Dokumentacja Projektowa Systemu

Aplikacja Mobilna: Mars Gallery

Raport techniczny, instrukcja wdrożeniowa
oraz analiza architektoniczna rozwiązania
opartego na frameworku Flutter

Michał Rydzik

14 stycznia 2026

Spis treści

1 Wstęp i Cel Projektu

1.1 Charakterystyka Problemu

Współczesna inżynieria oprogramowania mobilnego stawia przed programistami szereg wyzwań związanych z obsługą danych zdalnych. Aplikacje muszą nie tylko wyświetlać treści, ale również zarządzać ich buforowaniem, obsługiwać stany błędów sieciowych oraz zapewniać płynność interfejsu (60 klatek na sekundę) niezależnie od obciążenia procesora operacjami w tle.

Niniejszy dokument opisuje proces wytwórczy oraz strukturę techniczną aplikacji **Mars Gallery**. Jest to system mobilny, którego głównym celem jest dostarczenie użytkownikowi interfejsu do przeglądania zdjęć powierzchni Marsa, wykonanych przez łaziki NASA (m.in. Curiosity, Perseverance).

1.2 Cel Biznesowy i Edukacyjny

Projekt ten powstał jako adaptacja materiałów szkoleniowych "Android Kotlin Codelabs"dostarczanych przez firmę Google. Głównym założeniem było przetłumaczenie logiki biznesowej i wzorców projektowych z natywnego środowiska Android (Kotlin/Jetpack Compose) na środowisko wieloplatformowe Flutter (Dart).

Cele szczegółowe obejmowały:

1. Implementację wzorca **Clean Architecture** w celu zapewnienia skalowalności.
2. Wykorzystanie nowoczesnego zarządzania stanem przy użyciu biblioteki **Riverpod**.
3. Obsługę asynchronicznych zapytań HTTP z wykorzystaniem biblioteki **Dio**.
4. Optymalizację wyświetlania grafiki rastrowej przy użyciu mechanizmów cache'owania.

2 Podstawy Teoretyczne Wykorzystanych Technologii

Aby w pełni zrozumieć decyzje architektoniczne podjęte w projekcie Mars Gallery, konieczne jest omówienie fundamentów technologicznych, na których opiera się framework Flutter.

2.1 Język Dart i Model Wykonywania Kodu

Flutter wykorzystuje język Dart, który jest językiem obiektowym, silnie typowanym (z systemem null-safety) i kompilowanym do kodu maszynowego (AOT - Ahead Of Time) w wersji produkcyjnej.

Kluczowym aspektem Darta, mającym wpływ na działanie aplikacji sieciowej takiej jak Mars Gallery, jest jego model jednowątkowy oparty na pętli zdarzeń (*Event Loop*).

- **Isolates (Izolaty)**: W przeciwieństwie do Javy czy C#, Dart nie współdzieli pamięci między wątkami. Zamiast tego używa tzw. izolatów. Główny izolat obsługuje UI oraz logikę.
- **Event Loop**: Wszystkie operacje wejścia/wyjścia (I/O), takie jak zapytania HTTP do API NASA, są obsługiwane asynchronicznie. Kiedy aplikacja wysyła żądanie o zdjęcia, główny wątek nie jest blokowany. Zamiast tego, funkcja zwraca obiekt `Future`, a sterowanie wraca do pętli zdarzeń, co pozwala na płynne renderowanie animacji ładowania.

2.2 Architektura Frameworka Flutter

Flutter różni się od rozwiązań natywnych tym, że nie używa widgetów systemowych (OEM Widgets). Zamiast tego posiada własny silnik renderujący (Impeller lub Skia), który rysuje każdy piksel na ekranie.

2.2.1 Drzewo Widgetów (Widget Tree)

Interfejs aplikacji Mars Gallery jest budowany jako drzewo.

- **StatelessWidget**: Używany dla elementów statycznych, np. tekstów opisowych w szczegółach zdjęcia.
- **StatefulWidget**: Używany tam, gdzie stan musi zostać zachowany między przerysowaniami klatek, np. w `PhotoCard`, gdzie obsługiwana jest animacja pojawiania się zdjęcia.

