

# CMPS 312



# Supabase Cloud Services



**Database**



**Authentication**



**Storage**

**Dr. Abdelkarim Erradi**

**CSE@QU**

# Outline

1. Supabase Database
2. CRUD Operations
3. File Storage
4. Authentication
5. Access Image Gallery and Camera

# Backend-as-Service (BaaS)

- **Purpose:** Provide ready-made backend for web & mobile apps
- **Benefits:**
  - No need to build/manage servers, databases, or APIs
  - Speeds up development and reduces infrastructure complexity & cost
  - Allows developers focus on frontend and core business logic
- **Common Features:** User authentication, Managed databases, File storage, Serverless functions, Notifications & analytics
- **Examples:** Supabase Firebase AWS Amplify



# What is Supabase?

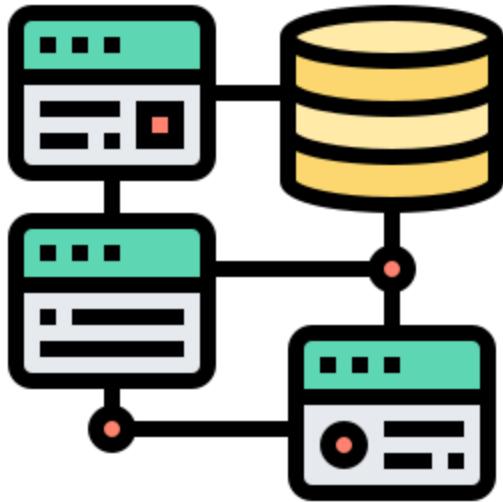
Supabase = Open-source Backend-as-a-Service (BaaS) built on PostgreSQL. Core Features:

- Database: Managed PostgreSQL with Row-Level Security (RLS)
- Authentication: Secure user sign-in via email/password & OAuth providers
- Storage: Scalable file storage with public/signed URLs
- Realtime: Broadcast database updates in real time
- Edge Functions: Deploy serverless functions for custom logic
- Developer Tools: SDKs for **Flutter/Dart**, JavaScript, and more

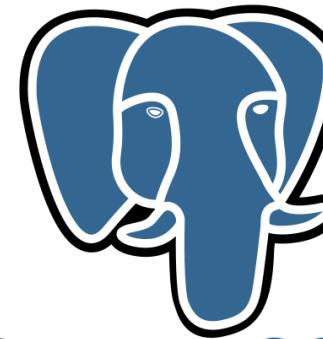
# Getting Started

- Add `supabase_flutter` to `pubspec.yaml`
- Initialize Supabase in `main.dart`
  - This Enables database, authentication, and storage features in your Flutter app.

```
await Supabase.initialize(  
    url: 'https://your-project.supabase.co',  
    anonKey: 'your-anon-key',  
);
```



# Supabase Database



Postgre<sup>SQL</sup>

# Supabase Database

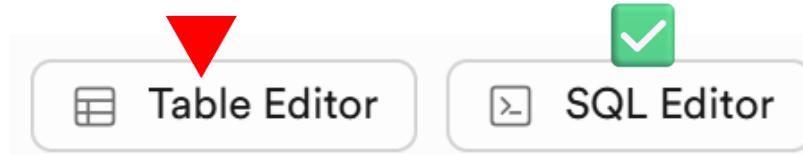
- Managed PostgreSQL: Includes SQL, views, triggers, policies, and functions
- Auto-Generated REST & GraphQL APIs for every table, view and function
  - Use `.from('table')` to work with tables
- Schema First Design: Create tables via SQL or Supabase dashboard
- Row-Level Security (RLS) with customizable policies

```
// Fetching data
final response = await Supabase.instance.client
    .from('profiles')
    .select()
    .execute();
```



