Google Developer Groups
George Town

DEV DOJO

Develop Your App Easier and Faster with UI-Data Separation

Erico Darmawan Handoyo

- Parahyangan Catholic University
- Bandung Institute of Technology

- Lecturer @ Maranatha Christian University
- Flutter tutorial creator @ YouTube
- Flutter Trainer

youtube.com/@ericodarmawan

s.id/fb-ericodh

s.id/komunitas-flutter

s.id/ig-ericodh

s.id/flixid

s.id/in-ericodh

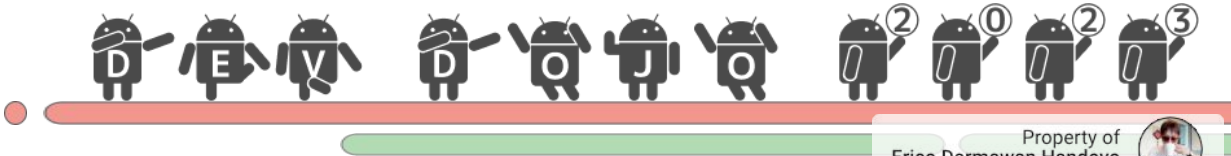
ericodarmawan.com

Background

DevDojo is hosted to give the participants of DevHack (hackathon) some knowledge to help them build their app on time.

Two things to be concerned:

1. Building app in a short period of time.
2. Having a good task distribution.



Building Your App on Time

1. Don't over complicate your codes.
2. Use available package to fasten the development process
3. Good team work.

Working as team

1. Don’t wait for each other.
2. Don’t let your code affect your partner’s codes.

⇩

Separation of concerns

Recommended Package & Database

Database Recommendations

1. Firebase (Nosql Database)
2. Supabase (Sql Database)

Reasons:

1. Support stream.
2. Provide enough freebies.
3. Easy to implement with Flutter project.
4. Easy to learn.

Support Stream ➡ Stream Builder

	Without Stream	Stream + Stream Builder
Adding Data	1. **Add** new data 2. If the creation success, **get** the updated list of data. 3. **Update** the UI.	1. **Add** new data 2. If the creation success, the UI is updated automatically
Updating Data	1. **Update** the data 2. If the update success, **get** the updated data. 3. **Update** the UI.	1. **Update** the data 2. If the update success, the UI is updated automatically
Deleting Data	1. **Delete** a data 2. If the deletion success, **get** the updated list of data. 3. **Update** the UI.	1. **Delete** a data 2. If the deletion success, the UI is updated automatically

Free plan

Firebase

1. **Authentication services**. Phone auth: 10 SMS sent/day.
2. Up to **1 GB database space**.
    a. Document writes: 20K writes / day
    b. Document reads: 50K reads / day
    c. Document deletes: 20K deletes / day
3. **Cloud Storage:** up to **5GB**
    a. Downloads: 1 GB / day
    b. Upload: 20K times / day
    c. Download: 50K times / day

Supabase

1. **Authentication services**.
2. Up to **500 MB database space**.
    a. Up to 5GB bandwidth.
3. **File Storage: 1 GB**
    a. Up to 50MB file uploads.

Free projects are paused after 1 week of inactivity.

Use Freezed package to help you:

1. Make immutable class.
2. Do object comparison by its properties.
3. Make a copy of the object.
4. Convert an object to a json.
5. Make an object from a json.
6. Override toString method to return a string representing the object.

Immutable class

Allowing you to make const Constructor → constant object:

1. Instantiated at compile-time → faster.
2. Can be reused → efficient.

```
Row(
  children: [
    Text('hello'),
    Text('hello'),
  ],
);
```

2 different object of Text

```
Row(
  children: [
    const Text('hello'),
    const Text('hello'),
  ],
);
```

1 object of Text used twice.

Object comparison

1. Default object comparison → if both objects are the same object.
2. In most of time, we want to compare objects by its properties, e.g. to test a return value from an API.

⬇

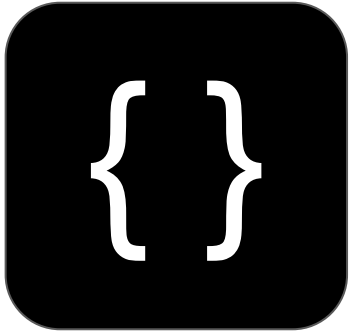
We need to override the equal operator and the getter of hashcode.

Make a copy of the object

Immutable object cannot be changed → need to make a copy

Object ↔ JSON

JSON is a common format used for transferring data from server to client and vice versa.



