UTM
UNIVERSITI TEKNOLOGI MALAYSIA
School of Professional and Continuing Education (SPACE)

**UNIVERSITI TEKNOLOGI MALAYSIA**

**SEMESTER 1, SESSION 2025/2026**

**SECJ3623:**

**MOBILE APPLICATION PROGRAMMING**

**GROUP NAME:**

**SCRUMSHANK REDEMPTION**

**Technical Report:**

**PetPal – Smart Pet Care Management Mobile Application**



| NAME | METRIC NUMBER |
|---|---|
| AIMAN FITRI BIN YUSRI | SX202209ECJHF04 |
| AMMAR IBRAHIM BIN MOHAMED | SX220326ECJHS04 |
| NABILAH ABD RAHMAN | SX202285ECJHF04 |
| EFFA AIZA SULAIMAN | SX202206ECJHF04 |

LECTURER'S NAME : DR MOHD SHAHIZAN OTHMAN

# Table of Contents

## 1.0 Executive Summary

PetPal is a comprehensive pet care management mobile application developed using Flutter and Firebase. The application connects pet owners with veterinarians, pet sitters, and hotel services while providing robust pet management, booking, and activity tracking capabilities. This report outlines the technical architecture, development approach, and implementation details of the PetPal application.

## 2.0 Development Approach and Workflow

### 2.1 Architecture Pattern

The application follows a BLoC (Business Logic Component) architecture pattern, which provides:

- Clear separation of concerns between UI, business logic, and data layers
- Reactive state management using streams
- Testability and maintainability
- Predictable state transitions

### 2.2 Project Structure

The application follows a layered architecture:

PetPalApp Architecture
── Presentation Layer (UI)
│ ├── Screens (Views)
│ └── Widgets (Reusable Components)
├── Business Logic Layer
│ ├── BLoCs (State Management)
│ └── Events & States
├── Data Layer
│ ├── Repositories (Data Abstraction)
│ ├── Services (External APIs)
│ └── Models (Data Structures)
└── Core/Utils
├── Constants
└── Helpers

### 2.3 Development Workflow

1. Feature-Based Development: Each feature (authentication, pet management, bookings) is developed as an independent module
2. Dependency Injection: All dependencies are injected through the widget tree using RepositoryProvider
3. State Management: BLoC pattern ensures unidirectional data flow
4. Firebase Integration: Backend-as-a-Service for authentication, database, storage, and notifications

## 3.0 Software, Tools, and Frameworks

### 3.1 Core Technologies

| Technology | Version | Purpose |
|---|---|---|
| Flutter | Latest Stable | Cross-platform mobile framework |
| Dart | 3.0+ | Programming language |
| Firebase | Latest | Backend services |

### 3.2 Key Dependencies

```
flutter_bloc:  ^8.0.0                        //  BLoC  pattern  implementation
firebase_core:   ^2.0.0                           //   Firebase   initialization
firebase_auth:    ^4.0.0                                 //    Authentication
cloud_firestore:    ^4.0.0                             //    NoSQL    database
firebase_storage:     ^11.0.0                          //     File     storage
firebase_messaging:     ^14.0.0                    //     Push     notifications
flutter_local_notifications:                                          ^15.0.0
pdf:                                PDF                                  generation
image_picker:                           Image                             selection
intl: Internationalization and date formatting
```

### 3.3 Development Tools

- IDE: Android Studio / Visual Studio Code
- Version Control: Git
- Firebase Console: Backend management
- Flutter DevTools: Debugging and profiling

## 4.0 Database and Data Handling Design

### 4.1 Firebase Firestore Structure

The application uses Cloud Firestore with the following collection structure:

```
firestore
├── users/
│   ├── {userId}
│   │   ├── email: string
│   │   ├── name: string
│   │   ├── phoneNumber: string
│   │   ├── address: string
│   │   ├── profileImageUrl: string
│   │   └── role: string (owner/vet/sitter)
│
├── pets/
│   ├── {petId}
│   │   ├── ownerId: string
│   │   ├── name: string
│   │   ├── species: string
│   │   ├── breed: string
│   │   ├── age: number
│   │   ├── imageUrl: string
│   │   └── medicalHistory: array
│
├── bookings/
│   ├── {bookingId}
│   │   ├── ownerId: string
│   │   ├── petId: string
│   │   ├── providerId: string (vet/sitter)
│   │   ├── serviceType: string
│   │   ├── date: timestamp
│   │   ├── status: string
│   │   └── notes: string
│
├── vets/
│   ├── {vetId}
│   │   ├── userId: string
│   │   ├── clinicName: string
│   │   ├── specialization: string
│   │   ├── rating: number
│   │   └── availability: array
│
├── sitters/
│   ├── {sitterId}
│   │   ├── userId: string
│   │   ├── experience: string
│   │   ├── hourlyRate: number
│   │   └── availability: array
│
└── activities/
    ├── {activityId}
        ├── petId: string
        ├── type: string (feeding/walking/medication)
        ├── timestamp: timestamp
        └── notes: string
```

