PocketBase with Dart (Part 2)

Student DB Example

Student Data Model

- In this example, we model a student with the following information.

```
Name
Age
Major
createdAt
```

- To store the information in the DB, we need additional information.

```
id // automatically generated from pocketbase
createdAt
```

JSON

```
{
    "name": "Alice Johnson",
    "age": 21,
    "major": "Computer Science",
    "createdAt": "2025-08-25 11:15:14.123Z"
}
```

Pocketbase RecordModel

- RecordModel is the representation of information in Pocketbase.

```
RecordModel {
  // Automatically generated Unique identifier
  id: "abc123def456",
  // Your actual data
  data: {
    "name": "Alice Johnson",
    "age": 21,
    "major": "Computer Science",
    "createdAt": "2025-08-25 11:15:14.123Z"
  },
}
```

models/student.dart

```dart
/// Student model for PocketBase
class Student {
  final String id;         // PocketBase document ID
  final String name;       // Student name
  final int age;           // Student age
  final String major;      // Student's major
  final DateTime createdAt; // Timestamp
  // constructor
  const Student({
    required this.id,
    required this.name,
    required this.age,
    required this.major,
    required this.createdAt,
  });
```

JSON service functions

```
  Map<String, dynamic> toJson() {
    return {
      'name': name,
      'age': age,
      'major': major,
      'createdAt': createdAt.toIso8601String(),
    };
  }
  factory Student.fromJson(Map<String, dynamic> map) {
    return Student(
      id: map['id'] as String? ?? '',
      name: map['name'] as String? ?? '',
      age: map['age'] as int? ?? 0,
      major: map['major'] as String? ?? '',
      createdAt: _parseDateTime(map['createdAt']),
    );
  }
```

Collection Setup

- We already created Student collection.
  - Check `scripts/create_collection.sh`

```
curl -X POST http://localhost:8090/api/collections \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer YOUR_ADMIN_TOKEN" \
  -d '{ ... }'
```

- We can use the collections.create() function.

```
await pb.collections.create(body: {
  'name': 'students',
  'type': 'base',
  'schema': [
    {'name': 'name', 'type': 'text', 'required': true},
    {'name': 'age', 'type': 'number', 'required': true},
    {'name': 'major', 'type': 'text', 'required': true},
    {'name': 'createdAt', 'type': 'date', 'required': true},
  ],
  'createRule': '@request.auth.id != ""',
  'updateRule': '@request.auth.id != ""',
  'deleteRule': '@request.auth.id != ""',
  'listRule': '',
  'viewRule': '',
});
```

CRUD

- The next step is to create CRUD functions.
- PocketBaseCrudService class contains the CRUD service functions for the Student collection.

PocketBaseCrudService

```
class PocketBaseCrudService {
  // Instance variables instead of static
  final PocketBase _pb;
  final String _collection = 'students';

  // Constructor - this makes it an instance-based class
  PocketBaseCrudService({
    String baseUrl = 'http://127.0.0.1:8090',
  }) : _pb = PocketBase(baseUrl);

  /// Get reference to students collection
  RecordService get _studentsRef =>
    _pb.collection(_collection);
```

```dart
  /// Initialize service and authenticate
Future<void> initialize() async {
  try {
    await pb.health.check();
    // Authenticate if needed
    await pb.collection('users').authWithPassword(
      'admin@example.com', 'password123'
    );
    print('✅ PocketBase service initialized');
  } catch (e) {
    print('❌ PocketBase initialization failed: $e');
    rethrow;
  }
}
}
```

CREATE Operations

```dart
/// CREATE: Add new student to PocketBase
Future<String> createStudent(Student student) async {
  try {
    final record = await pb.collection(_collection).create(
      body: student.toMap()
    );

    print('✅ CREATE: Student added with ID: ${record.id}');
    return record.id;

  } catch (e) {
    print('❌ CREATE Error: $e');
    throw Exception('Failed to create student: $e');
  }
}
```