Override toString method

Default toString method returns the runtype of the object.

```
Instance of 'User'
```

→

```
User(id: 1, email: erico.dh@blackpink.com)
```


```dart
@immutable
class User {
  final String? id;
  final String email;

  const User({required this.id, required this.email});

  factory User.fromJson(Map<String, dynamic> json) {
    return User(id: json['id'], email: json['email']);
  }

  Map<String, dynamic> toJson() {
    return {'id': id, 'email': email};
  }

  User copyWith({String? id, String? email}) {
    return User(id: id ?? this.id, email: email ?? this.email);
  }

  @override
  String toString() {
    return 'User(id: $id, email: $email)';
  }

  @override
  bool operator ==(Object other) {
    if (identical(this, other)) return true;

    return other is User && other.id == id && other.email == email;
  }

  @override
  int get hashCode => Object.hash(runtimeType, id, email);
}
```

```dart
@freezed
class User with _$User {
  const factory User({
    String? id,
    required String email,
  }) = _User;

  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
}
```

Using Freezed package

Without using Freezed package

Separation of Concerns

Combining UI & data dayer is a bad practice

1. It's difficult to be developed by a group of developers.
2. Each system component is interconnected with each other.
    a. Updating codes in one place may affect the other codes.
    b. It's hard to test.

Cannot be done in parallel

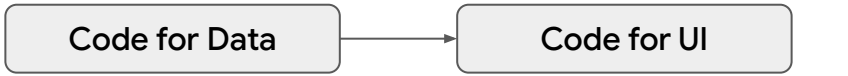
Code for UI → Code for Data

```
ElevatedButton(
  onPressed: () async {
    try {
      await Supabase.instance.client.auth.signInWithPassword(
          email: _emailController.text,
          password: _passwordController.text);
    } catch (e) {
      // do something with the error
    }
  },
  child: const Text('Login'),
),
```

1. It's **hard** to **divide** the **work**, because all the **codes** are in **one place**.
2. **Need UI to test** the database related code.
3. If there is **any changes** to **database-related codes**, you **must update all UI** that contains the codes. E.g. The syntax of getting Supabase instance change.

Many duplication of codes will risk more bugs.

Cannot be done in parallel




```
ElevatedButton(
  onPressed: () async {
    try {
      await SupabaseAuthentication().login(
          email: _emailController.text,
          password: _passwordController.text);
    } catch (e) {
      // do something with the error
    }
  },
  child: const Text('Login'),
),
```

1. **Need to wait** the **database related class** (SupabaseAuthentication) to use in the UI.
2. **Changes** in **database** related class **may affect** the **UI** related codes. E.g. Changing database, changing method in the database-related class.

Changing code that is already correct may risk introducing bugs.

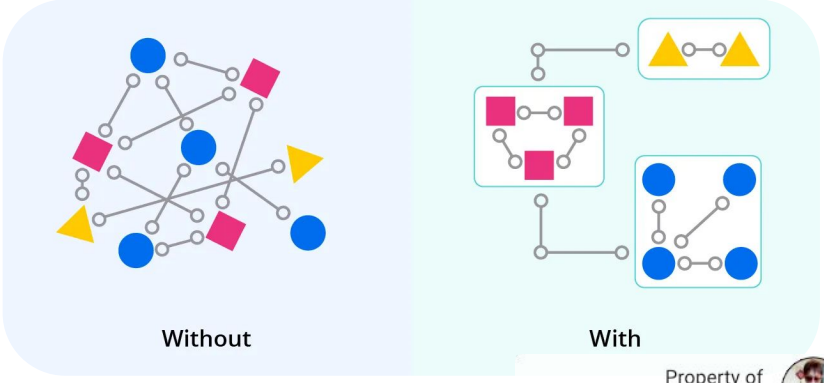
Separation of concerns

Separate your codes / system components based on its purpose and its possibility of change.