### 4.2 Data Models

Example AppUser Model:

```
class AppUser {
  final String id;
  final String email;
  final String name;
  final String? phoneNumber;
  final String? address;
  final String? profileImageUrl;
  final UserRole role;

  AppUser({
    required this.id,
    required this.email,
    required this.name,
    this.phoneNumber,
    this.address,
    this.profileImageUrl,
    required this.role,
  });

  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'email': email,
      'name': name,
      'phoneNumber': phoneNumber,
      'address': address,
      'profileImageUrl': profileImageUrl,
      'role': role.toString(),
    };
  }

  factory AppUser.fromMap(Map<String, dynamic> map) {
    return AppUser(
      id: map['id'],
      email: map['email'],
      name: map['name'],
      phoneNumber: map['phoneNumber'],
      address: map['address'],
      profileImageUrl: map['profileImageUrl'],
      role: UserRole.values.firstWhere(
        (e) => e.toString() == map['role'],
      ),
    );
  }
}
```

## 5.0 Implementation of CRUD Operations

### 5.1 Create Operation

BLoC Event Handler for Adding a Pet:

```
on<AddPet>((event, emit) async {
  emit(state.copyWith(status: PetStatus.loading));

  try {
    String? imageUrl;

    if (event.imageFile != null) {
      imageUrl = await _storageService.uploadPetImage(
        event.pet.id,
        event.imageFile!,
      );
    }

    final pet = event.pet.copyWith(imageUrl: imageUrl);
    await _petRepository.addPet(pet);

    emit(state.copyWith(
      status: PetStatus.success,
      pets: [...state.pets, pet],
    ));
  } catch (e) {
    emit(state.copyWith(
      status: PetStatus.error,
      errorMessage: e.toString(),
    ));
  }
});
```

### 5.2 Read Operation

Real-time Data Subscription:

```
on<LoadPets>((event, emit) async {
  emit(state.copyWith(status: PetStatus.loading));

  await emit.forEach<List<Pet>>(
    _petRepository.getPetsByOwner(event.ownerId),
    onData: (pets) => state.copyWith(
      status: PetStatus.success,
      pets: pets,
    ),
    onError: (error, stackTrace) => state.copyWith(
      status: PetStatus.error,
      errorMessage: error.toString(),
    ),
  );
});
```

### 5.3 Update Operation

BLoC Handler for Updating a Pet:

```
on<UpdatePet>((event, emit) async {
  emit(state.copyWith(status: PetStatus.loading));

  try {
    String? imageUrl = event.pet.imageUrl;

    if (event.newImageFile != null) {
      if (imageUrl != null) {
        await _storageService.deleteFile(imageUrl);
      }

      imageUrl = await _storageService.uploadPetImage(
        event.pet.id,
        event.newImageFile!,
      );
    }

    final updatedPet = event.pet.copyWith(imageUrl: imageUrl);
    await _petRepository.updatePet(updatedPet);

    final updatedPets = state.pets.map((pet) =>
      pet.id == updatedPet.id ? updatedPet : pet
    ).toList();

    emit(state.copyWith(
      status: PetStatus.success,
      pets: updatedPets,
    ));
  } catch (e) {
    emit(state.copyWith(
      status: PetStatus.error,
      errorMessage: e.toString(),
    ));
  }
});
```

### 5.4 Delete Operation

BLoC Handler for Deleting a Pet:

```
on<DeletePet>((event, emit) async {
  emit(state.copyWith(status: PetStatus.loading));

  try {
    final pet = state.pets.firstWhere((p) => p.id == event.petId);

    if (pet.imageUrl != null) {
      await _storageService.deleteFile(pet.imageUrl!);
    }

    await _petRepository.deletePet(event.petId);

    final updatedPets = state.pets.where((p) => p.id !=
event.petId).toList();

    emit(state.copyWith(
      status: PetStatus.success,
      pets: updatedPets,
    ));
  } catch (e) {
    emit(state.copyWith(
      status: PetStatus.error,
      errorMessage: e.toString(),
    ));
  }
});
```

## 6.0 Important Source Code Snippets with Explanations

### 6.1 Main Application Initialization

From main.dart - Application entry point:

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform
  );

  Bloc.observer = AppBlocObserver();

  final notificationService = NotificationService(
    FirebaseMessaging.instance,
    FlutterLocalNotificationsPlugin(),
  );

  final authService = FirebaseAuthService(FirebaseAuth.instance);
  final firestoreService = FirestoreService(FirebaseFirestore.instance);
  final storageService = StorageService(FirebaseStorage.instance);
  final pdfService = PdfService();

  final authRepository = AuthRepository(
    authService: authService,
    firestoreService: firestoreService,
  );

