CMPS 312

Supabase Database

Dr. Abdelkarim Erradi

CSE@QU

Outline

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
- Examples: **Supabase**, Firebase, AWS Amplify

What is Supabase?

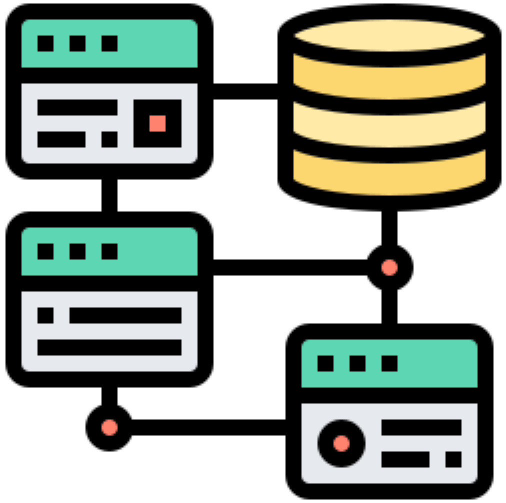
Supabase = Open-source **Backend-as-a-Service** (BaaS) built on PostgreSQL. Core Features:

- 🗄️ **Database**: Managed PostgreSQL with Row-Level Security (RLS)
- ☁️ **Storage**: Scalable file storage with public/signed URLs
- 🔐 **Authentication**: Secure user sign-in via email/password & OAuth providers
- 🔔 **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',
);
```



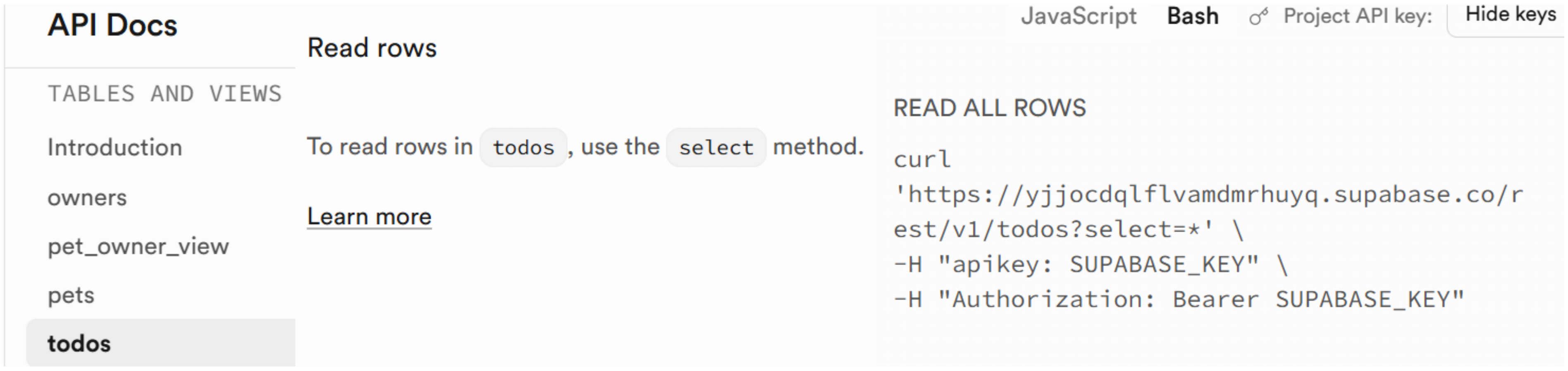
Supabase Database

PostgreSQL

< Back

Supabase Database

- **Managed PostgreSQL**: Includes tables, views, triggers, and functions
- **Auto-Generated Web APIs** for every table, view and function
  - Use `.from('table')` to read/write from/to tables



Read rows

To read rows in `todos`, use the `select` method.

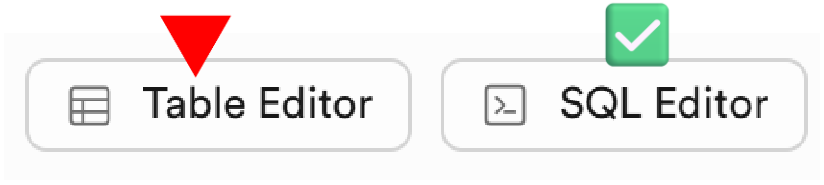
READ ALL ROWS

