PocketBase with Dart

Open-Source Real-time Database
for Modern Applications

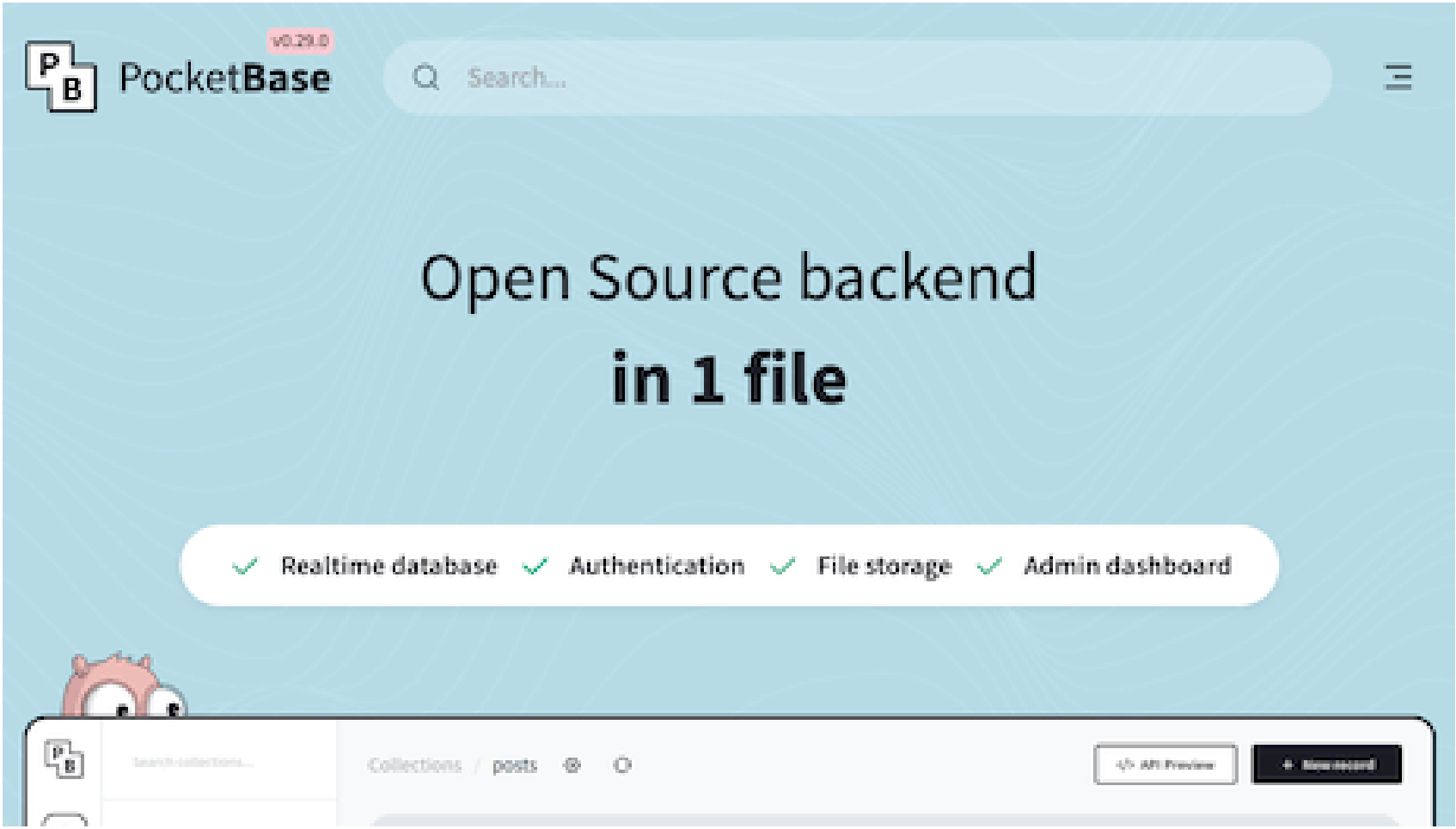
What is PocketBase?

