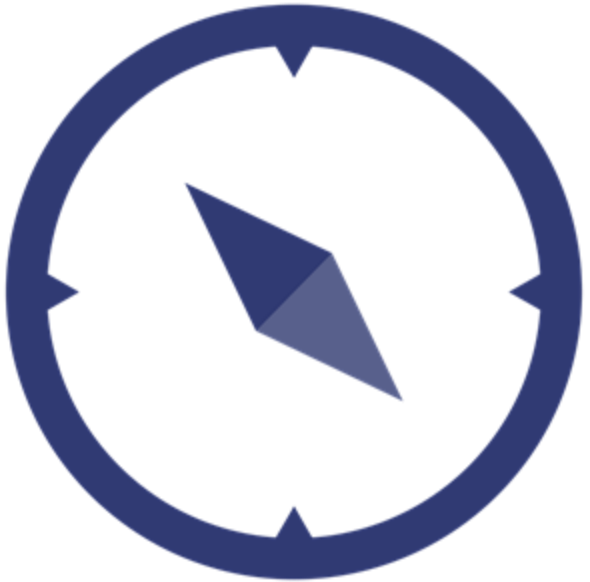
FAKULTAS
ILMU
KOMPUTER

Navigation, Networking, and Integration

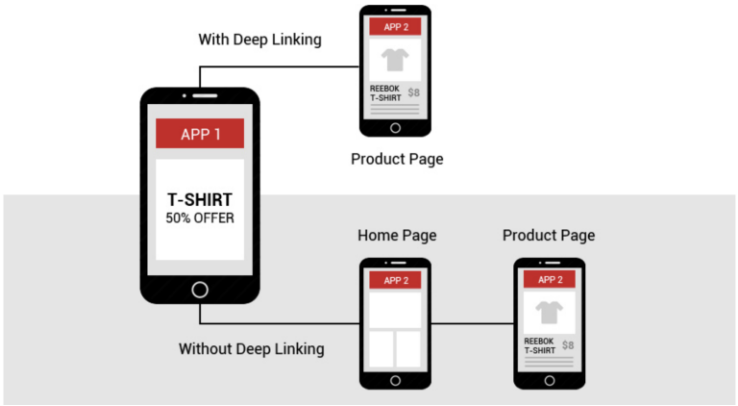
Tim Dosen PBP

Navigation



Navigation and Routing

- Flutter provides a complete system for navigating between screens and handling deep links.
  - Deep links are a type of link that send users directly to an app instead of a website or a store. They are used to send users straight to specific in-app locations.
  - Deep linking is the ability to link into a specific page inside of a native iOS or Android mobile app (as opposed to a mobile website).
  - https://docs.flutter.dev/development/ui/navigation/deep-linking
  - Example: https://tokopedia.link/yQEajbpyutb



- Image from https://neilpatel.com/blog/mobile-deep-linking/

Navigation and Routing

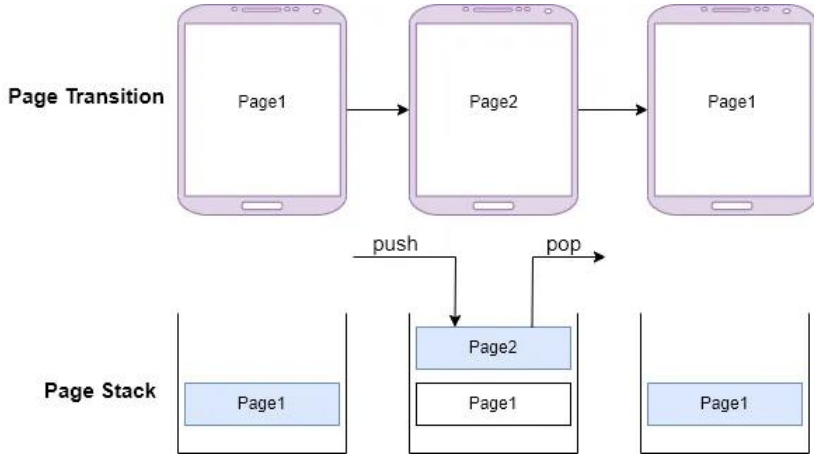
- Small applications without complex deep linking can use `Navigator`, while apps with specific deep linking and navigation requirements should also use the `Router` to correctly handle deep links on Android and iOS, and to stay in sync with the address bar when the app is running on the web.
- https://docs.flutter.dev/development/ui/navigation