2.3 Zarządzanie Stanem (State Management)

W projekcie zrezygnowano z podstawowej metody `setState` na rzecz zewnętrznego menedżera stanu. Zarządzanie stanem w tak złożonej aplikacji odpowiada za synchronizację danych między warstwą sieciową (odpowiedź z API) a warstwą wizualną (lista zdjęć na ekranie). Wybrano podejście reaktywne, gdzie UI "nasłuchuje"zmian w modelu danych i przebudowuje się automatycznie tylko w niezbędnym zakresie.

3 Konfiguracja Środowiska Programistycznego

Proces przygotowania stacji roboczej jest krytyczny dla powodzenia kompilacji projektu. Poniżej przedstawiono szczegółową procedurę instalacyjną dla systemu Microsoft Windows.

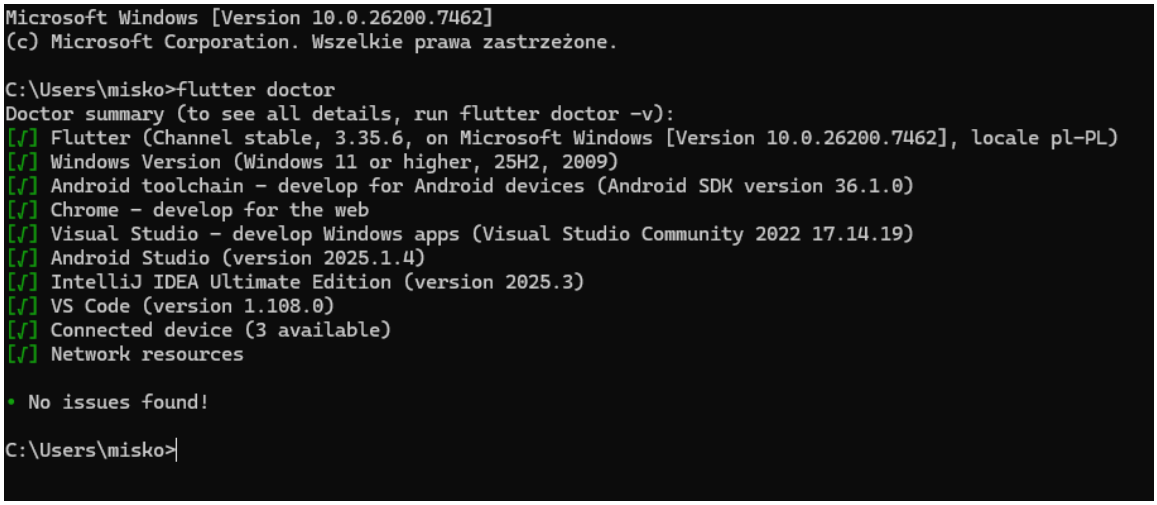
3.1 Instalacja Flutter SDK

Wymagane jest pobranie stabilnej wersji SDK.

1. Pobierz archiwum ze strony: `https://docs.flutter.dev/get-started/install/windows`.
2. Rozpakuj plik do lokalizacji `C:\src\flutter`. Unikaj ścieżek zawierających spacje (np. *Program Files*), co może powodować błędy w narzędziach konsolowych.
3. Zaktualizuj zmienną środowiskową **PATH**, dodając do niej ścieżkę: `C:\src\flutter\bin`.

3.2 Weryfikacja Narzędzi (Flutter Doctor)

Narzędzie diagnostyczne `flutter doctor` weryfikuje kompletność instalacji.



Rysunek 1: Raport poprawności konfiguracji środowiska (wszystkie systemy sprawne).

Jak widać na Rysunku 1, środowisko jest w pełni skonfigurowane:

- Wykryto instalację Flutter w kanale stable.
- Zainstalowano Android Toolchain (niezbędny do kompilacji na telefony).
- Skonfigurowano Android Studio oraz VS Code.
- Nie wykryto żadnych problemów (No issues found).