# Creating Database Table

- Design tables using SQL scripts or Supabase dashboard (visual editor)



```
-- Example table: todos
create table if not exists todos (
    id uuid primary key default gen_random_uuid(),
    description text not null,
    -- Enforce data integrity with constraints
    type text not null check (type in
        ('personal','work','family')),
    completed boolean not null default false,
    -- Timestamp when todo was created
    created_at timestampz not null default now(),
    -- Link todos to authenticated users
    user_id uuid references auth.users(id)
);
```

# PostgreSQL Common Data Types

- **Numeric**
  - INTEGER – General whole numbers
  - BIGINT – Large IDs or counters
  - NUMERIC(p,s) – Exact precision (money)
  - SERIAL – Auto-increment IDs
- **Character**
  - VARCHAR(n) – Variable-length text
  - TEXT – Large/unlimited text
- **Date/Time**
  - DATE – Calendar dates
  - TIMESTAMPTZ – Date/time with time zone
- **Boolean** – True/False flags
- **Other**
  - UUID – Unique identifiers
  - JSONB – Semi-structured data (queryable)

# Database Auto-Assigned IDs

Two common strategies for primary keys:

- Use PostgreSQL's **SERIAL** for efficient auto-incremented numeric ID

```
CREATE TABLE todos (
    id SERIAL PRIMARY KEY,
    ...);
```

- Use PostgreSQL's **gen\_random\_uuid()** for globally unique IDs

```
create table todos (
    id uuid primary key default gen_random_uuid(),
    ...);
```

# PostgreSQL: One-to-Many Relationships

- One parent row relates to many child rows (e.g., one author has many books)
  - Enforced via foreign key (FK) from child → parent primary key (PK)
  - Use ON DELETE CASCADE to remove child rows when parent is deleted

```
-- Parent table (one)
CREATE TABLE authors (
    author_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL);
```

```
-- Performance: index FK to
speed-up joins & deletes
CREATE INDEX idx_books_author_id
ON books(author_id);
```

```
-- Child table (many)
CREATE TABLE books (
    book_id SERIAL PRIMARY KEY,
    author_id INT NOT NULL,
    title TEXT NOT NULL, ...
    CONSTRAINT fk_books_author
        FOREIGN KEY (author_id)
        REFERENCES authors(author_id)
        ON DELETE CASCADE
);
```

# Row Level Security (RLS)

- RLS: Ensures users can only access and modify their own data
  - You can think of the RLS as automatically inserted WHERE clauses during query and mutation.

-- *Enable RLS on the table*

```
alter table todos enable row level security;
```

-- *Policy: Read own rows. The user\_id is a built-in variable representing the current user*

```
create policy "read own" on todos
  for select using (auth.uid() = user_id);
```

-- *Policy: Modify own rows*

```
create policy "modify own" on todos
  for all using (auth.uid() = user_id);
```

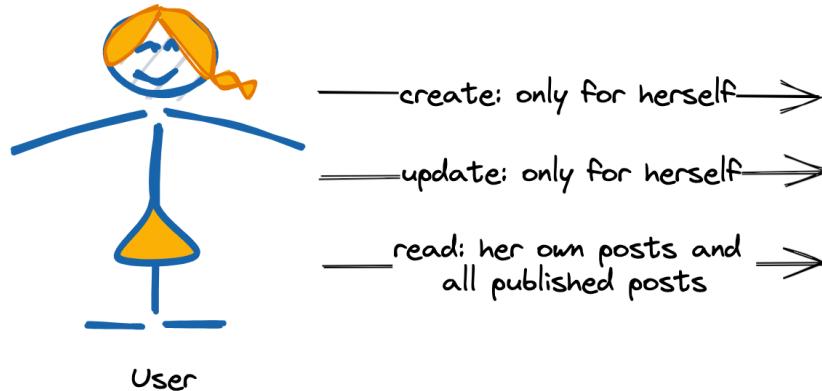
# RSL Syntax

- USING filters **which existing rows** the user may see, modify or delete
- WITH CHECK restricts **which values a user may create or change to**
  - For SELECT: only USING runs
  - For INSERT: only WITH CHECK runs (no existing rows yet)
  - For UPDATE: both may apply (USING on the target row; WITH CHECK on the row after UPDATE)

