version number.

This number might increment independently based on Android-specific updates.

3. App BLoC Integration:

App BLoC Version Check:

The App BLoC within the application actively monitors the application's version number.

Any discrepancies or changes in version numbers trigger specific actions or checks within the application logic.

Note:

Synchronized Versioning: Ensuring synchronization between the server and the application's version numbers is crucial for consistent functionality and compatibility across platforms.

Platform-Specific Builds: The platform-specific build numbers help differentiate versions for iOS and Android, facilitating version-specific functionalities or updates.

Versioning Implications: Changing version numbers might trigger specific app behaviors, updates, or compatibility checks within the application's logic.

Maintaining Consistency: Regularly update version numbers in the YAML file, considering both platform-specific requirements and server synchronization for seamless functionality.

8. Usage Example: Retrieving Brands

1. Screen Initialization:

Retrieve Brands in Screen:

When the screen initializes, it calls the respective Cubit responsible for handling brand-related data retrieval.

here we add a listener on the scrolling to make the call whenever it reach the end of the page

```dart
@override
void initState() {
  super.initState();
  fetchData();
  _scrollController.addListener(() {
    if (_scrollController.position.pixels ==   Apple, 4/24/22, 3:34 PM • the
        _scrollController.position.maxScrollExtent) {
      brandProductsCubit.getBrandProducts(
          brandId: widget.id, page: ++_page, name: _searchController.text);
    }
  });
}
fetchData() async {
  _page = 1;
  brandProductsCubit.getBrandProducts(
      brandId: widget.id, page: _page, name: _searchController.text);
}
```

2. Cubit Logic:

Handling Data Retrieval:

Within the Cubit class (responsible for state management and logic), a method is implemented to fetch brand data based on the requested page number.

The Cubit manages the business logic and state changes related to fetching brand data.

3. Repository Function:

Brand Data Request:

The Cubit calls a method within the repository class associated with brand data retrieval.

The repository method specifies the endpoint to access and the type of data expected to be returned (e.g., a list of brand models).

```dart
Future<ProductMiniResponse> getBrandProducts(   Apple, 4/24/22, 3:34 PM • the already exi
    {int brandId = 0, page, name}) async {
  try {
    dynamic response = await _apiService.getRequest(
        "${ApiEndPoints.brandProducts}/${brandId.toString()}?&page=$page&name=$name");

    return productMiniResponseFromJson(response);
  } catch (e) {
    rethrow;
  }
}
```

4. API Service Interaction:

API Endpoint Definition:

Inside the repository, the endpoint for fetching brand data is defined.

The repository leverages the API Service to handle the API request and response handling.

5. Model Definition:
Brand Data Model:
The brand data model is defined in the models directory, representing the structure of brand-related information received from the API.
This model outlines the attributes and structure of brand data received from the backend.

Note:
Cubit Pattern: Utilizing Cubit architecture separates business logic and state management from the UI, promoting a more organized and scalable structure.
Repository as Mediator: The repository acts as a mediator between the Cubit and the API Service, abstracting data retrieval details.
Clear Endpoint Definition: Defining endpoints and expected data types in the repository ensures consistency and clarity in data retrieval processes.
Model Consistency: The defined model structure ensures uniformity and clarity in handling brand data across the application.

9. Troubleshooting and Debugging

1. Dependency Issues:
Problem: Incompatibility or conflicts between dependencies causing build errors or runtime issues.
Solution: Ensure compatible versions of packages are used. Run flutter clean and flutter pub get to clear caches and retrieve dependencies again.

2. Gradle Build Failures:
Problem: Gradle build failures, especially in Android projects.
Solution: Verify Gradle settings, update Gradle versions, and check for network issues. Run flutter clean and try rebuilding the project.

3. Flutter SDK Update Problems:
Problem: Issues arising after updating Flutter SDK.
Solution: Ensure proper SDK installation and update. Check for breaking changes or deprecated APIs in the new SDK version.

4. Plugin Compatibility:
Problem: Plugins not working correctly or causing conflicts.
Solution: Check plugin documentation for compatibility with Flutter versions. Report issues to plugin maintainers or try using alternative plugins.