Navigation

- The main players in the navigation layer are as follows:
  1. `Navigator`: The `Route` manager
     - The `Navigator` widget is the main player in the task of moving from one screen to another. The `Navigator` widget is responsible for managing screen changes with a logical history idea.
  2. `Overlay`: Navigator uses this to specify appearances of the routes
     - Overlays let independent child widgets appear on top of other widgets by inserting them into the overlay's Stack.
     - Although you can create an `Overlay` directly, it's most common to use the overlay created by the `Navigator` in a `WidgetsApp` or a `MaterialApp`.
  3. `Route`: A navigation endpoint
     - When we want to navigate to a new screen, we define a new `Route` widget to it, in addition to some parameters defined as a `RouteSettings` instance.

Navigation

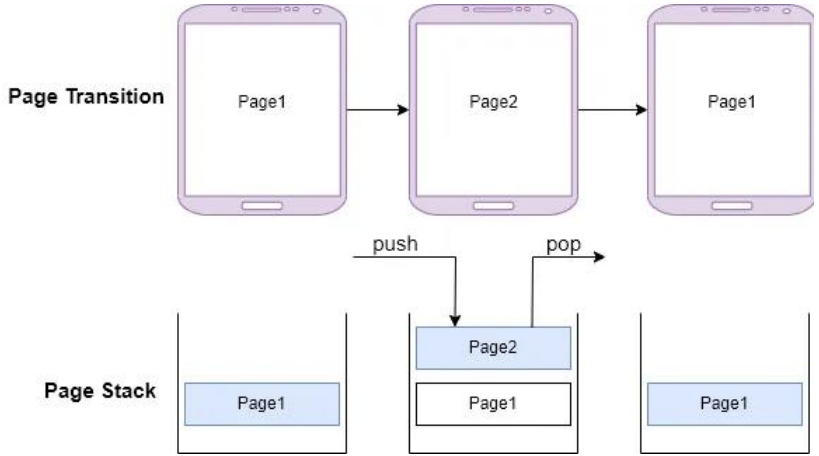
- Navigation in Flutter is made in a *stack* structure.
- The stack structure is suitable for this task because its concept is very similar to a screen's behavior:
  - We have one element at the *top of the stack*. In `Navigator`, the top-most element on the stack is the currently visible screen of the app.
  - The last element inserted is the first to be removed from the stack (commonly refereed as **last in first out** (**LIFO**)). The last screen visible is the first that is removed.
  - Like stack, the `Navigator` widget's main methods are `push()` and `pop()`



https://www.technicalfeeder.com/2021/11/flutter-page-transition/

Navigation

- The `push()` method adds a new screen to the top of the navigation stack.
- The `pop()` method removes a screen from the navigation stack.
- The navigation stack is also known as navigation history.
- The navigation stack elements are `Route` objects. There are multiple ways to define them in Flutter.
- The `Navigator` keeps a stack of `Route` objects (representing the history stack).



Navigation

Navigator.push(context, [MaterialPageRoute instance]);
Navigator.pop(context);

The `MaterialPageRoute` object is a subclass of `Route` that specifies the transition animations for Material Design.