```
curl
'https://yjjocdqlflvamdmrhuyq.supabase.co/rest/v1/todos?select=*' \
-H "apikey: SUPABASE_KEY" \
-H "Authorization: Bearer SUPABASE_KEY"
```

- **Schema First Design**: Create tables via SQL or Supabase dashboard
- **Row-Level Security (RLS)** with customizable policies

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 timestamptz 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

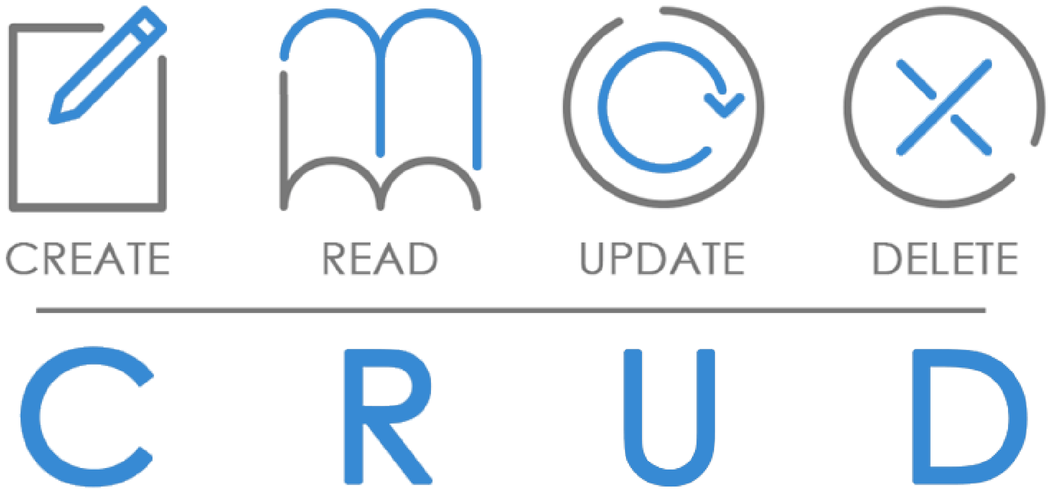
- One parent row relates to many child rows (e.g., one author has many books)
  - Enforced via foreign key (FK) from child → parent primary key (PK)
  - Use ON DELETE CASCADE to remove child rows when parent is deleted

```
-- 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
);
```

CRUD Operations



< Back

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

```dart
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;
}
```

Common Filter Methods and Examples

Method	Description	Example
eq	Exact match	`client.from('cities').select().eq('name', 'Doha');`
neq	Not equal	`client.from('cities').select().neq('country_id', 1);`
lt / lte	Less than / Less or equal	`client.from('cities').select().lt('population', 1000000);`
gt / gte	Greater than / Greater or equal	`client.from('cities').select().gte('population', 500000);`
like	Case-sensitive pattern matching used to find rows where a column contains specific text	`client.from('cities').select().like('name', '%Do%');`
ilike	Case-insensitive pattern matching	`client.from('cities').select().ilike('name', '%shire%');`
in_	Membership in a list	`client.from('cities').select().in_('country_id', [1, 2, 3]);`

Combine Filters for Complex Queries

- This query combines multiple filters to return only students who meet all conditions

```
final students = await client
    .from('students')
    .select()
    .ilike('name', '%ali%')      // name contains "ali"
    .eq('status', 'active')      // only active students
    .gt('gpa', 3.0);             // GPA > 3.0
```

- Find products that are out of stock or very cheap

```
final result = await supabase
    .from('products')
    .select()
    .or('quantity.eq.0,price.lt.10’); // out of stock OR cheap
```

Best Practices

- Apply **Repository Pattern**: implement data read/write operations in repositories
- Use Providers to expose Supabase client and Repositories for easy access across the app
- **Optimize Queries**:
  - Use **projections**: `select('id, name')` instead of `select()` to reduce payload
  - Apply **server-side filtering** to retrieve only relevant data, improving application efficiency and responsiveness
  - Paginate large lists with `range(start, end)`
- **Error Handling**
  - Wrap calls in try/catch and show **user-friendly messages**
  - Log errors for debugging

Providers to expose Supabase client and Repositories

- 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);
});
```

🔐 Row Level Security (RLS)

< Back

Row Level Security (RLS)