-- Skeleton

```
CREATE POLICY policy_name
ON schema.table_name
TO { public | authenticated }    -- or other roles
FOR { SELECT | INSERT | UPDATE | DELETE }    -- (or: FOR ALL)
-- SELECT/UPDATE/DELETE gate: what rows the user is allowed to
select/update/delete
USING ( boolean_expression_for_existing_rows ) -- (true: FOR ALL)
-- INSERT/UPDATE gate: checks if the new/updated row complies with
the policy expression
WITH CHECK ( boolean_expression_for_new_or_updated_rows );
```

# RLS Examples



## -- 🏃 Read Policy

```
-- allows anyone to read published posts  
-- or the author to read their own (including drafts)
```

```
create policy "read published or own"
```

```
on posts
```

```
for select to authenticated, public  
using (published = true OR author_id = auth.uid());
```

## -- 🖌 Insert/Update/Delete Policy

```
-- Owner manage writes (INSERT/UPDATE/DELETE)
```

```
create policy "owner writes"
```

```
on posts
```

```
for all to authenticated
```

```
-- SELECT/UPDATE/DELETE gate: what rows the user is allowed to select/update/delete
```

```
using (author_id = auth.uid())
```

```
-- Checks if the new/updated row complies with the policy expression (i.e., author_id  
must be current user id. Users cannot change the author_id or add for another author)
```

```
with check (author_id = auth.uid());
```



# CRUD Operations



CREATE



READ



UPDATE



DELETE

---

C R U D

# CRUD

## CRUD

- Create: Add new records
- Read: Retrieve existing records
- Update: Modify existing records
- Delete: Remove records
- In Supabase perform CRUD via the auto-generated REST APIs

```
// Create
await Supabase.instance.client
  .from('tasks')
  .insert({ 'title': 'New Task' })
  .execute();

// Update
await Supabase.instance.client
  .from('tasks')
  .update({ 'done': true })
  .eq('id', 1)
  .execute();

// Delete
await Supabase.instance.client
  .from('tasks')
  .delete()
  .eq('id', 1)
  .execute();
```



# Database CRUD Operations

```
final client = Supabase.instance.client;
// CREATE
Future<void> addTodo(Todo todo) async {
  await client.from('todos').insert(todo.toJson());
}
// READ (list)
Future<List<Todo>> getTodos() async {
  final data = await client.from('todos').select().order('created_at', ascending: false);
  return (data as List).map((j) => Todo.fromJson(j)).toList();
}
// READ (single)
Future<Todo?> getTodoById(String id) async {
  final json = await client.from('todos').select().eq('id', id).maybeSingle();
  return json == null ? null : Todo.fromJson(json);
}
// UPDATE
Future<void> updateTodo(Todo todo) async {
  await client.from('todos').update(todo.toJson()).eq('id', todo.id);
}
// DELETE
Future<void> deleteTodo(String id) async {
  await client.from('todos').delete().eq('id', id);
}
// COUNT
Future<int> getTodosCount() async {
  final res = await client.from('todos').select().count(CountOption.exact);
  return res.count;
}
```