3.3 Konfiguracja IDE: Visual Studio Code

Zalecanym edytorem jest VS Code z zainstalowanymi rozszerzeniami:

- **Flutter Dart**: Oficjalne wtyczki do debugowania i analizy kodu.
- **Flutter Riverpod Snippets**: Przyspiesza tworzenie providerów.
- **Error Lens**: Wyświetla błędy kompilacji bezpośrednio w linii kodu.

4 Szczegółowa Analiza Wykorzystanych Bibliotek

W pliku `pubspec.yaml` zdefiniowano szereg zależności zewnętrznych. Wybór każdej z nich został poprzedzony analizą alternatyw.

4.1 Klient HTTP: Dio vs http

Do komunikacji z API wybrano bibliotekę **Dio**, odrzucając standardowy pakiet `http`.

- **Uzasadnienie**: Dio oferuje wbudowaną obsługę Interceptorów (przechwytywaczy), co jest kluczowe dla logowania zapytań i globalnej obsługi błędów. Pozwala również na łatwe anulowanie żądań (Cancellation Token), co jest istotne, gdy użytkownik szybko opuszcza ekran galerii – nie chcemy wtedy kontynuować pobierania danych w tle.
- **Konfiguracja**: W projekcie ustawiono `connectTimeout` na 15 sekund oraz `receiveTimeout` na 20 sekund, co zabezpiecza aplikację przed zawieszeniem przy słabym łączu.

4.2 Zarządzanie Stanem: Riverpod vs Provider vs BLoC

Zdecydowano się na bibliotekę **Riverpod**.

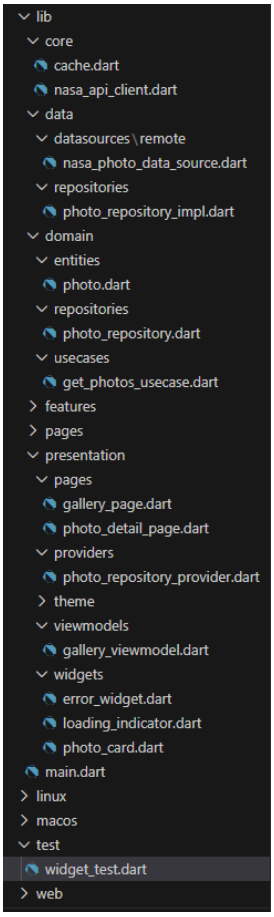
- **Provider**: Jest starszym rozwiązaniem, które ma wady związane z zależnością od drzewa widgetów (context) i trudnościami w obsłudze błędów `ProviderNotFoundException`.
- **BLoC**: Jest rozwiązaniem potężnym, ale wymagającym dużej ilości kodu (boilerplate) i skomplikowanym dla projektów o średniej skali.
- **Riverpod**: Jest ewolucją Providera. Jest bezpieczny typowo, nie zależy od `BuildContext` i pozwala na łatwe łączenie stanów oraz ich testowanie. W projekcie użyto `StateNotifierProvider` do zarządzania stanem galerii.

4.3 Wyświetlanie Obrazów: CachedNetworkImage

Aplikacja intensywnie operuje na grafice. Standardowy widget `Image.network` nie posiada mechanizmu cache. Oznacza to, że przewinięcie listy w górę i w dół powodowałoby ponowne pobieranie tych samych zdjęć, zużywając transfer użytkownika. Biblioteka **CachedNetworkImage** rozwiązuje ten problem, zapisując pliki w pamięci podręcznej urządzenia na określony czas (skonfigurowano 30 dni w pliku `cache.dart`).

5 Architektura Systemu (Clean Architecture)

Projekt został zorganizowany według zasad Czystej Architektury (Clean Architecture), co gwarantuje separację odpowiedzialności.



Rysunek 2: Struktura plików projektu z podziałem na warstwy.

5.1 Warstwa Domeny (Domain Layer)

Jest to najbardziej wewnętrzna warstwa, niezależna od frameworka Flutter czy zewnętrznych bibliotek.