- **Open-source backend** in a single executable file written in Go
- **Real-time SQLite database** with REST API
- **Built-in authentication** and authorization
- **Perfect for rapid prototyping** and small to medium apps
- **Zero configuration** - runs immediately
- **Admin dashboard** included

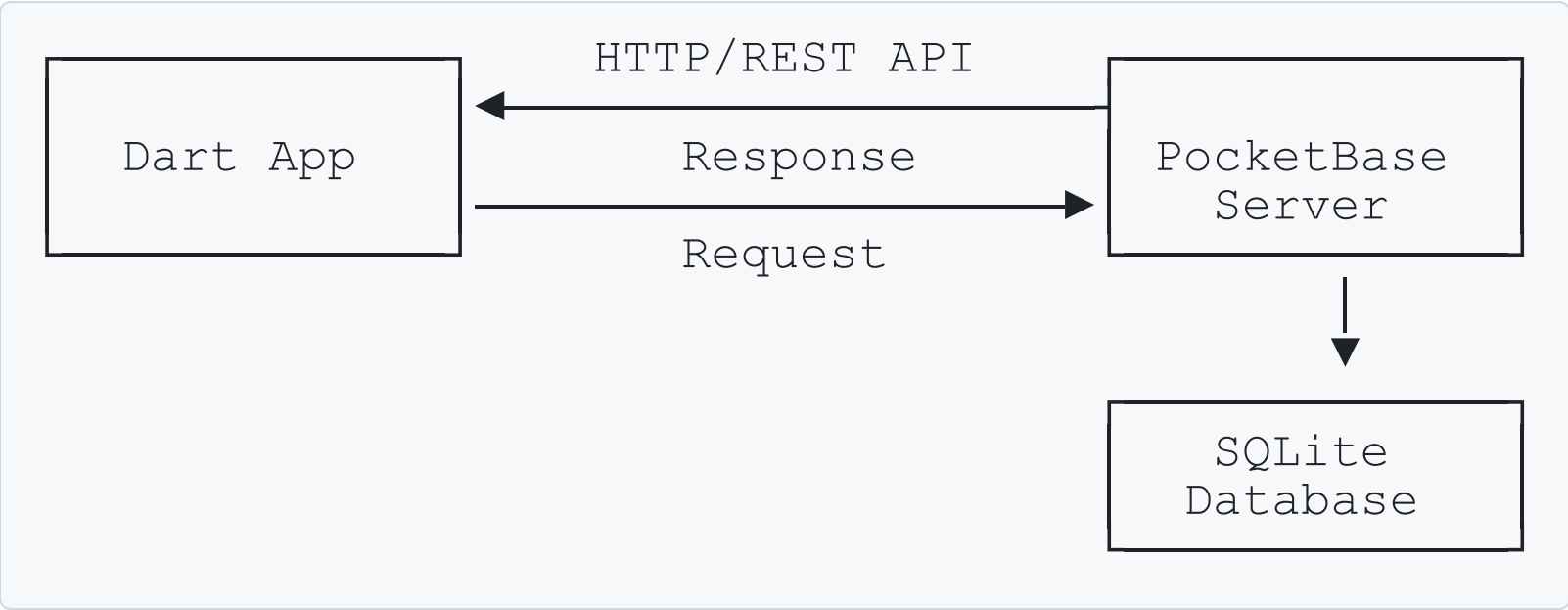
PocketBase Features

- SQLite database with REST API
- Real-time subscriptions
- File storage
- Admin dashboard
- Email templates
- OAuth2 providers

Used by: Developers for startups, personal projects, educational apps


PocketBase
v0.29.0
Search...
Open Source backend
in 1 file
Realtime database
Authentication
File storage
Admin dashboard
Collections / posts
API Preview
New record

Architecture Overview



- Three-tier architecture: Client - Backend - Database
  - Client (Dart/Flutter)
  - Backend (Go executable PocketBase)
  - Database (Single file SQLite)

Project Setup

```
# pubspec.yaml
dependencies:
  # PocketBase client
  pocketbase: ^0.23.0

  # Utility packages
  uuid: ^4.5.1
  intl: ^0.20.2

dev_dependencies:
  lints: ^6.0.0
  test: ^1.21.0
```

- Use Pocketbase_Quick_Start_Guide

- Run `dart pub get` to get all the dependencies installed.
- Run `dart pub outdated` to find newer verions incompatible with dependency constraints.
- To update these dependencies, edit pubspec.yaml, or run `dart pub upgrade --major-versions`.