GO TO NEXT PAGE:

```
Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => const SecondRoute()),
);
```

BACK:

```
Navigator.pop(context);
```

Example GO TO

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('First Route'),
    ),
    body: Center(
      child: ElevatedButton(
          child: const Text('Open route'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => const SecondRoute()),
            );
          }
      ),
    ),
  );
}
```

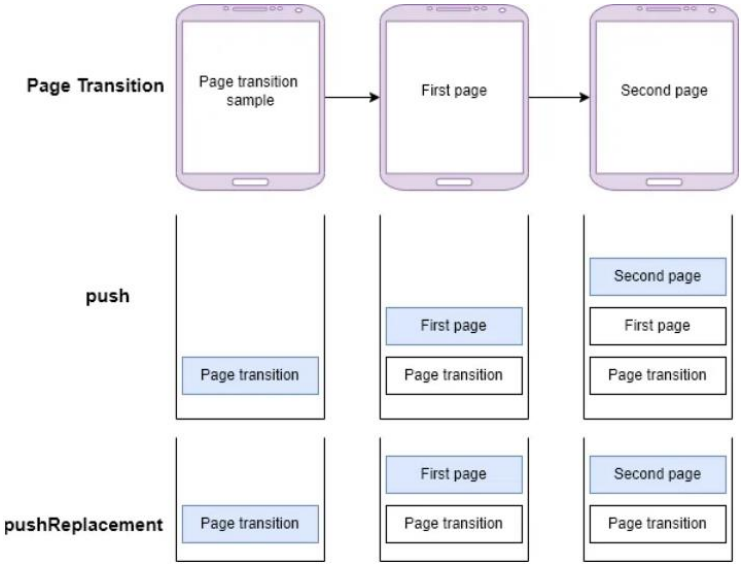
Example BACK

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text("Second Route"),
    ),
    body: Center(
      child: ElevatedButton(
        onPressed: () {
          Navigator.pop(context);
        },
        child: const Text('Go back!'),
      ),
    ),
  );
}
```

Navigation

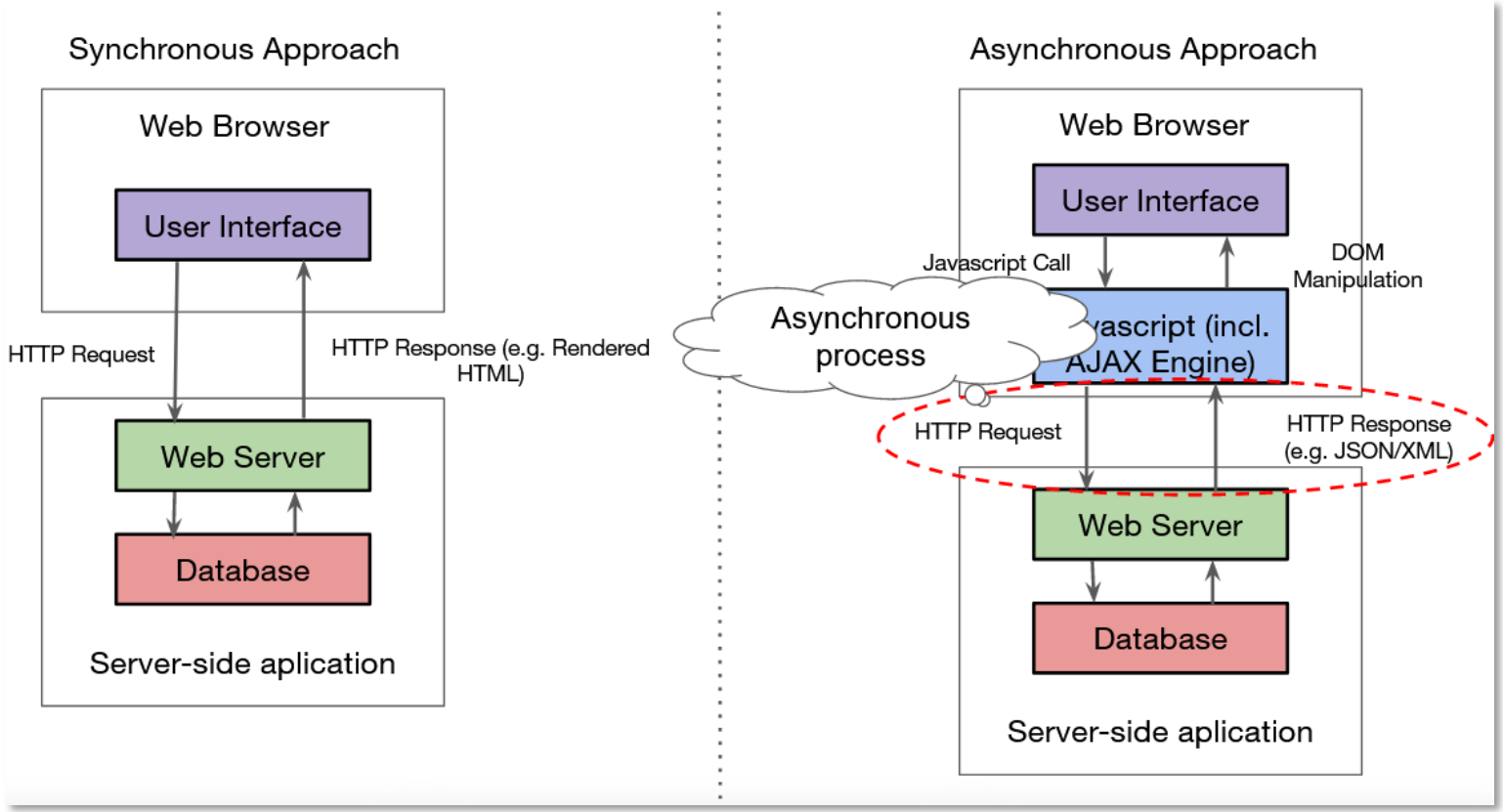
- More about `Navigator` widget: https://api.flutter.dev/flutter/widgets/Navigator-class.html