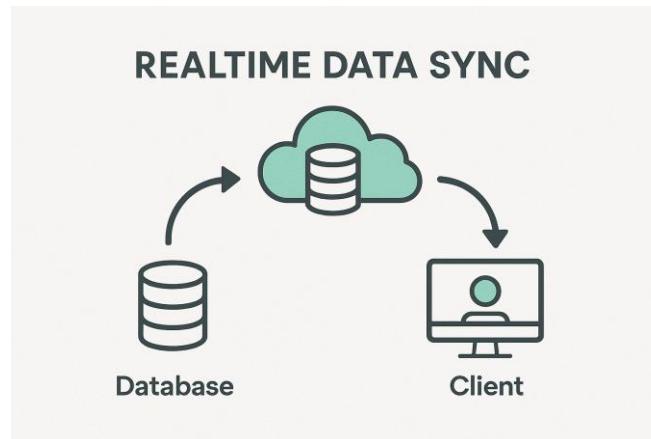
# Best Practices

- Implement data access in repositories
- Expose SupabaseClient as a provider
- Paginate with range() for large lists
- Use select() projections to limit payload
- Always handle errors (try/catch) and show user-friendly messages



# Real-Time Data Updates

- **What It Does:** Streams database changes instantly to connected clients
  - Enables **instant updates** without manual refresh, improving user engagement
- **Key Use Cases:** Live chat applications, Real-time dashboards, Multiplayer games
- **Best Practices:**
  - Subscribe only to relevant tables/rows to optimize performance
  - Integrate with state management for better user experience (UX)





# Listen to Database Realtime Updates

Postgres Stream:

```
.from('table').stream(primaryKey: ['id'])
```

Stream for ToDo list

```
Stream<List<Todo>> observeTodos() {  
    final client = Supabase.instance.client;  
    return client  
        .from('todos')  
        .stream(primaryKey: ['id'])  
        .order('created_at', ascending: false)  
        .map((rows) =>  
            rows.map(Todo.fromJson).toList());  
}
```

# File Storage





# File Storage

## What It Does:

- Upload, manage, and serve files securely

## Key Features:

- Has a simple bucket/folder/file structure
- Upload/download user files or images
- Create storage buckets via Supabase dashboard
- Define file access rules (public, private, signed URLs)

## Common Use Cases:

- Store Profile pictures, Documents, Images

## Best Practices

- Use UUID file names to avoid collisions
- Keep buckets private and use signed URLs where possible



# File Upload

```
final storage = Supabase.instance.client.storage;
// Upload an avatar using a file path
Future<String> uploadAvatarFromPath(String filePath, String userId) async {
  final file = File(filePath);
  final fileName =
    'avatars/$userId-${DateTime.now().millisecondsSinceEpoch}.png';
  await storage.from('avatars').upload(fileName, file,
    fileOptions: const FileOptions(contentType: 'image/png'),
  );
// If bucket is public → returns public URL
return storage.from('avatars').getPublicUrl(fileName);
}
```

```
// Signed URL for private buckets
Future<Uri> getSignedUrl(String path,
  {Duration ttl = const Duration(minutes: 5)}) async {
  final signedUrl = await storage
    .from('avatars')
    .createSignedUrl(path, ttl.inSeconds);
  return Uri.parse(signedUrl);
}
```

# List files in a bucket

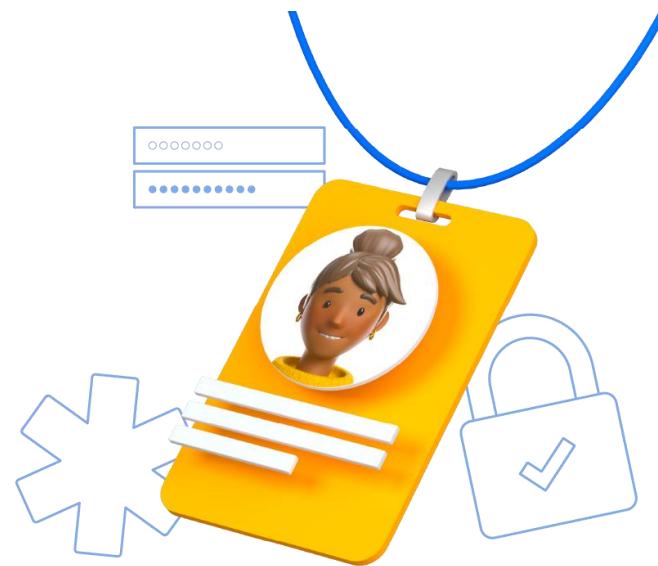
- Get URLs of files in particular subfolder

```
Future<List<String>> getImageUrls() async {
    final storage = Supabase.instance.client.storage;
    final files = await storage.from('images').list(path: '');
    return files.map((f) =>
        storage.from('images').getPublicUrl(f.name)).toList();
}
```