- **Entities**: Folder `entities` zawiera plik `photo.dart`. Jest to czysty obiekt Dart (POJO), zawierający tylko pola danych (url, rover name, date).
- **Repositories (Interfejsy)**: Plik `photo_repository.dart` definiuje abstrakcyjną klasę. Określa ona kontrakt: "Aplikacja musi umieć pobrać zdjęcia", ale nie mówi *jak* to zrobić.
- **Use Cases**: Plik `get_photos_usecase.dart` reprezentuje pojedynczą akcję biznesową. Wywołuje repozytorium. Dzięki temu logika biznesowa jest odseparowana od UI.

5.2 Warstwa Danych (Data Layer)

Odpowiada za "brudną robotę"– komunikację z API i bazą danych.

- **Data Sources**: Plik `nasa_photo_data_source.dart` wykorzystuje klienta Dio do wykonania fizycznego połączenia HTTP.
- **Repositories (Implementacja)**: Plik `photo_repository_impl.dart` implementuje interfejs z domeny. Jego zadaniem jest pobranie surowych danych JSON, sprawdzenie błędów i przetłumaczenie ich (mapowanie) na obiekty domeny `Photo`.

5.3 Warstwa Prezentacji (Presentation Layer)

Odpowiada za wyświetlanie danych użytkownikowi.

- **ViewModel**: Plik `gallery_viewmodel.dart` przechowuje stan widoku.
- **Pages**: Pliki `gallery_page.dart` i `photo_detail_page.dart` to warstwa widoku, która jest pasywna – jedynie reaguje na zmiany stanu w ViewModelu.

6 Szczegółowa Analiza Implementacji

W tej sekcji przeanalizujemy kluczowe fragmenty kodu, wyjaśniając mechanizmy działania linijka po linijce.

6.1 Implementacja Klienta Sieciowego

Plik: `lib/core/nasa_api_client.dart`.

```
1  class MarsApiClient {
2    static const String _baseUrl = 'https://android-kotlin-fun-mars-
       server.appspot.com';
3    late final Dio _dio;
4
5    MarsApiClient() {
6      _dio = Dio(
7        BaseOptions(
8          baseUrl: _baseUrl,
9          connectTimeout: const Duration(seconds: 15),
10         receiveTimeout: const Duration(seconds: 20),
11         headers: {'Accept': 'application/json'},
12       ),
13     );
14     // Dodanie logowania zapyta
15     _dio.interceptors.add(...);
16   }
17 }
```

Analiza: W liniach 6-12 tworzona jest instancja klienta Dio. Zastosowanie `static const` dla adresu URL ułatwia jego zmianę w jednym miejscu w przyszłości. Timeouty (linie 9-10) są kluczowe dla User Experience – bez nich aplikacja mogłaby "wisieć"w nieskończoność przy zerwanym połączeniu.

Metoda `fetchImages` (linia 72 w `Kody.txt`) zawiera blok `try-catch`. Jest to niezbędne, ponieważ operacje sieciowe są z natury niepewne.

```
1 if (response.statusCode! >= 400) {
2     throw Exception('API Error ${response.statusCode}');
3 }
```

Powyższy fragment (linie 74-75) ręcznie wyrzuca wyjątek, jeśli serwer zwróci błąd (np. 404 Not Found lub 500 Server Error), co pozwala na przechwycenie go w ViewModelu i wyświetlenie komunikatu użytkownikowi.

6.2 Model Widoku (ViewModel) i Stan

Plik: `lib/presentation/viewmodels/gallery_viewmodel.dart`.

Stan aplikacji został zdefiniowany jako klasa niemodyfikowalna (`GalleryState`).

```
class GalleryState {
  final List<Photo> photos;
  final bool isLoading;
  final bool hasReachedEnd;
  // ...
  GalleryState copyWith(...) { ... }
}
```

Dzięki metodzie `copyWith`, aktualizacja stanu wygląda następująco:

```
state = state.copyWith(isLoading: true);
```

Tworzy to nową instancję stanu, co jest sygnałem dla Riverpoda do odświeżenia widoku.