- `popUntil(BuildContext context, RoutePredicate predicate)`
  - Calls `pop` repeatedly on the navigator until the predicate returns true.
- `pushAndRemoveUntil<T extends Object?>(BuildContext context, Route<T> newRoute, RoutePredicate predicate)`
  - `Push` the given route onto the navigator, and then remove all the previous routes until the predicate returns true.
- `pushReplacement`
  - Replace the current route of the navigator by pushing the given route and then disposing the previous route once the new route has finished animating in.



Asynchronous Call

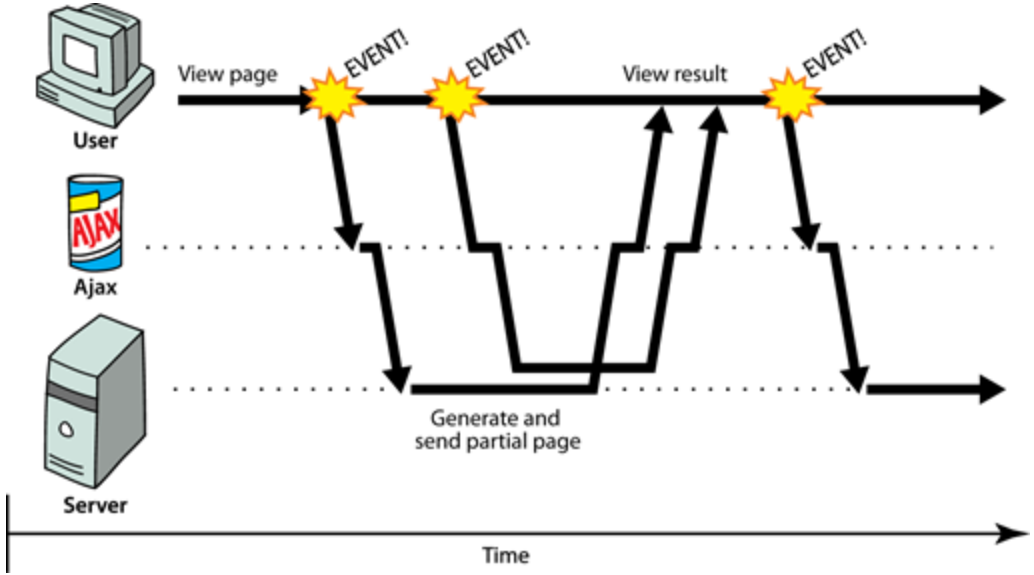
Review: Synchronous & Asynchronous Approach



Asynchronous Call

user can keep interacting with page while data loads

Source: https://courses.cs.washington.edu/courses/cse154/12au/lectures/slides/lecture22-ajax.shtml#slide3



Make Asynchronous Requests

Just like AJAX (Asynchronous JavaScript And XML) Request in web programming, asynchronous operations let your program complete work while waiting for another operation to finish. Here are some common asynchronous operations:

- Fetching data over a network.
- Writing to a database.
- Reading data from a file.