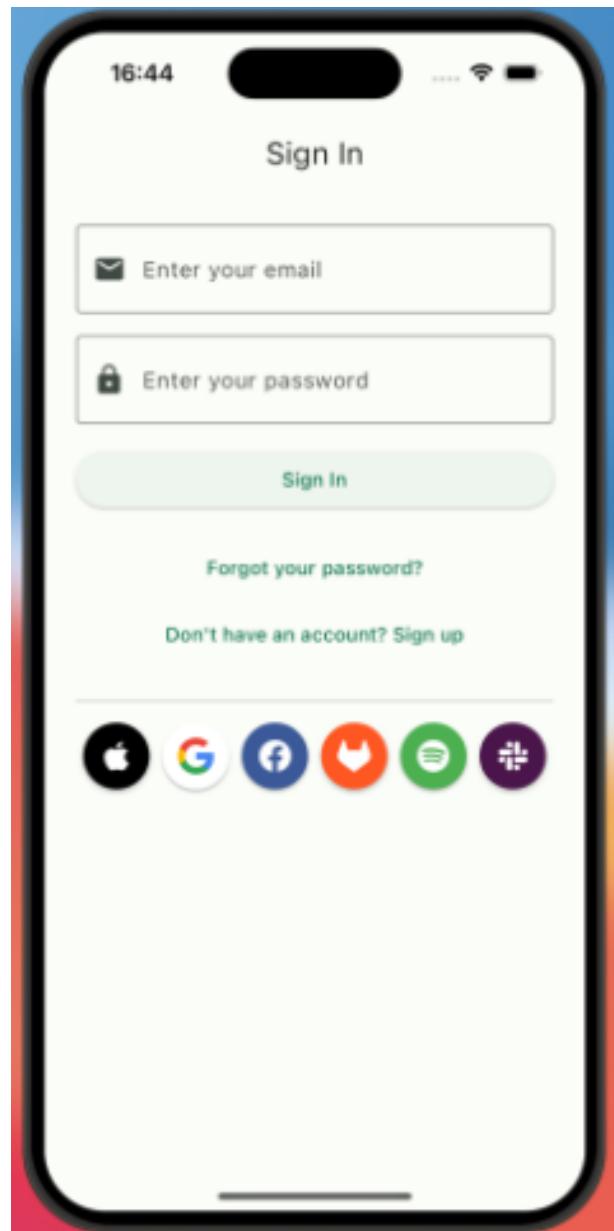
 If the bucket is **private**, use signed URLs instead:

```
Future<List<String>> getImageUrls() async {
    final storage = Supabase.instance.client.storage;
    final files = await storage.from('images').list(path: '');
    return Future.wait(files.map((f) =>
        storage.from('images').createSignedUrl(f.name, 3600)));
}
```