Bulk Student Creation

```
Future<List<String>> createMultipleStudents(List<Student> tudents) async {
  List<String> createdIds = [];
  try {
    for (Student student in students) {
      final id = await createStudent(student);
      createdIds.add(id);
    }
    print('✅ CREATE: ${students.length} students created');
    return createdIds;
  } catch (e) {
    print('❌ BULK CREATE Error: $e');
    throw Exception('Failed to create students: $e');
  }
}
```

Create: Usage Example

```
// Create single student
Student alice = Student(
  id: '', // Will be auto-generated
  name: 'Alice Johnson',
  age: 20,
  major: 'Computer Science',
  createdAt: DateTime.now(),
);

String studentId =
  await PocketBaseStudentService.createStudent(alice);
```

Generated PocketBase Record:

```
{
  "id": "abc123def456",
  "name": "Alice Johnson",
  "age": 20,
  "major": "Computer Science",
  "createdAt": "2024-01-15T10:30:00.000Z"
}
```

READ Operations

Get All Students with Pagination

```
static Future<List<Student>> getAllStudents({
  int page = 1, int perPage = 20 }) async {
  try {
    final result = await pb.collection(_collection).getList(
      page: page, perPage: perPage,
      sort: '+createdAt',
    );
    List<Student> students = result.items
        .map((record) => Student.fromRecord(record))
        .toList();
    return students;
  } catch (e) {
    throw Exception('Failed to get students: $e');
  }
}
```

Get Single Student

```dart
/// READ: Get specific student by ID
static Future<Student?> getStudentById(String id) async {
  try {
    final record = await pb.collection(_collection).getOne(id);

    Student student = Student.fromRecord(record);
    print('✅ READ: Found student with ID $id');
    return student;

  } catch (e) {
    if (e.toString().contains('404')) {
      print('❌ READ: No student found with ID $id');
      return null;
    }
    print('❌ READ Error: $e');
    throw Exception('Failed to get student: $e');
  }
}
```

Get Students by Major

```
/// READ: Search students by major
static Future<List<Student>> getStudentsByMajor(String major) async {
  try {
    final result = await pb.collection(_collection).getList(
      filter: 'major = "$major"',
      sort: '+name',
    );

    List<Student> students = result.items
        .map((record) => Student.fromRecord(record))
        .toList();

    print('✅ READ: Found ${students.length} $major students');
    return students;
  } catch (e) {
    print('❌ SEARCH Error: $e');
    throw Exception('Failed to search students: $e');
  }
}
```

Get Students by Age

```
Future<List<Student>> getStudentsByAgeRange(int minAge, int maxAge) async {
  try {
    final result = await _studentsRef.getList(
      page: 1,
      perPage: 500,
      filter: 'age >= $minAge && age <= $maxAge',
      sort: '+age',
    );

    List<Student> students =
        result.items.map((record) => Student.fromJson(record.data)).toList();

    print(
        '✅ READ: Retrieved ${students.length} students aged $minAge-$maxAge');
    return students;
  } catch (e) {
    print('❌ READ Error: $e');
    throw Exception('Failed to get students by age range: $e');
  }
}
```

Read: Usage Example

```
// 2-1. READ - Get all students
print('\n2 Reading all students...');
List<Student> allStudents = await pbgetAllStudents();
 for (Student student in allStudents) {
  print('  📖 $student');
}
// 2-2. READ - Get specific student
print('\n3 Reading specific student..');
Student? foundAlice = await pbgetStudentById(aliceId);
 if (foundAlice != null) {
  print('  📖 Found: $foundAlice');
 }
```

```dart
// 3-1. Search by Major
List<Student> students =
  await pb.getStudentsByMajor("Computer Science");
for (Student student in students) {
  print('CS $student');
}
// 3-2. Search by Age Range
List<Student> youngStudents =

  await pb.etStudentsByAgeRange(18, 21);
for (Student student in youngStudents) {
  print('Age 18 - 21  🎓 $student');
}
```