Code Skeleton

```dart
import 'package:pocketbase/pocketbase.dart';

// Connect to PocketBase server
final pb = PocketBase('http://127.0.0.1:8090');
```

- Initialize Client

Authentication

```
// Test authentication
await pb.collection('users').uthWithPassword(
  'user@example.com',
  'password'
);
print('✅ Authentication successful');
```

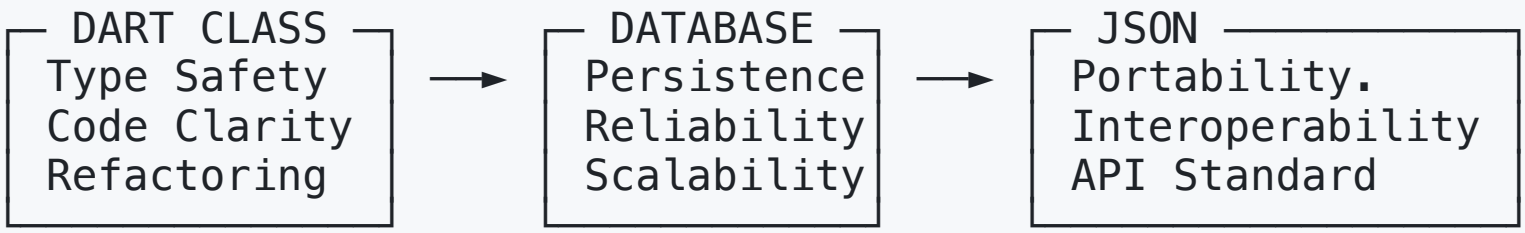
Test Connection

```
// Test health check
await pb.health.check();
print('✅ PocketBase server is running');
```

Data Model Overview

- A **data model** is essential for all software architecture and design.
- The student data model is defined as a **Dart class** to ensure strong typing and seamless integration with the application.

- This model should be **persisted in a database** for reliable data storage and retrieval.
- **JSON format** is used for API communication, enabling data exchange between the app and external services.



Data model: foobar

- When we design database, we should design how the information is represented in JSON, DATABASE, and Dart Class.
- This is our **data model**.
- Examples are in the `foobar/` directory.

```
foo: String
bar: int
```

JSON

- We represent the data model in JSON as follows.

```
{ "foo": "val1", "bar": 323 }
```

- In Dart, this is represented as a `Map<String, dynamic>` datatype.

```
Map<Stirng, dynamic> json = {
  "foo": "val1", "bar": 323
}
```

- We can have multiple JSON objects in a list.

```
[
    { "foo": "val1", "bar": 123 },
    { "foo": "val2", "bar": 223 },
    { "foo": "val3", "bar": 323 }
]
```

Pocketbase RecordModel

- RecordModel is the representation of information in Pocketbase.

```
RecordModel {
  // Automatically generated Unique identifier
  id: "abc123def456",
  // Your actual data
  data: {
    "foo": "randomstring",
    "bar": 42
  },
}
```

RecordModel and JSON

- We can directly access pocketbase collection reference.
- Using the reference, we can retrieve the RecordModel data.
- The data field of the RecordModel is the JSON data structure.

```
RecordService get _foobarRef =>
    _pb.collection('foobar');
final record = await _studentsRef.getOne(id);
final json = record.data;
```

Dart Class

```dart
class FooBar {
  String foo;
  int bar;
  // constructor
  FooBar({
    required this.foo,
    required this.bar,
  });
```

Service function for JSON conversion

```
factory FooBar.fromJson(Map<String, dynamic> json) {
  return FooBar(
    foo: json['foo'] as String,
    bar: json['bar'] as int,
  );
}
Map<String, dynamic> toJson() => {
  'foo': foo,
  'bar': bar,
};
```

- The elements in the record RecordModel can be empty, so we use the `record.data['name'] as String? ?? ''` expression to set the default value.
- The expression is the same as the following code.

```
value = '';
if (record.data['foo'] != null) {
  value = record.data['foo'];
}

record.data['name'] as String? ?? ''
```

- So, we use the following to safely convert from JSON to the FooBar object.
- Remember that JSON is the same as record.data.

```
factory FooBar.fromJson(Map<String, dynamic> json) {
  return FooBar(
    foo: json['foo'] as String? ?? '',
    bar: json['bar'] as int ?? 0,
  );
}
```

Service function for Object comparison

- It is the same as Java's `equals`.

```
@override
bool operator ==(Object other) {
  if (identical(this, other)) return true;
  return other is FooBar
    && other.foo == foo
    && other.bar == bar;
}

@override
int get hashCode => foo.hashCode ^ bar.hashCode;
```