# Authentication



# Authentication

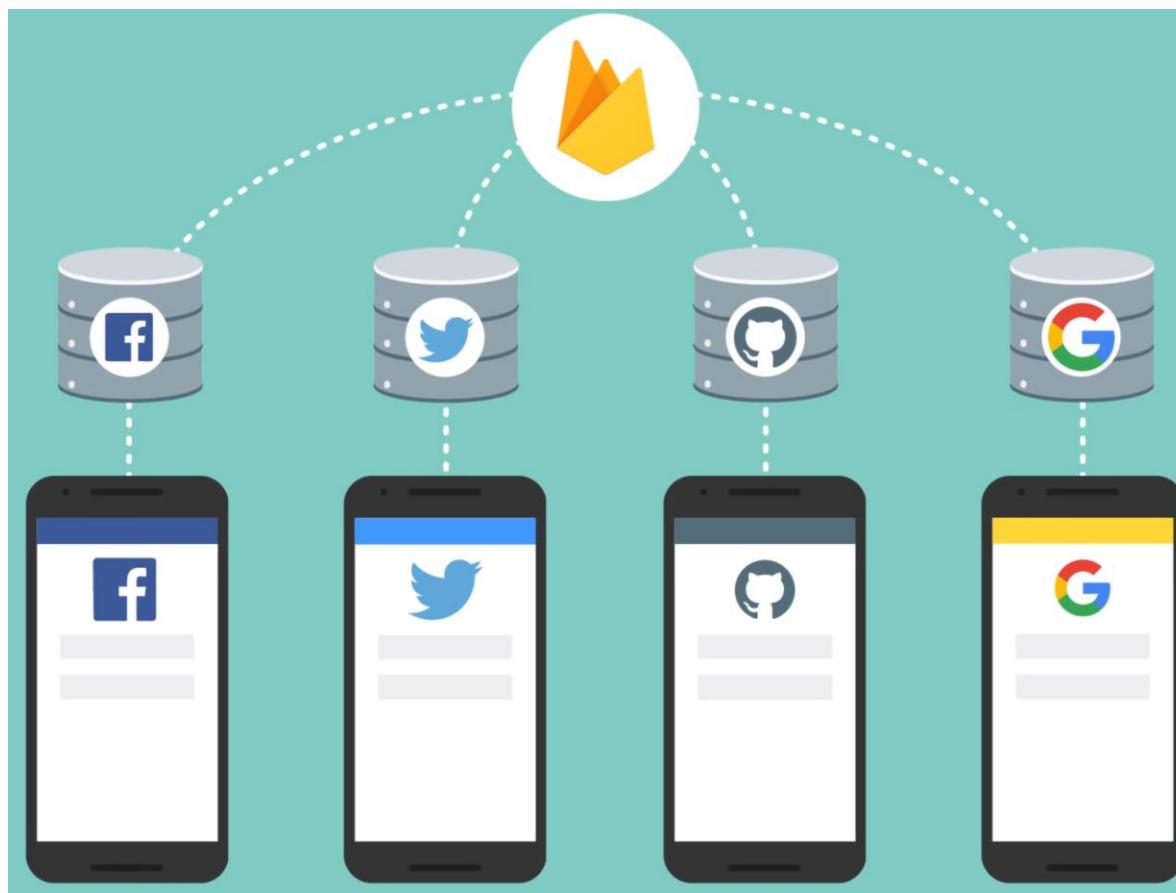


- **Authentication = Identity verification:**
  - Verify the identity of the user given the credentials received
  - Making sure the user is who he claims to be
- Every user gets a unique ID
- Restrict who can read and write what data