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

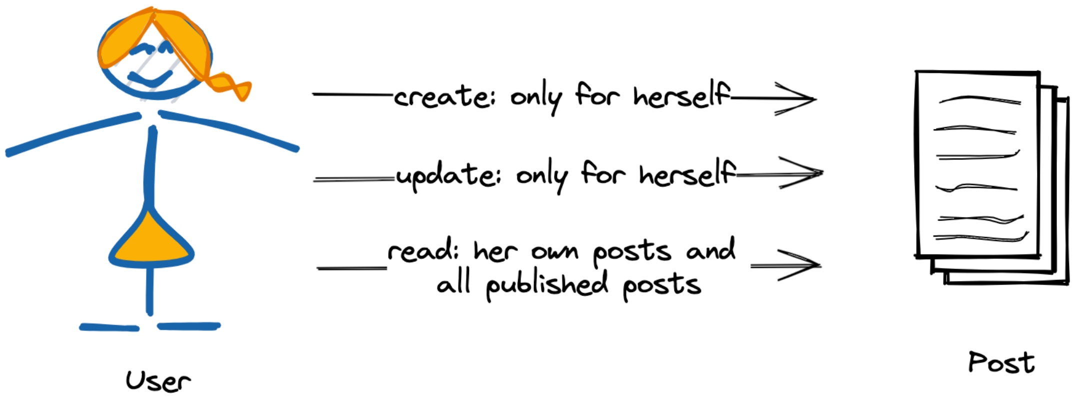
```
-- Enable RLS on the table
alter table todos enable row level security;
-- Policy: Read own rows. The user_id is a built-in variable
representing the current user
create policy "read own" on todos
  for select using (auth.uid() = user_id);
-- Policy: Modify own rows
create policy "modify own" on 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 );
```

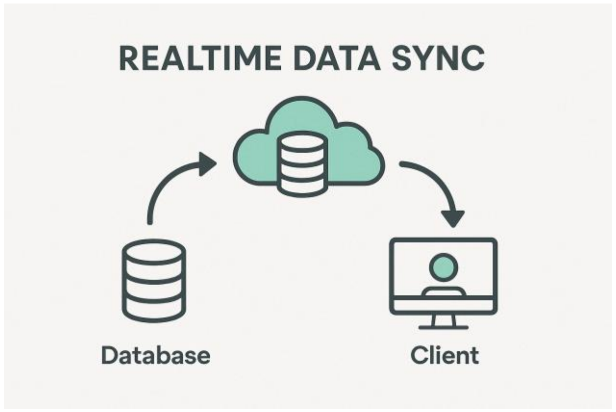
RLS Examples



```sql
-- 👁️ 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());
```