To perform asynchronous operations in Dart, you can use the **`Future`** class and the **`async`** and **`await`** keywords.

Future

- A future (lower case “f”) is an instance of the `Future` (capitalized “F”) class.
- https://api.flutter.dev/flutter/dart-async/Future-class.html
- A future represents the result of an asynchronous operation and can have two states: **uncompleted** or **completed**.
  - **Uncompleted**: When you call an asynchronous function, it returns an uncompleted future. That future is waiting for the function’s asynchronous operation to finish or to throw an error.
  - **Completed**: If the asynchronous operation succeeds, the future completes with a value. Otherwise, it completes with an error.

`async` and `await`

- Remember these two basic guidelines when using `async` and `await`:
  - **To define an async function, add `async` before the function body**
    - Example: `Future<void> main() async { ··· }`
  - **The `await` keyword works only in async functions**
    - Example 1: `var order = await fetchUserOrder();`
    - Example 2: `print(await createOrderMessage());`
- An async function runs synchronously until the first `await` keyword. This means that within an async function body, all synchronous code before the first `await` keyword executes immediately.

`async` and `await` Pattern

Try it in Dartpad!
https://www.dartpad.dev/

```
Future<void> printOrderMessage() async {
  print('Awaiting user order...');
  var order = await fetchUserOrder();
  print('Your order is: $order');
}

Future<String> fetchUserOrder() {
  // Imagine that this function is more complex and slow.
  return Future.delayed(const Duration(seconds: 4), () => 'Large Latte');
}

Future<void> main() async {
  countSeconds(4);
  await printOrderMessage();
}

// You can ignore this function - it's here to visualize delay time in this example.
void countSeconds(int s) {
  for (var i = 1; i <= s; i++) {
    Future.delayed(Duration(seconds: i), () => print(i));
  }
}
```

Integration to Webservice

How to Add the http Package

- https://docs.flutter.dev/cookbook/networking/fetch-data
- To install the latest http package, either add it using flutter pub add http command in your shell or manually add it to the dependencies section of the pubspec.yaml file

```
# Latest version of `http` package as of 13 November 2023
# Will install `http` package version >=1.1.0 && version < 2.0.0
dependencies:
  http: ^1.1.0
```

- In your Dart file, import the http package.

```
import 'package:http/http.dart' as http;
```

- Additionally, in your AndroidManifest.xml file, add the Internet permission.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Cross-Platform HTTP Networking

- The `http` package provides the simplest way to issue HTTP requests.
- This package is supported on Android, iOS, and the web.
- Examples of networking tasks:
  - fetch data from the internet
  - send data to the internet
  - update data over the internet
  - parse data in the background
- https://docs.flutter.dev/development/data-and-backend/networking

JSON and Serialization

- When making network-connected apps, the chances are that it needs to consume JSON, sooner or later.
- Two general strategies for working with JSON:

  1. Manual serialization
  - Use manual serialization for smaller projects
  - Manual JSON decoding refers to using the built-in JSON decoder in `dart:convert`

  2. Automated serialization using code generation
  - Use code generation for medium to large projects
  - JSON serialization with code generation means having an external library generate the encoding boilerplate for you

JSON - manual serialization

- Flutter has a built-in dart:convert library that includes a straightforward JSON encoder and decoder.

```
import 'dart:convert';
```

- With dart:convert, you can serialize this JSON model in two ways:
  1. Serializing JSON inline
  2. Serializing JSON inside model classes