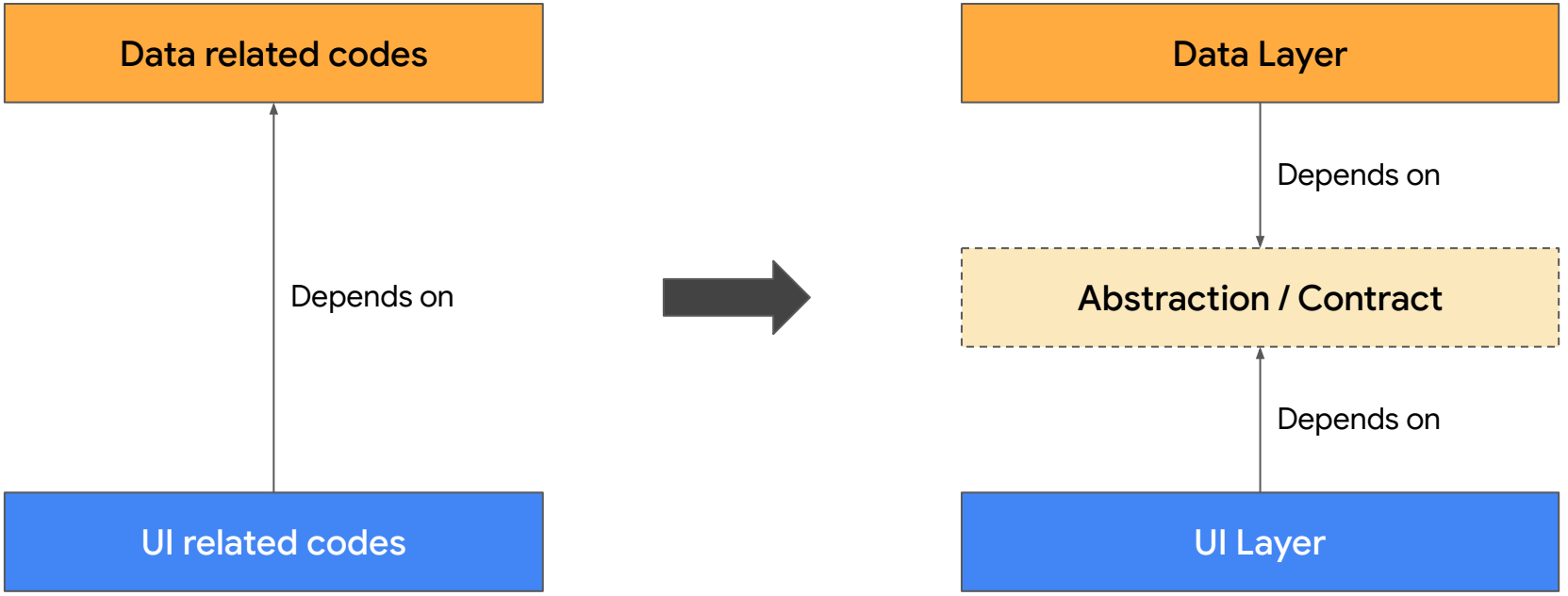
1. Each system components can be done separately in the same time.
2. Updating some codes will give no effect to the other codes.
3. Replacing a component will not affect the other part of the system.

How to do separation of concerns

1. Group all components that function related to be one group / layer / module.
2. Separate those groups / layers / modules with an abstraction / contract between them as a reference.
3. Each layer of the system should depends on the abstraction / contract, not the other layers.



The simplest separation of concerns



Flutter Showcase: Simple List to Do

Showcase 1 - List To Do (Bad ver. 1)

1. It's **difficult** to **distribute** the **task**.
2. It's **hard to test** → **Cannot be sure** that every important code is **correct** → **Difficult** to **find** the **cause of a bug** / error.
3. If you want to **change** the **database**, you have to **re-code** almost **everything** → will **risk introducing** new **bugs**.
4. It takes too **much time** and **effort** every time you **update** / **fix** your application.

Showcase 2 - List To Do (Bad ver. 2)

1. You have to **wait** the **data related codes** to finish your UI related codes.
2. **Changing** the **data related codes** may **affect** the **UI related code**. E.g. Changing the method name.
3. **Changing** the **data source** is **troublesome**.
4. If the **new database** has **differences** with the **old database**, you must **change** the **data model** and **UI codes**. E.g. Supabase table auto-id is an integer while Firebase collection id is a String.

Showcase 3 - List To Do with UI-Data separation

1. You can **complete UI code without waiting** for data-related code to complete.
2. You can **change** the **data source easily**.
3. You can **easily test** your data classes to make sure everything is working properly.
4. Your **UI doesn't care** about the **data source**. Each **data source** class **must meet** the **requirements** described in the **abstraction** (interface).
5. The **difference** between data sources **doesn't effect** the data model used in the app.

Have enough time?

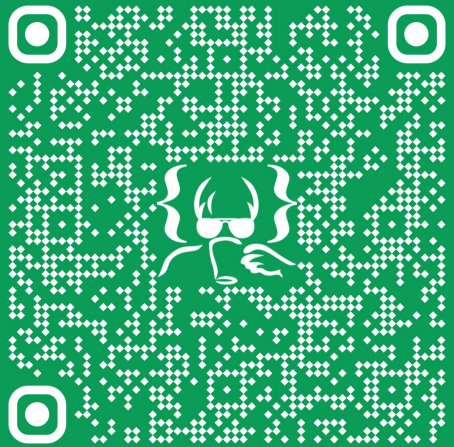
1. Dependency Injection.
2. Managing states of stream / future method.
3. Managing application states.

Riverpod

```
@riverpod
Stream<List<Todo>> todos(TodosRef ref, String uid) =>
    ref.watch(todoRepositoryProvider).todos(uid);
```

```
ref.watch(todosProvider(user.id!)).when(
  data: (todos) {
    int count = todos.where((todo) => !todo.completed).length;
    return Text(count > 0
        ? 'You have $count thing(s) to do.'
        : 'Congrats! You have nothing to do.');
  },
  error: (error, stackTrace) => const Text('Error loading data'),
  loading: () => const Center(
    child: CircularProgressIndicator(),
  ),
)
```


Thank you



youtube.com/@ericodarmawan

s.id/komunitas-flutter

s.id/flixid

ericodarmawan.com

s.id/fb-ericodh

s.id/ig-ericodh

s.id/in-ericodh