  // ... other repositories initialization

  runApp(PetPalApp(
    authRepository: authRepository,
    // ... other dependencies
  ));
}
```

Explanation: Ensures Flutter bindings are initialized before Firebase, initializes all Firebase services and custom services, and creates repository instances with dependency injection for clean separation of concerns.

### 6.2 Dependency Injection Setup

Widget tree dependency configuration:

```
@override
Widget build(BuildContext context) {
  return MultiRepositoryProvider(
    providers: [
      RepositoryProvider.value(value: widget.authRepository),
      RepositoryProvider.value(value: widget.userRepository),
      RepositoryProvider.value(value: widget.petRepository),
      // ... other repositories
    ],
    child: MultiBlocProvider(
      providers: [
        BlocProvider(
          create: (_) => AuthBloc(widget.authRepository)
            ..add(const AuthStatusRequested()),
        ),
        BlocProvider(
          create: (_) => PetBloc(
            petRepository: widget.petRepository,
            storageService: widget.storageService,
          ),
        ),
        // ... other BLoCs
      ],
      child: MaterialApp(/* ... */),
    ),
  );
}
```

Explanation: MultiRepositoryProvider makes repositories available throughout the widget tree. MultiBlocProvider creates and provides BLoC instances with their required dependencies. Initial events are dispatched on creation.

### 6.3 Authentication Flow with BLoC Listener

```
BlocListener<AuthBloc, AuthState>(
  listenWhen: (previous, current) => previous.status != current.status,
  listener: (context, state) {
    if (state.status == AuthStatus.unauthenticated) {
      _navigatorKey.currentState?.pushNamedAndRemoveUntil(
        LoginScreen.routeName,
        (route) => false,
      );
    }
  },
  child: MaterialApp(
    navigatorKey: _navigatorKey,
    // ... routes
  ),
)
```

Explanation: BlocListener monitors auth state changes without rebuilding UI. listenWhen prevents unnecessary listener executions. Global navigation key allows navigation from outside widget context, and pushNamedAndRemoveUntil clears the navigation stack when logging out.

### 6.4 Firestore Service Implementation

```
class FirestoreService {
  final FirebaseFirestore _firestore;

  FirestoreService(this._firestore);

  CollectionReference collection(String path) {
    return _firestore.collection(path);
  }

  Future<DocumentSnapshot> getDocument(String path, String id) async {
    return await _firestore.collection(path).doc(id).get();
  }

  Stream<QuerySnapshot> getCollectionStream(
    String path,
    {List<WhereCondition>? where, String? orderBy}
  ) {
    Query query = _firestore.collection(path);

    if (where != null) {
      for (var condition in where) {
        query = query.where(condition.field, isEqualTo: condition.value);
      }
    }

    if (orderBy != null) {
      query = query.orderBy(orderBy);
    }

    return query.snapshots();
  }
}
```

Explanation: Wraps FirebaseFirestore for easier testing and mocking. Provides generic CRUD operations, supports real-time streams for reactive UI, and allows query composition with where clauses and ordering.

### 6.5 Storage Service for Image Handling

```
class StorageService {
  final FirebaseStorage _storage;

  StorageService(this._storage);

  Future<String> uploadPetImage(String petId, File imageFile) async {
    try {
      final ref = _storage.ref()
          .child('pets/$petId/${DateTime.now().millisecondsSinceEpoch}');

      final uploadTask = ref.putFile(imageFile);
      final snapshot = await uploadTask.whenComplete(() {});

      final downloadUrl = await snapshot.ref.getDownloadURL();
      return downloadUrl;
    } catch (e) {
      throw Exception('Failed to upload image: $e');
    }
  }