JSON - manual serialization

1. Serializing JSON inline
   - decode the JSON by calling the jsonDecode() function, with the JSON string as the method argument
   - jsonDecode() returns a Map<String, dynamic>, meaning that you do not know the types of the values until runtime
2. Serializing JSON inside model classes
   - Inside your model class, you can use:
     - fromJson() method for constructing an instance from a map structure
     - toJson() method for converting an instance into a map

JSON - manual serialization

https://docs.flutter.dev/development/data-and-backend/json#serializing-json-inline

Serializing JSON inline example:

- JSON

```
{
  "name": "John Smith",
  "email": "john@example.com"
}
```

- Dart code

```
Map<String, dynamic> user = jsonDecode(jsonString);

print('Hello, ${user['name']}!');
print('We sent the verification link to ${user['email']}.');
```

JSON - manual serialization

https://docs.flutter.dev/development/data-and-backend/json#serializing-json-inside-model-classes

Serializing JSON inside model classes example:

```
class User {
  final String name;
  final String email;

  User(this.name, this.email);

  User.fromJson(Map<String, dynamic> json)
      : name = json['name'],
        email = json['email'];

  Map<String, dynamic> toJson() => {
        'name': name,
        'email': email,
      };
}
```

JSON - manual serialization

More examples:

- https://docs.flutter.dev/cookbook#networking
- https://docs.flutter.dev/cookbook/networking/fetch-data#complete-example
- https://flutter.github.io/samples/jsonexample.html

JSON - automated serialization using code generation

- Choose a library and include it in your project
- You can choose one of libraries on pub.dev, that is json_serializable
- json_serializable is an automated source code generator that generates the JSON serialization boilerplate for you
- https://docs.flutter.dev/development/data-and-backend/json#serializing-json-using-code-generation-libraries
- https://pub.dev/packages/json_serializable
- https://app.quicktype.io/

Fetching data over network

Example GET from Django:

```
Future<void> fetchData() async {
  const url = 'http://127.0.0.1:8000/view_anggota';
  try {
    final response = await http.get(Uri.parse(url));
    print(response.body);
    Map<String, dynamic> extractedData = jsonDecode(response.body);
    extractedData.forEach((key, val) {
      print(val);
    });
  } catch (error) {
    print(error);
  }
}
```

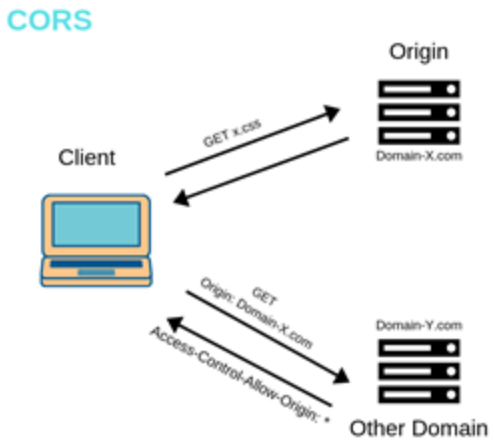
Fetching data over network

Example Views in Django:

https://docs.djangoproject.com/en/4.1/ref/csrf/#django.views.decorators.csrf.csrf_exempt

```
from django.shortcuts import render
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import time

@csrf_exempt
def index(request):
    data = {
            'nama': 'My Name',
            'alamat': 'My Location'
    }
    time.sleep(10)
    return JsonResponse(data, safe=False)
```