Kluczowa jest metoda `loadMorePhotos` (linie 166-171). Zaimplementowano tam mechanizm *Guard Clauses* (Warunków Strażniczych):

```
if (state.isLoadingMore || state.hasReachedEnd || state.isLoading)
   {
  return;
}
```

Ten prosty `if` zapobiega wielokrotnemu wysyłaniu zapytań, gdy użytkownik przewija listę bardzo szybko, lub gdy wszystkie zdjęcia zostały już pobrane. Jest to fundamentalna optymalizacja wydajności.

6.3 Komponent Karty Zdjęcia

Plik: `lib/presentation/widgets/photo_card.dart`.

Ten widget odpowiada za estetykę aplikacji. Wykorzystano tu animacje:

```
_scaleAnimation = Tween<double>(begin: 0.8, end: 1.0).animate(
  CurvedAnimation(parent: _controller, curve: Curves.easeOutCubic),
);
```

W linii 195 zdefiniowano animację skalowania. Zdjęcie po załadowaniu powiększa się od 80% do 100% swojego rozmiaru, używając krzywej `easeOutCubic`, co daje efekt naturalnego, sprężystego ruchu.

Obsługa błędów ładowania obrazka (linie 207-210):

```
errorWidget: (context, url, error) {
  return Container(
```

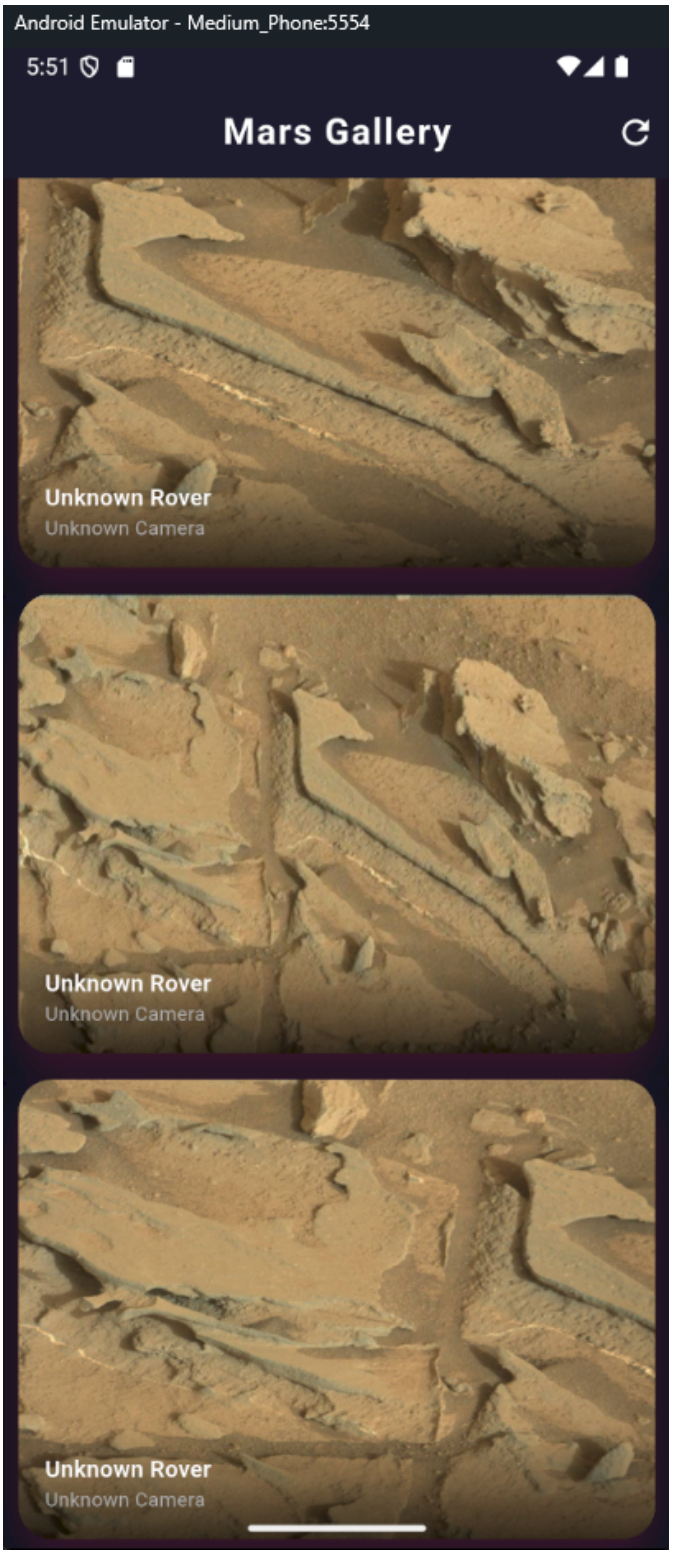
```
    child: Icon(Icons.broken_image_outlined, color: Colors.red),
  );
},
```