5. Device Emulator Issues:
Problem: Emulator not launching or encountering errors.
Solution: Restart the emulator, ensure it's properly configured, or try using a different emulator. Update emulator images if necessary.

6. IDE-Specific Problems:
Problem: IDE-related issues such as hot reload not working or code analysis errors.

Solution: Restart the IDE, clear caches, and reconfigure settings. Update IDE and Flutter/Dart plugins/extensions.

7. Platform-Specific Bugs:
Problem: Bugs occurring on a specific platform (iOS/Android) but not on others.
Solution: Test and debug on the affected platform. Check platform-specific code or settings.

8. Network Connectivity:
Problem: API calls failing due to network issues or incorrect URL setup.
Solution: Check network connectivity, verify URL configurations, and handle exceptions for network calls.

9. Memory Leaks or Performance Issues:
Problem: App crashing due to memory leaks or performance issues.
Solution: Use Flutter DevTools to analyze memory usage, optimize code, and employ best practices for memory management.

10. Flutter Engine Issues:
Problem: Rare cases of Flutter engine errors or crashes.
Solution: Report issues to the Flutter team, try updating to the latest Flutter stable release, or check GitHub issues for similar problems and workarounds.

10. Deployment Instructions

1. Understand Deployment Requirements:
Customer Preferences:
Identify the customer's preferred deployment platform (Google Play Store, Apple App Store, etc.) and any specific requirements or changes requested for re-deployment.

2. Bundle ID Modification:
Bundle ID Change (if required):
If re-deploying to new accounts or different platforms, modify the application's bundle ID accordingly.
Update bundle IDs in project configurations to match the requirements of the new deployment.

3. Version Number Update:
Incrementing Version Number:
Increment the version number (iOS and Android) in the project's configuration files (e.g., pubspec.yaml for Flutter projects).
Ensure the version number complies with the requirements of the deployment platform.

4. Review Google and Apple Guidelines:
Platform-Specific Guidelines:
Refer to the latest Google Play Store and Apple App Store guidelines for app submission and deployment requirements.
Check for any recent changes or updates in their respective documentation.

5. Configuration Adjustments:
Check Configuration Files:
Review and update any configuration files related to deployment settings, URLs, API keys, or environment-specific variables.

6. Testing and Quality Assurance:
Quality Checks:
Conduct thorough testing after making modifications to ensure the application functions correctly post-deployment.
Check for any unforeseen issues or bugs resulting from the changes made for re-deployment.

7. Re-submission Process:
Submission to App Stores:
Follow the submission process for Google Play Store and Apple App Store accordingly.
Submit the updated application version adhering to the respective platform's guidelines and policies.

Note:
Account Credentials: Ensure access to the respective developer accounts associated with the deployment platforms.
Documentation Updates: Keep records of changes made during the re-deployment process for future reference or documentation purposes.
Compliance and Policies: Adhere to the guidelines, policies, and regulations of the app stores to prevent any rejection or delay in the re-submission process.

11. Additional Resources

1. Official Documentation:
Flutter Official Documentation: Extensive resources, guides, and API references provided by the Flutter team.

2. Community Forums and Platforms:
Stack Overflow - Flutter: Questions and answers related to Flutter programming by the developer community.
Flutter Community: A collection of articles, packages, and resources contributed by the Flutter community.

3. GitHub Repositories:
Flutter Samples: Official Flutter samples showcasing various Flutter features and best practices.
Awesome Flutter: A curated list of awesome Flutter libraries, tools, and resources.

4. Flutter Packages:
Pub.dev: Official repository for Flutter packages. Developers can search and find packages for various functionalities required in their applications.

5. Medium Articles and Blogs:
Medium - Flutter Community: Articles, tutorials, and insights shared by Flutter enthusiasts and experts.
Flutter Dev Blogs: Official Flutter blogs offering insights into new features, updates, and best practices.

6. Flutter DevTools:
Flutter DevTools: Official debugging and performance analysis tool for Flutter developers.

12. Conclusion

This documentation aims to empower developers with a comprehensive understanding of the application's structure, features, deployment procedures, and recommended resources, fostering a streamlined development experience for maintaining, improving, and expanding the Touri application in Flutter.