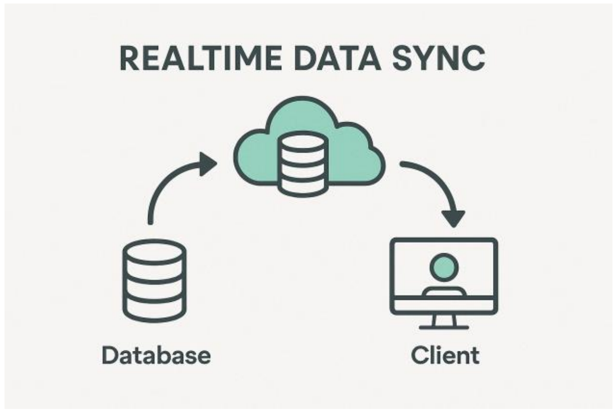
📡 Real-Time Updates



< Back

Real-Time Updates

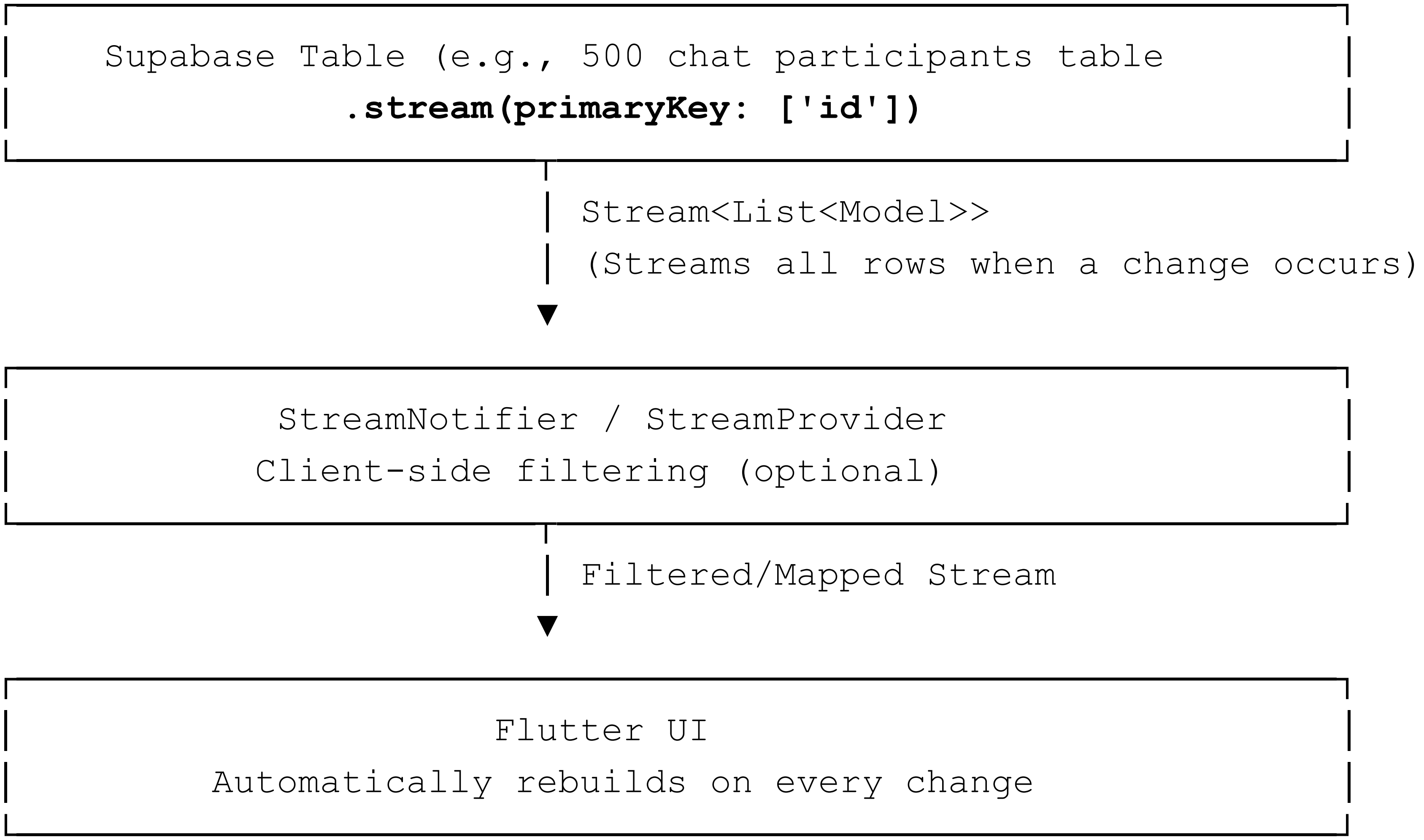
- **What It Does:** Broadcasts database changes instantly to connected clients
  - Enables **instant updates** without manual refresh, improving user engagement
  - Once realtime has been enabled, the table will broadcast any changes to authorized subscribers (i.e., multiple users see updates in real-time)
- **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 (e.g., Riverpod providers) for better user experience (UX)



Approach 1: Streams (`.stream()`)

- **Purpose:** streams continuously push **all table data** to your app
  - Best for small (< 1,000 rows), frequently-changing dataset where you need the entire table in memory
- **Real-World Scenarios:**
  - 🎮 **Gaming leaderboards** (top 100 players)
  - 💭 **Chat room participant list** (active users)
  - 📊 **Live dashboard metrics** (system stats)
  - 👥 **Online user presence** (who’s online)
- ❌ **Don’t Use Streams When:**
  - Table has 1,000+ rows (memory issues)
  - You only need filtered subset
  - Performance/scalability critical

Architecture Diagram - Streams


Supabase Table (e.g., 500 chat participants table
.stream(primaryKey: ['id'])
Stream<List<Model>>
(Streams all rows when a change occurs)
StreamNotifier / StreamProvider
Client-side filtering (optional)
Filtered/Mapped Stream
Flutter UI
Automatically rebuilds on every change

```
Events: INSERT → New row appears instantly
        UPDATE → Row updates instantly
        DELETE → Row disappears instantly
```