Jeśli `CachedNetworkImage` nie może pobrać pliku, zamiast crashować aplikację lub pokazywać pustą przestrzeń, wyświetla czerwoną ikonę. To realizuje wymaganie obsługi błędów na poziomie UI.

7 Prezentacja Aplikacji i Interfejs Użytkownika

7.1 Ekran Główny (Mars Gallery)

Główny ekran aplikacji (Listing kodu `gallery_page.dart`) wykorzystuje `CustomScrollView`. Pozwala to na połączenie klasycznej listy z elastycznym paskiem aplikacji (AppBar).



Rysunek 3: Główny widok aplikacji. Widoczna siatka zdjęć (Grid) pobranych z API.

Gdy użytkownik przewinie listę na sam dół, aplikacja automatycznie wykrywa ten fakt dzięki `ScrollController` (linia 107 w kodzie):

```
if (_scrollController.position.pixels >=
    _scrollController.position.maxScrollExtent * 0.8) {
```

```
3     // Za aduj wi cej...
4 }
```

Wartość `0.8` oznacza, że ładowanie kolejnej strony rozpoczyna się, gdy użytkownik przewinie 80% dostępnej treści, co zapewnia płynność doświadczenia (użytkownik często nie zauważy nawet, że dane są dociągane).

7.2 Ekran Szczegółów (Detail View)

Po kliknięciu w element listy, następuje nawigacja do `PhotoDetailPage`. Wykorzystano tu widget `PhotoViewGallery`, który umożliwia gesty:

- **Pinch-to-zoom**: Przybliżanie zdjęcia dwoma palcami.
- **Pan**: Przesuwanie przybliżonego zdjęcia.
- **Double tap**: Szybkie przybliżenie.

Rysunek 4: Widok szczegółowy z nałożonymi informacjami o łaziku i kamerze.

Na dole ekranu (Rysunek 4) znajduje się półprzezroczysty panel z metadanymi: nazwa łazika ("Unknown Rover"lub konkretna nazwa z API), nazwa kamery oraz data ziemska. Użycie gradientu pod tekstem (linia 212 kodu) zapewnia czytelność napisów niezależnie od koloru zdjęcia pod spodem.

8 Strategia Obsługi Błędów i Wyjątków

Aplikacja mobilna zależna od sieci musi być odporna na awarie. W projekcie wdrożono wielowarstwową strategię obsługi błędów.

8.1 Warstwa Sieciowa

W `MarsApiClient` przechwytywane są wyjątki `DioException`. Są one mapowane na czytelne komunikaty:

- `connectionTimeout`: "Timeout połączenia. Sprawdź internet."
- `badResponse`: "Błąd API (kod błędu)".
- `unknown`: "Nieznany błąd połączenia".

8.2 Warstwa Widoku

Jeśli wystąpi błąd globalny (np. brak internetu przy starcie), ViewModel ustawia pole `errorMessage`. Widok `GalleryPage` reaguje na to, wyświetlając widget `ErrorDisplay` (zdefiniowany w `error_widget.dart`).