Don't forget about CORS Headers
https://pypi.org/project/django-cors-headers/

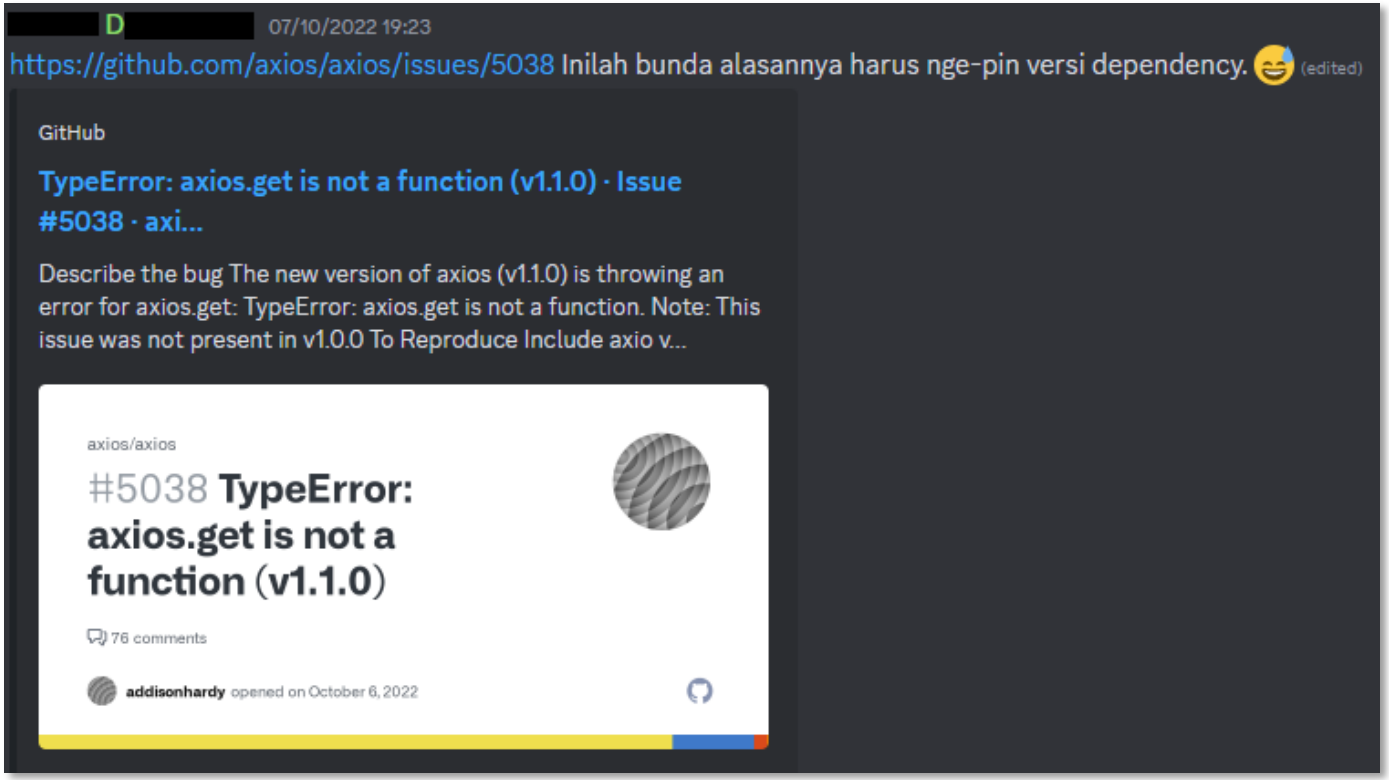
References

- Flutter documentation:
  - https://docs.flutter.dev/development/ui/navigation
  - https://docs.flutter.dev/development/ui/navigation/deep-linking
  - https://docs.flutter.dev/development/data-and-backend/networking
  - https://docs.flutter.dev/development/data-and-backend/json
  - https://docs.flutter.dev/cookbook/networking/fetch-data
- Dart and Flutter packages and API:
  - https://api.flutter.dev/flutter/widgets/Navigator-class.html
  - https://api.flutter.dev/flutter/dart-async/Future-class.html
  - https://pub.dev/packages/json_serializable

References (2)