# Authentication

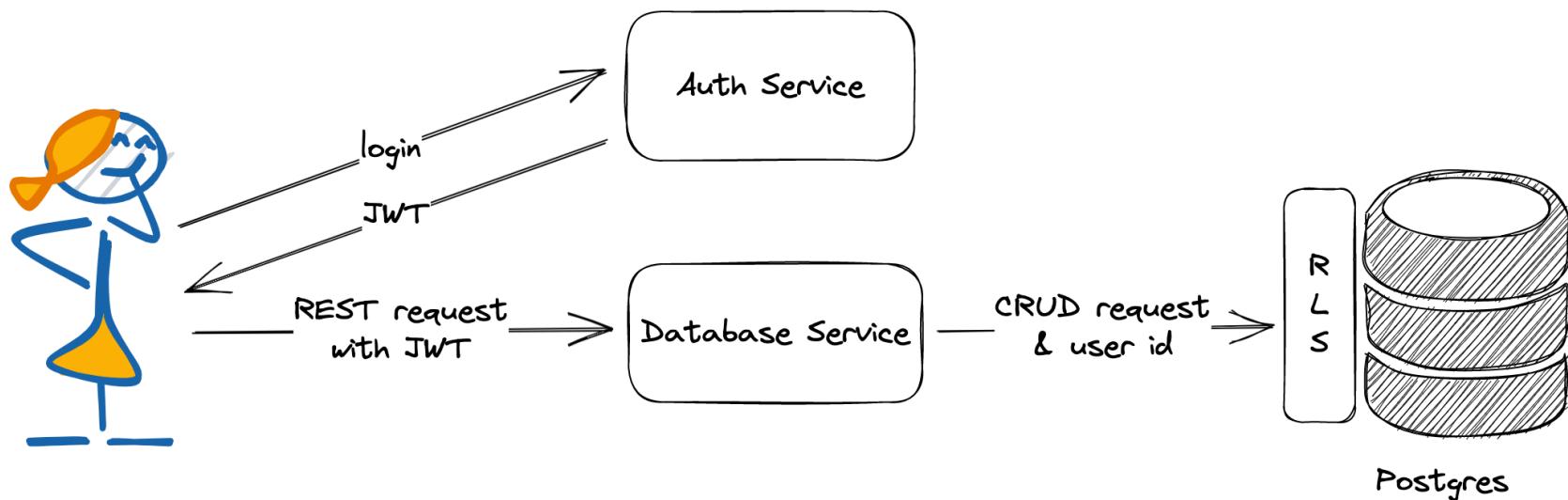
- Email/password, OTP/magic links, and Auth providers (Google, Apple, etc.)
  - User sign-up, login, session management, password reset
  - Session-based auth with refresh tokens



Multiple Identity Providers can be used for Authentication

# Authentication and Authorization flow

- **Login:** Client sends credentials → Auth Service returns a **JWT (JSON Web Token)**
- **Requests:** Client includes JWT in **REST API calls** to Database Service
- **Database:** Service validates JWT, extracts **user ID**, performs **CRUD** on PostgreSQL
- **RLS:** PostgreSQL applies **Row-Level Security** using user ID to restrict access to authorized rows





# Authentication

```
final auth = Supabase.instance.client.auth;

// Sign up
Future<void> signUp(String email, String password) async {
  await auth.signUp(email: email, password: password);
}

// Sign in
Future<void> signIn(String email, String password) async {
  await auth.signInWithEmailAndPassword(email: email, password:
password);
}

// Sign out
Future<void> signOut() async {
  await auth.signOut();
}
```

# Sign up and save Profile data

```
Future<User?> signUp(User user) async {
  try {

    final auth = supabase.auth;
    // ----- 1) Sign up -----
    final response = await auth.signUp(
      email: user.email,
      password: user.password,
      data: {
        // Optional metadata stored inside auth.users
        'firstName': user.firstName,
        'lastName': user.lastName,
      },
    );

    // ----- 2) Save profile -----
    await supabase.from('profiles').upsert({
      'id': authUser.id,
      'first_name': user.firstName,
      'last_name': user.lastName,
      'avatar_url': 'http://test.com/spongebob.png',
    });
    return response.user;
  } catch (e) {
    print('Error during sign up: $e');
    return null;
  }
}
```

```
create table profiles (
  id uuid primary key references
    auth.users(id) on delete cascade,
  first_name text,
  last_name text,
  avatar_url text,
  created_at timestamp default now()
);
```

# Get current user details

- Anywhere in the app you can access the details of current user

```
void getCurrentUser() {  
    User? user = supabase.auth.currentUser;  
    if (user != null) {  
        print('User is signed in! Id: ${user.id}');  
        print('User is signed in! Email: ${user.email}');  
        print('Metadata: ${user.userMetadata}');  
    } else {  
        print('No user is signed in.');  
    }  
}
```