Run foobar_json_test.dart

- Run `dart test`

```
pocketbase> dart test test/foobar/foobar_json_test.dart
00:00 +11: All tests passed!
```

- Or run `dart`

```
smcho@mac pocketbase> dart test/foobar/foobar_json_test.dart
00:00 +0: FooBar Constructor Tests should create FooBar with required parameters
00:00 +1: FooBar Constructor Tests should create FooBar with different values
00:00 +2: FooBar JSON Serialization Tests should convert FooBar to JSON correctly

...

00:00 +9: FooBar Edge Cases should throw when JSON has wrong types
00:00 +10: FooBar Edge Cases should throw when JSON is missing required fields
00:00 +11: All tests passed!
```

Pocketbase Collection

- We need to create a collection to make a RecordModel.
- We created a collection using API.

```
curl -X POST http://localhost:8090/api/collections \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer YOUR_ADMIN_TOKEN" \
  -d '{"name": "students",  "type": "base",
    "schema": [{"name": "name", "type": "text", "required": true,
        "options": {
          "min": 1,
          "max": 100
        }
```

Programmatic Collection Setup

- We can create the collection programatically.

```
final collection = await pb.collections.create(body: {
  'name': 'foobar',
  'type': 'base',
  'fields': [
    {'name': 'foo', 'type': 'text', 'required': true},
    {'name': 'bar', 'type': 'number', 'required': true},
  ],
  'createRule':
      '@request.auth.id != ""', // Only authenticated users can create
  'updateRule':
      '@request.auth.id != ""', // Only authenticated users can update
  'deleteRule':
      '@request.auth.id != ""', // Only authenticated users can delete
  'listRule': '', // Anyone can list records
  'viewRule': '', // Anyone can view records
});
```

Why Do I Need Rules in PocketBase?

- Access rules control **who can access or change your data**.
- If you **skip them**, your API may be **completely locked down**.

What Are Rules?

Each collection can have:

Rule	Controls
`createRule`	Who can create new records
`updateRule`	Who can update existing records
`deleteRule`	Who can delete records
`listRule`	Who can list all records
`viewRule`	Who can view a single record

What Happens If You Skip Rules?

Rule	Default Behavior
`createRule`	❌ No one can create
`updateRule`	❌ No one can update
`deleteRule`	❌ No one can delete
`listRule`	❌ No one can list
`viewRule`	❌ No one can view

Result: Your collection is inaccessible!

Examples of FooBar

- Public Read, Authenticated Write

```
{
  createRule: '@request.auth.id != ""',
  updateRule: '@request.auth.id != ""',
  deleteRule: '@request.auth.id != ""',
  listRule: '',
  viewRule: ''
}
```

Real-world equivalent: Database table creation with permissions

Usage

```dart
  // Example: Create a record with direct field access
  final record =
      await pb.collection('foobar').create(
        body: {'foo': 'val1', 'bar': 323});

  // Access fields directly
  print('id  value: ${record.data['id']}'); // Automatically assigned
  print('foo value: ${record.data['foo']}'); // Output: val1
  print('bar value: ${record.data['bar']}'); // Output: 323
```

```
RecordModel {
  data: {
    id: "abc123def456",
    "foo": "randomstring",
    "bar": 42
  },
}
```

The CRUD Pattern

Universal Database Operations

Operation	Purpose	HTTP Method	SQL Equivalent
Create	Add new data	`POST`	`INSERT`
Read	Retrieve data	`GET`	`SELECT`
Update	Modify existing	`PUT/PATCH`	`UPDATE`
Delete	Remove data	`DELETE`	`DELETE`

Key Insight: This pattern works with **any database**

- SQL: MySQL, PostgreSQL
- NoSQL: MongoDB, Firebase
- Same concepts, different implementation
- **Abstract thinking** for better software design

CREATE

- Add a new FooBar record to the database
  - Returns the created FooBar with its generated ID

```
Future<FooBar> create(FooBar foobar) async {
  try {
    // Convert FooBar object to JSON and send to PocketBase
    final record = await _pb.collection(_collectionName).create(
          body: foobar.toJson(),
        );
    // Convert the returned record back to FooBar object
    return FooBar.fromJson(record.data);
  } catch (e) {
    throw Exception('Failed to create FooBar: $e');
  }
}
```