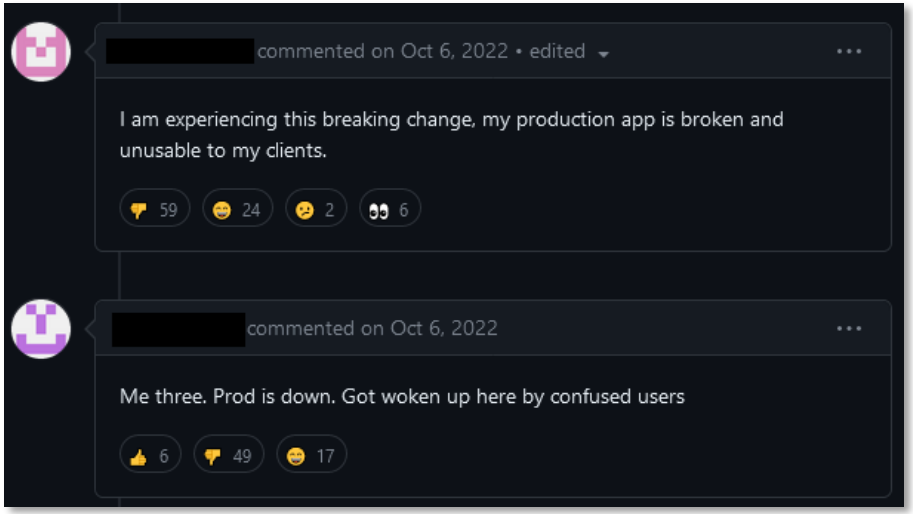
- Other references:
  - https://www.technicalfeeder.com/2021/11/flutter-page-transition/
  - https://courses.cs.washington.edu/courses/cse154/12au/lectures/slides/lecture22-ajax.shtml#slide3
  - https://dart.dev/codelabs/async-await
  - https://flutter.github.io/samples/jsonexample.html
  - https://pypi.org/project/django-cors-headers/
  - https://docs.djangoproject.com/en/4.1/ref/csrf/#django.views.decorators.csrf.csrf_exempt

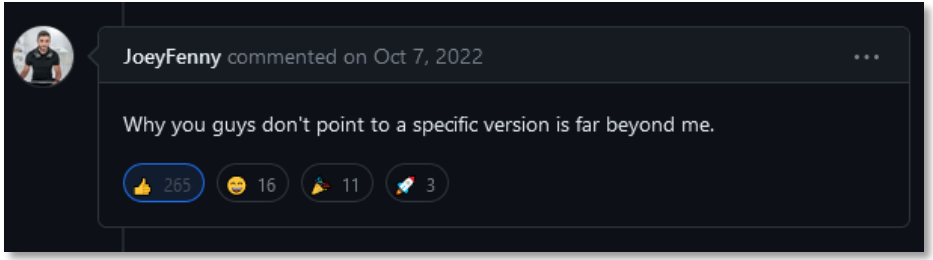
Additional Material: Dependency Version Pinning



Source: https://github.com/axios/axios/issues/5038 (*it is not a Django or Flutter related, but the moral of the story still applies*)

Additional Material: Dependency Version Pinning





Additional Material: Dependency Version Pinning

- `latest` version is not always the greatest, especially when using 3rd party packages
  - Even a **minor** version update in a dependency could break the app
  - Unpinned version may lead to reproducibility issue, e.g., inconsistent build and development environment among project collaborators
- Solution: dependency version pinning using Semantic Versioning
  - Version number follows format: `X.Y.Z`
    - X: major version
    - Y: minor version
    - Z: patch version
  - Version range symbol: ^, ~
    - ^: Pin the **major version**, but not the minor and patch version
    - ~: Pin the **major and minor version**, but not the patch version
    - *No symbol*: Pin exact version as written
  - Example: `http ^1.1.0, http ~1.1.0, http 1.1.0` in Flutter
    - `http ^1.1.0` will install the latest compatible version in range **>= 1**.1.0 and range **< 2**.0.0
    - `http ~1.1.0` will install the latest compatible version in range **>= 1.1**.0 and range **< 1.2**.0
    - `http 1.1.0` will ONLY install http at exact version **1.1.0**
- When you are collaborating with others, make sure you have locked the dependencies and avoid upgrading package version independently!
  - This is already been handled by Flutter by having `pubspec.lock` file auto-updated each time someone installed new package into the project
- Reference:
  - https://devhints.io/semver