UPDATE Operations

- **Update Specific Fields**

```
/// UPDATE: Update specific student fields
Future<void> updateStudent(String id,
                Map<String, dynamic> updates) async {
  try {
    await pb.collection(_collection).update(id, body: updates);
    print('✅ UPDATE: Student $id updated successfully');
  } catch (e) {
    print('❌ UPDATE Error: $e');
    throw Exception('Failed to update student: $e');
  }
}
```

- **Update Entire Student**
- From the Student object, get the id to update the whole data.

```
/// UPDATE: Replace entire student record
Future<void> updateEntireStudent(Student student) async {
  try {
    await pb.collection(_collection).update(
      student.id,
      body: student.toMap()
    );
    print('✅ UPDATE: Student ${student.id} replaced successfully');
  } catch (e) {
    print('❌ UPDATE Error: $e');
    throw Exception('Failed to update student: $e');
  }
}
```

Update Examples

```
// Update specific fields
await PocketBaseStudentService.updateStudent(storedId, {
  'age': 21,
  'major': 'Data Science'
});

// Update entire record
Student updatedAlice = Student(
  id: 'abc123',
  name: 'Alice Johnson-Smith',
  age: 21,
  major: 'Data Science',
  createdAt: DateTime.now(),
);
await PocketBaseStudentService.updateEntireStudent(updatedAlice);
```

DELETE Operations

Delete Single Student

```
/// DELETE: Remove student by ID
Future<void> deleteStudent(String id) async {
  try {
    await pb.collection(_collection).delete(id);
    print('✅ DELETE: Student $id deleted successfully');

  } catch (e) {
    print('❌ DELETE Error: $e');
    throw Exception('Failed to delete student: $e');
  }
}
```

Delete All Students

```dart
/// DELETE: Remove all students (batch operation)
Future<void> deleteAllStudents() async {
  try {
    int page = 1;
    int totalDeleted = 0;

    while (true) {
      final result = await pb.collection(_collection).getList(
        page: page, perPage: 100
      );
      if (result.items.isEmpty) break;
      for (final record in result.items) {
        await pb.collection(_collection).delete(record.id);
        totalDeleted++;
      }
    }

    print('✅ DELETE: $totalDeleted students deleted successfully');
  } catch (e) {
    print('❌ DELETE ALL Error: $e');
    throw Exception('Failed to delete all students: $e');
  }
}
```

Databases

Feature	PocketBase	Firebase	SQLite	IndexedDB
Type	Server + SQLite	Cloud NoSQL	File-based SQL	Browser NoSQL
Location	Self-hosted	Google Cloud	Local file	Browser storage
Real-time	✅ Built-in	✅ Built-in	❌ None	❌ None
Authentication	✅ Built-in	✅ Complete	❌ Manual	❌ Manual
Scalability	⚠️ Manual	✅ Automatic	❌ Single user	❌ Single user
Queries	✅ REST API	✅ Rich NoSQL	✅ Full SQL	❌ Key-value
Offline	❌ Network only	✅ Smart sync	✅ Always	✅ Always
Cost	🆓 Free hosting	💰 Pay-per-use	🆓 Free	🆓 Free

Decision Framework

> **Choose PocketBase** for: Self-hosted real-time apps, educational projects, MVPs, data control `

- **Choose IndexedDB** for: Browser-only applications, offline-first web apps, client-side caching

- **Choose SQLite** for: Single-user apps, offline-first, embedded applications
- **Choose Firebase** for: Global scale, automatic scaling, rapid development without hosting

PocketBase Limitations

Scalability Limitations:

- Single server architecture
- Manual scaling required
- SQLite performance limits
- No automatic load balancing

Feature Limitations:

- No complex JOINs in filters
- Limited aggregation functions
- No server-side transactions
- Basic search capabilities

Operational Considerations:

- Requires server management
- Manual backup strategies
- Security configuration needed
- Monitoring and maintenance overhead