  Future<void> deleteFile(String url) async {
    try {
      final ref = _storage.refFromURL(url);
      await ref.delete();
    } catch (e) {
      throw Exception('Failed to delete file: $e');
    }
  }
}
```

Explanation: Encapsulates Firebase Storage operations, generates unique file paths, returns download URLs for Firestore storage, and handles cleanup when images are updated/deleted.

## 7.0 Technical Challenges and Solutions

### 7.1 Asynchronous Initialization

- Problem: Firebase and notification services require async initialization that could block app startup.
- Solution: Initialize critical services (Firebase) synchronously in main(), then initialize non-critical services (notifications) asynchronously in widget's initState().
- Outcome: Faster app startup with better user experience.

### 7.2 State Management Complexity

- Problem: Managing multiple interconnected states across different screens.
- Solution: Implemented BLoC pattern with separate domain-specific BLoCs (AuthBloc, PetBloc, BookingBloc).
- Outcome: Predictable state transitions, easy debugging, and testable business logic.

### 7.3 Real-time Data Synchronization

- Problem: Keeping UI synchronized with Firestore changes from multiple users/devices.
- Solution: Use Firestore streams instead of futures. BLoCs subscribe to streams using emit.forEach().
- Outcome: Automatic UI updates when Firestore data changes, no manual refresh required.

### 7.4 Navigation After Logout

- Problem: Users could navigate back to authenticated screens after logout.
- Solution: Use pushNamedAndRemoveUntil() to clear entire navigation stack on logout.
- Outcome: Complete navigation stack cleared, better security, consistent auth flow.

### 7.5 Image Upload and Management

- Problem: Handling image uploads, storage, and cleanup efficiently.
- Solution: Encapsulate image operations in StorageService, delete old images before uploading new ones.

- Outcome: Optimized storage usage, proper error handling, consistent image URLs.

### 7.6 Role-Based Access Control

- Problem: Different user types need access to different features.
- Solution: Use UserRole enum, implement conditional routing and UI rendering based on role.
- Outcome: Secure access control, customized user experience per role.

## 8.0 Security Considerations

### 8.1 Authentication Security

- Firebase Authentication handles password hashing and secure token management
- Email verification for new accounts
- Password reset functionality with secure tokens

### 8.2 Data Access Rules

Firestore security rules should enforce user-specific access:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read: if request.auth != null;
      allow write: if request.auth.uid == userId;
    }

    match /pets/{petId} {
      allow read: if request.auth != null;
      allow write: if request.auth != null &&
                    resource.data.ownerId == request.auth.uid;
    }

    match /bookings/{bookingId} {
      allow read: if request.auth != null &&
                  (resource.data.ownerId == request.auth.uid ||
                   resource.data.providerId == request.auth.uid);
    }
  }
}
```

### 8.3 Storage Security

- Firebase Storage rules for authenticated access only:
- User-specific paths prevent unauthorized access

## 9.0 Performance Optimizations

### 9.1 Lazy Loading

- BLoCs created only when needed
- Images loaded on demand with caching

### 9.2 Real-time Data Optimization

- Firestore queries with indexes for faster retrieval
- Limited collection queries with where clauses
- Pagination for large lists (recommended implementation)

### 9.3 State Management Efficiency

- listenWhen and buildWhen prevent unnecessary rebuilds
- emit.forEach handles stream subscriptions efficiently
- Immutable state with copyWith for predictable updates

## 10.0 Testing Strategy

### 10.1 Unit Testing

- BLoC business logic tested in isolation:
- Mock repositories and services
- Test state transitions and event handling

```
void main() {
  group('PetBloc', () {
    late PetRepository mockRepository;
    late PetBloc petBloc;

    setUp(() {
      mockRepository = MockPetRepository();
      petBloc = PetBloc(petRepository: mockRepository);
    });

    test('emits success state when pets are loaded', () async {
      when(() => mockRepository.getPetsByOwner(any()))
          .thenAnswer((_) => Stream.value([mockPet]));
```

```
      expectLater(
        petBloc.stream,
        emitsInOrder([
          isA<PetState>().having((s) => s.status, 'status',
PetStatus.loading),
          isA<PetState>().having((s) => s.status, 'status',
PetStatus.success),
        ]),
      );

      petBloc.add(LoadPets(ownerId: 'test-owner'));
    });
  });
}
```

### 10.2 Widget Testing

- UI components tested in isolation
- Mock BLoC states
- Test user interactions

### 10.3 Integration Testing

- Complete user flows tested
- Firebase integration tested
- Navigation flows tested

## 11.0 Future Enhancements

### 11.1 Recommended Technical Improvements

1. Pagination: Implement pagination for large lists
2. Offline Support: Enhanced offline capabilities with local database (Hive/Drift)
3. Error Tracking: Integrate Crashlytics for production monitoring
4. Analytics: Add Firebase Analytics for user behavior tracking
5. Automated Testing: Increase test coverage to 80%+
6. CI/CD: Set up automated build and deployment pipelines
7. Performance Monitoring: Integrate Firebase Performance Monitoring
8. Search Functionality: Implement Algolia or Elasticsearch
9. Payment Integration: Add Stripe/PayPal for booking payments
10. Chat Feature: Real-time messaging between owners and providers

## 12.0 Conclusion

PetPal demonstrates a well-architected Flutter application following industry best practices. The application successfully integrates multiple Firebase services (Authentication, Firestore, Storage, Messaging) with a robust state management solution, providing a solid foundation for a production-ready pet care management platform.

Key Strengths:

- Clean Architecture: Clear separation of concerns with BLoC pattern
- Scalability: Modular design allows easy feature additions
- Maintainability: Consistent code structure and patterns
- Real-time Capabilities: Leveraging Firebase for real-time data sync
- User Experience: Responsive UI with proper loading and error states
- Security: Firebase Authentication and Firestore rules for data protection