🔔 Listen to Database Realtime Updates using Streams

Postgrest 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());
}
```

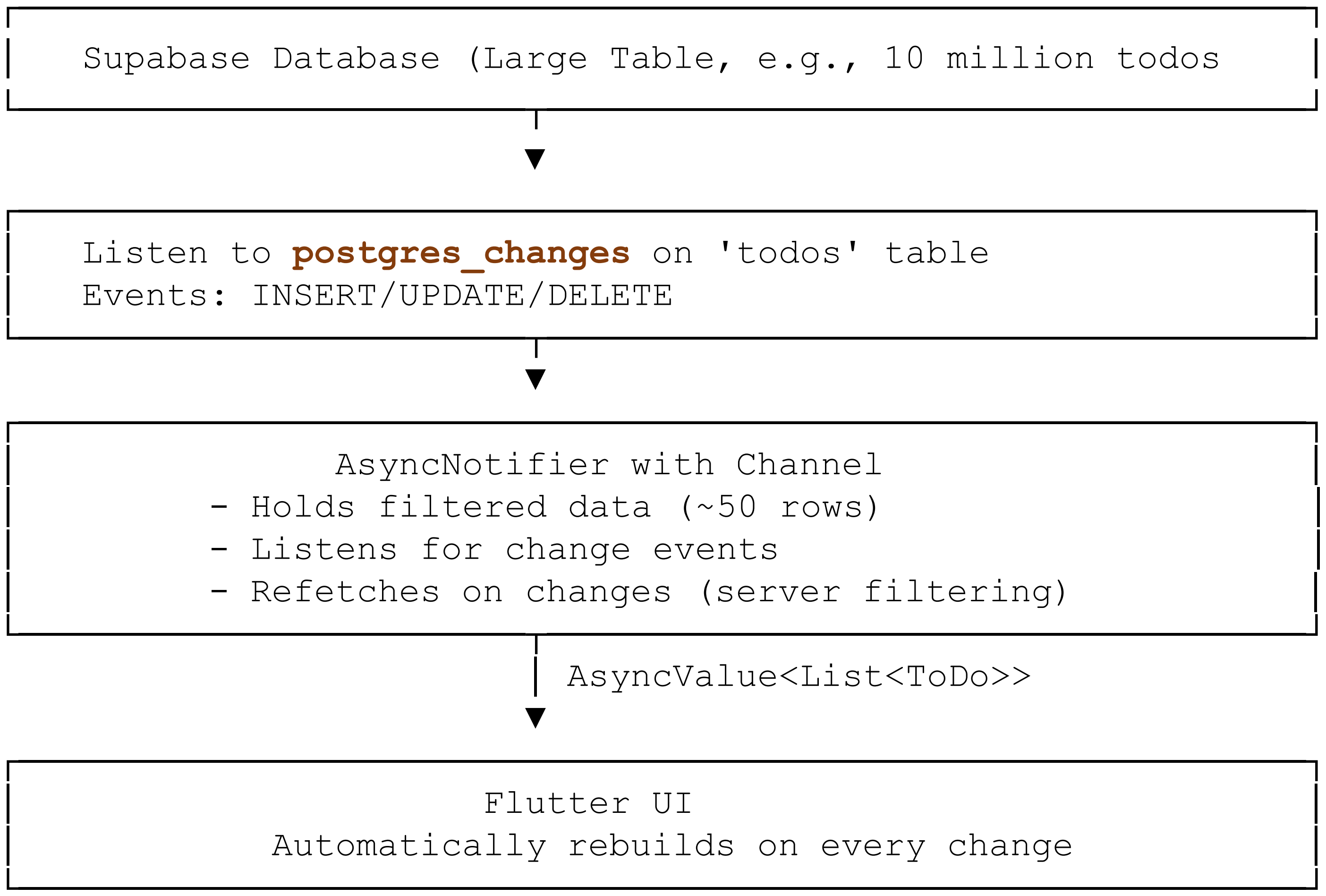
Implementation - Streams