READ Operations

Fetching One Data from ID

```
Future<FooBar> getById(String id) async {
  try {
    // Fetch record from PocketBase using the ID
    final record = await _pb.collection(_collectionName).getOne(id);

    // Convert record data to FooBar object
    return FooBar.fromJson(record.data);
  } catch (e) {
    throw Exception('Failed to get FooBar with ID $id: $e');
  }
}
```

Fetching All Data

```
Future<List<FooBar>> getAll() async {
  try {
    // Fetch all records from the collection
    final resultList = await _pb.
      collection(_collectionName)
      .getFullList();

    // Convert each record to FooBar object
    // and return as list
    return resultList.map((record) =>
      FooBar.fromJson(record.data)).toList();
  } catch (e) {
    throw Exception('Failed to get all FooBar records: $e');
  }
}
```

Fetching Data with Pagination

```
Future<List<FooBar>> getRecord(
  [int page = 1, int perPage = 5]) async {
  try {
    // Get paginated list (ResultList wrapper object)
    final resultList = await _pb
        .collection(_collectionName)
        .getList(page: page, perPage: perPage);

    return resultList.items
        .map((record) => FooBar.fromJson(record.data))
        .toList();
  } catch (e) {
    throw Exception('Failed to get all FooBar records: $e');
  }
}
```

Read the next 5 pages

```
getRecord([int page = 5+1, int perPage = 5])
```

Pagination Benefits

- **Memory efficiency** - Don't load all data at once
- **Better user experience** - Faster loading
- **Scalability** - Works with millions of records

Search FooBar records with filtering

```
// Example usage:
//  searchByFoo("hello") returns
//  all recordswhere foo contains "hello"
Future<List<FooBar>> searchByFoo(String searchTerm) async {
  try {
    // Use PocketBase filter syntax to search
    final resultList = await _pb.collection(_collectionName)
      .getFullList(
        filter: 'foo ~ "$searchTerm"', // ~ means "contains"
      );
    return resultList.map((record) =>
      FooBar.fromJson(record.data)).toList();
  } catch (e) {
    throw Exception('Failed to search FooBar records: $e');
  }
}
```

UPDATE Operations

Modifying Existing Records

```
Future<FooBar> update(String id, FooBar updatedFoobar) async {
  try {
    // Send updated data to PocketBase
    final record = await _pb.collection(_collectionName).update(
          id,
          body: updatedFoobar.toJson(),
        );

    // Return the updated FooBar object
    return FooBar.fromJson(record.data);
  } catch (e) {
    throw Exception('Failed to update FooBar with ID $id: $e');
  }
}
```

UPDATE Operations - Best Practices

1. Defensive Programming

```
// Always check if fields exist before updating
if (updatedFoobar.toJson().containsKey('foo')) {
  // Safe to access nested data
}
```

2. Partial Updates

```
// Only update specific fields, not entire record
var curretFooBar = currentFooBar.toJson();
var currentBar = currentFooBar['bar'];
updateData = {
  {'foo': 'P-$currentFoo', 'bar': currentBar},
};
// PocketBase merges with existing data
```

3. Error Handling

```
try {
  final updated = await pb.collection('foobar').update(id, body: updateData);
} catch (err) {
  // Handle network errors, validation errors, etc.
  print('Update failed: $err');
}
```

DELETE Operations

1. Single Record Deletion

```
Future<bool> delete(String id) async {
  try {
    // Delete the record from PocketBase
    await _pb.collection(_collectionName).delete(id);
    return true;
  } catch (e) {
    throw Exception('Failed to delete FooBar with ID $id: $e');
  }
}
```

2. Batch Deletion with Pagination

```dart
Future<void> deleteAllRecords() async {
  int page = 1;
  const int perPage = 10;
  int totalDeleted = 0;

  while (true) {
    final result = await pb.collection('records')
        .getList(page: page, perPage: perPage);

    if (result.items.isEmpty) break;

    for (final item in result.items) {
      await pb.collection('records').delete(item.id);
      totalDeleted++;
    }
  }
  print('Total records deleted: $totalDeleted');
}
```

Foobar Utility

- We started with the Data model: foobar.dart.
- We provided the CRUD service: foobar_crud.dart.
- We created unittests for each module.

- The next step is to provide the utility service.
  - Read the source: `foobar/lib/foobar_utility.dart`
  - Read the tests:
    `foobar/test/foobar_utility_test.dart`
  - Read the doc: `foobar/test/foobar_utility.md`