# Listen to auth state

- Real-time Updates: If you need to react to authentication state changes (e.g., a user logs in or out), you should listen to the **onAuthStateChange** stream provided by Supabase Auth

```
supabase.auth.onAuthStateChange.listen((data) {  
    final AuthChangeEvent event = data.event;  
    final Session? session = data.session;  
  
    if (event == AuthChangeEvent.signedIn) {  
        print('User signed in: ${session?.user?.email}');  
    } else if (event == AuthChangeEvent.signedOut) {  
        print('User signed out');  
    }  
    // Handle other events like AuthChangeEvent.userUpdated, etc.  
});
```

# Route Auth Guard

- Auth Guard (GoRouter + Riverpod)
- Use guards tied to auth state

```
final authStateProvider = StreamProvider((ref) {  
  return Supabase.instance.client.auth.onAuthStateChanged  
    .map((e) => e.session);  
});  
  
final authGuard = GoRoute(  
  path: '/account',  
  builder: (context, state) => const AccountScreen(),  
  redirect: (context, state) {  
    final session = context.read(authStateProvider).maybeWhen(  
      data: (s) => s,  
      orElse: () => null,  
    );  
    return session == null ? '/signin' : null;  
  },  
);
```



# Architecture & Patterns

- Use Riverpod providers to expose repositories

```
final supabaseClientProvider = Provider((ref) =>  
  Supabase.instance.client);
```

```
final todoRepositoryProvider = Provider((ref) {  
  final client = ref.watch(supabaseClientProvider);  
  return TodoRepository(client);  
});
```



# Best Practices for Supabase

## Security

- Enable **Row-Level Security (RLS)** on all tables
- Apply **least-privilege policies**
- Use **signed URLs** for private assets
- Set **short TTLs** for sensitive files

## Performance

- Use **projections** and **pagination**; avoid `SELECT *` in production
- Batch UI updates
- Keep **migration SQL** under version control
- Use **Edge Functions** for server-side logic

# Access Image Gallery and Camera



# Access Image Gallery and Camera

- Using `image_picker` package for picking images from the image gallery or taking new pictures with the camera

```
Future<File?> pickImage(ImageSource source) async {
    final imagePicker = ImagePicker();
    final pickedImage = await imagePicker.pickImage(
        source: source, // camera or gallery
        maxWidth: double.infinity,
    );
    if (pickedImage == null) return null;
    return File(pickedImage.path);
}
```

# image\_picker methods

```
final ImagePicker picker = ImagePicker();
// Pick an image
final XFile? image = await picker.pickImage(source: ImageSource.gallery);
// Capture a photo
final XFile? photo = await picker.pickImage(source: ImageSource.camera);
// Pick a video
final XFile? galleryVideo =
    await picker.pickVideo(source: ImageSource.gallery);
// Capture a video
final XFile? cameraVideo = await picker.pickVideo(source: ImageSource.camera);
// Pick multiple images
final List<XFile> images = await picker.pickMultiImage();
// Pick singe image or video
final XFile? media = await picker.pickMedia();
// Pick multiple images and videos
final List<XFile> medias = await picker.pickMultipleMedia();
```

# Summary

- **Database:** Store and query app data using tables with defined relationships
- **Authentication:** Built-in backend services for user sign-up and login
  - Supports **email/password** and **Auth providers** (e.g., Google)
- **File Storage:** Securely upload, store, and retrieve files
- **Security:** Protect user data with authentication and authorization policies



# References

- Supabase Flutter Docs:

<https://supabase.com/docs/guides/getting-started/quickstarts/flutter>

- Realtime:

<https://supabase.com/docs/guides/realtime>

- Storage:

<https://supabase.com/docs/guides/storage>

- Auth: <https://supabase.com/docs/guides/auth>

- RLS Policies:

<https://supabase.com/docs/guides/auth/row-level-security>