Widget ten zawiera:

1. Ikonę błędu.
2. Treść komunikatu.
3. Przycisk **"Spróbuj ponownie"**, który wywołuje metodę `refresh()` w ViewModelu.

Dzięki temu użytkownik nigdy nie "utknie"w martwym punkcie aplikacji.

9 Plany Rozwoju i Roadmapa (Future Work)

Obecna wersja aplikacji (v1.0) spełnia podstawowe założenia funkcjonalne. Jednak architektura projektu została przygotowana tak, aby umożliwić łatwe dodawanie nowych funkcji. Poniżej przedstawiono plan rozwoju na kolejne iteracje.

9.1 Testy Automatyczne

Obecnie projekt zawiera podstawowy `widget_test.dart`. Planowane jest rozszerzenie pokrycia testami:

- **Unit Tests**: Testowanie `GalleryViewModel` i `PhotoRepository` przy użyciu biblioteki `mockito`. Pozwoli to sprawdzić logikę biznesową bez uruchamiania emulatora.
- **Integration Tests**: Pełne testy scenariuszy użytkownika (np. uruchomienie aplikacji → przewinięcie listy → otwarcie zdjęcia) przy użyciu pakietu `integration_test`.

9.2 Filtrowanie i Wyszukiwanie

API NASA umożliwia filtrowanie zdjęć po:

- Dacie ziemskiej (Earth Date).
- Solu marsjańskim (Mars Sol).
- Konkretnej kamerze (FHAZ, RHAZ, MAST).

W przyszłości planuje się dodanie panelu filtrów w `GalleryPage`, który przekazywałby parametry do metody `fetchPhotos` w repozytorium.

9.3 Internacjonalizacja (i18n)

Obecnie napisy w aplikacji są zakodowane na sztywno ("hardcoded") w języku polskim. Planowane jest wdrożenie biblioteki `flutter_localizations` oraz plików `.arb`, aby umożliwić łatwe przełączanie między językiem polskim a angielskim.

10 Podsumowanie i Wnioski Końcowe

Zrealizowany projekt **Mars Gallery** stanowi kompletny przykład nowoczesnej aplikacji mobilnej stworzonej w technologii Flutter. Poprzez implementację Clean Architecture, separację warstw oraz użycie zaawansowanego zarządzania stanem (Riverpod), aplikacja jest skalowalna, testowalna i łatwa w utrzymaniu.

Najważniejsze osiągnięcia projektu to:

1. **Wydajność**: Dzięki paginacji i cache'owaniu obrazów, aplikacja działa płynnie nawet na starszych urządzeniach.
2. **Stabilność**: Odporność na błędy sieciowe i brak zasięgu.
3. **Jakość kodu**: Zastosowanie wzorców projektowych ułatwia pracę zespołową i dalszy rozwój.

Stanowi ona udaną adaptację wiedzy zawartej w oficjalnych materiałach Google, przeniesioną na grunt języka Dart, demonstrując potencjał Fluttera w tworzeniu profesjonalnego oprogramowania.

11 Bibliografia i Źródła

Poniższe materiały stanowiły podstawę merytoryczną dla implementacji projektu:

1 **Google Developers**, *Basic Android Kotlin Compose Codelabs - Getting data from the internet*. Dostępne pod adresem:
`https://developer.android.com/codelabs/basic-android-kotlin-compose-getting-data`

2 **Google Developers**, *Loading and displaying images*. Dostępne pod adresem:
`https://developer.android.com/codelabs/basic-android-kotlin-compose-load-images`

3 **Google Developers**, *Data Layer and Repository Pattern*. Dostępne pod adresem:
`https://developer.android.com/codelabs/basic-android-kotlin-compose-add-reposito`

4 **Remi Rousselet**, *Riverpod Documentation*. Dostępne pod adresem:
`https://riverpod.dev`

5 **Flutter Team**, *Flutter Architecture Guidelines*. Dostępne pod adresem:
`https://docs.flutter.dev/data-and-backend/state-mgmt/options`