```dart
// 1. Repository - Stream method
class MyRepository {
  final SupabaseClient _client;
  MyRepository(this._client);
  // Stream all rows from table
  Stream<List<MyModel>> observeItems() {
    return _client.from('my_table')
        .stream(primaryKey: ['id'])
        .map((data) => data.map((json) => MyModel.fromJson(json)).toList());
  }}
// 2. Provider - StreamNotifier
class MyItemsNotifier extends StreamNotifier<List<MyModel>> {
  @override
  Stream<List<MyModel>> build() {
    final repository = ref.watch(myRepositoryProvider);
    return repository.observeItems();
  }}
final myItemsProvider = StreamNotifierProvider<MyItemsNotifier, List<MyModel>>(() => MyItemsNotifier());
// 3. UI - Consume the stream
class MyScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final itemsAsync = ref.watch(myItemsProvider);
    return itemsAsync.when(
      data: (items) => ListView.builder(
        itemCount: items.length,
        itemBuilder: (context, index) => ListTile(
          title: Text(items[index].name),
        ),
      ),
      loading: () => CircularProgressIndicator(),
      error: (error, stack) => Text('Error: $error'),
    );
  }
}
```

Approach 2: Channels (`.channel()`)

- **Purpose:** Channels listen to **database change events** (INSERT/UPDATE/DELETE) and let you fetch only the data you need
  - Best for large datasets (**1,000+ rows**) with server-side filtering
  - Better performance and scalability
  - More control over what data is loaded
- **Real-World Scenarios:**
  - 📝 **Todo/Task management** (millions of todos, show only user’s)
  - 🛒 **E-commerce product catalog** (thousands of products, filtered view)
  - 📧 **Email inbox** (show only unread, specific folder)
  - 📊 **Analytics dashboard** (aggregated data)
  - 👤 **User activity feeds** (show last 50)

Architecture Diagram - Channels


Supabase Database (Large Table, e.g., 10 million todos
Listen to **postgres_changes** on 'todos' table
Events: INSERT/UPDATE/DELETE
AsyncNotifier with Channel
- Holds filtered data (~50 rows)
- Listens for change events
- Refetches on changes (server filtering)
AsyncValue<List<ToDo>>
Flutter UI
Automatically rebuilds on every change

Flow:

1. Initial load: Fetch with filters (e.g., 50 rows)
2. Setup: Subscribe to realtime channel
3. Event: INSERT/UPDATE/DELETE detected
4. Action: Re-fetch filtered data (still ~50 rows)
5. Result: UI updates with fresh data

Implementation - Channels

```
// 1. Repository - Regular filtered query
class MyRepository { ...
  // Filtered fetch (server-side WHERE clause)
  Future<List<MyModel>> getFilteredItems({ String? searchQuery }) async {
    var query = _client.from('my_table').select( ... );    ...
    return await query.map((json) => MyModel.fromJson(json)).toList();
  }}
// 2. Provider - AsyncNotifier with Channel
class MyItemsNotifier extends AsyncNotifier<List<MyModel>> {
  RealtimeChannel? _channel;
  @override
  Future<List<MyModel>> build() async {
    final searchQuery = ref.watch(searchQueryProvider);
    ...
    _channel = supabase.channel('my_items_changes')
        .onPostgresChanges(
          event: PostgresChangeEvent.all,
          schema: 'public', table: 'my_table',
          callback: (payload) {
             refresh(); // Refetch when changes detected
          },
        )
        .subscribe();
  }
  Future<void> refresh() async { ...
    state = await AsyncValue.guard(() => build());
  }
} ...
// 3. UI - Same as streams approach
```

Which Approach to Choose?

Quick Decision Tree

```
Do you have more than 1,000 rows?
├─ NO → Do you need ALL data displayed?
│    ├─ YES → Use STREAMS (simple, real-time)
│    └─ NO → Use CHANNELS (more efficient)
│
└─ YES → Use CHANNELS (scalability required)
```

Scenario 1: Small Real-Time App

- **Requirements:** Gaming leaderboard, 100 active players, need instant updates
- **Choice:** ✅ Streams
- **Reasoning:**
  - Small dataset, all data needed, low latency critical
- **Implementation:**

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

Scenario 2: Chat Application

- **Messages:** ✅ Channels (could be millions, paginated)
- **Participants:** ✅ Streams (< 500 users, all displayed)
- **Reasoning:** Different requirements per feature

Scenario 3: Social Media Feed

- **Requirements:** Millions of posts, show last 50, pagination
- **Choice:** ✅ Channels
- **Reasoning:** Massive dataset, paginated, filtered by date
- **Implementation:** `channel() + LIMIT 50`

Summary Recommendations

1. **Default Choice: Use Channels** for most production apps
   - Better scalability
   - More efficient
   - Handles large datasets
2. **Use Streams Only When:**
   - Dataset is small (< 1,000 rows)
   - Need all data
   - Simplicity is priority
3. **Always:**
   - Add database indexes
   - Enable RLS for security
   - Clean up subscriptions

📚 References

- **Supabase Flutter Docs:** https://supabase.com/docs/guides/getting-started/quickstarts/flutter
- **Supabase Database**

  https://supabase.com/docs/guides/database/overview
- **RLS Policies:** https://supabase.com/docs/guides/auth/row-level-security