Enforcing type safety

in Flutter & Dart

Guillaume Roux

Mobile Engineer at Shipfix
Co-organizer of Flutter Lyon

@TesteurManiak

What does “type safe” means?

“Extent to which a programming language discourages or prevents type errors.”

“Type enforcement can be static, catching potential errors at compile time, or dynamic, associating type information with values at run-time [...] or a combination of both.”

https://en.wikipedia.org/wiki/Type_safety

The Dart type system

*"The Dart language is **type safe**: it uses a **combination of static type checking and runtime checks** to ensure that a variable's value always matches the variable's static type, sometimes referred to as sound typing. [...]*

One benefit of static type checking is the ability to find bugs at compile time using Dart's static analyzer."

https://dart.dev/language/type-system

We won’t interact with the Dart Compiler

Linting rules

Enabling stricter type checks

If you want stricter static checks than the Dart type system requires consider enabling those modes.

You can use those modes together or separately (all default to false).

```yaml
analyzer:
    language:
        strict-casts: true
        strict-inference: true
        strict-raw-types: true
```

https://dart.dev/tools/analysis#enabling-additional-type-checks

Enabling stricter type checks

This mode ensures that the type inference engine never implicitly casts from `dynamic` to a more specific type.

```
analyzer:
    language:
        strict-casts: true
        strict-inference: true
        strict-raw-types: true
```

```
void foo(List<String> lines) {
  ...
}

void bar(String jsonText) {
  // Implicit cast
  foo(jsonDecode(jsonText));
}
```

Enabling stricter type checks

This mode ensures that the type inference engine never chooses the `dynamic` type when it can't determine a static type.

```
analyzer:
    language:
        strict-casts: true
        strict-inference: true
        strict-raw-types: true
```

```
// Inference failure
final lines = {};
```

Enabling stricter type checks

This mode ensures that the type inference engine never chooses the `dynamic` type when it can't determine a static type due to omitted type arguments.

```
analyzer:
    language:
        strict-casts: true
        strict-inference: true
        strict-raw-types: true
```

```
// List with raw type
List numbers = [1, 2, 3];

for (final n in numbers) {
  print(n.length); // Runtime error
}
```

Add linting rules

By adding linting rules you will be able to enforce a coding style and avoid silly mistakes.

```yaml
linter:
  rules:
    - cast_nullable_to_non_nullable
    - conditional_uri_does_not_exist
    - prefer_null_aware_method_calls
    - tighten_type_of_initializing_formals
    - type_annotate_public_apis
```

Add linting rules

DON'T cast a nullable value to a non nullable type. This hides a null check and most of the time it is not what is expected.

https://dart.dev/tools/linter-rules/cast_nullable_to_non_nullable

```
linter:
  rules:
    - cast_nullable_to_non_nullable
    - conditional_uri_does_not_exist
    - prefer_null_aware_method_calls
    - tighten_type_of_initializing_formals
    - type_annotate_public_apis
```

Add linting rules

DON'T reference files that do not exist in conditional imports.

Code may fail at runtime if the condition evaluates such that the missing file needs to be imported.

```
linter:
  rules:
    - cast_nullable_to_non_nullable
    - conditional_uri_does_not_exist
    - prefer_null_aware_method_calls
    - tighten_type_of_initializing_formals
    - type_annotate_public_apis
```

https://dart.dev/tools/linter-rules/conditional_uri_does_not_exist

Add linting rules

Instead of checking nullability of a function/method f before calling it you can use `f?.call()`.

https://dart.dev/tools/linter-rules/prefer_null_aware_method_calls

```
linter:
  rules:
    - cast_nullable_to_non_nullable
    - conditional_uri_does_not_exist
    - prefer_null_aware_method_calls
    - tighten_type_of_initializing_formals
    - type_annotate_public_apis
```

Add linting rules

Tighten the type of an initializing formal if a non-null assert exists. This allows the type system to catch problems rather than have them only be caught at run-time.

https://dart.dev/tools/linter-rules/tighten_type_of_initializing_formals

```
linter:
  rules:
    - cast_nullable_to_non_nullable
    - conditional_uri_does_not_exist
    - prefer_null_aware_method_calls
    - tighten_type_of_initializing_formals
    - type_annotate_public_apis
```

Add linting rules

Annotate the parameter and return types of public methods and functions to help users understand what the API expects and what it provides.

```
linter:
  rules:
    - cast_nullable_to_non_nullable
    - conditional_uri_does_not_exist
    - prefer_null_aware_method_calls
    - tighten_type_of_initializing_formals
    - type_annotate_public_apis
```

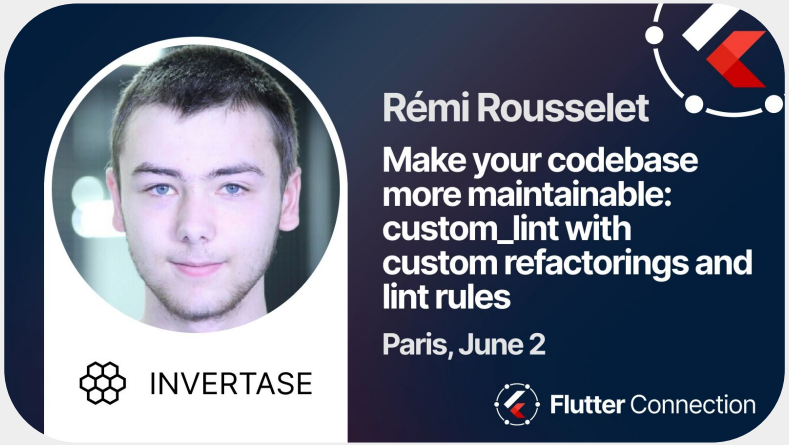
https://dart.dev/tools/linter-rules/type_annotate_public_apis

Check more at

https://dart.dev/tools/linter-rules

Create custom linting rules

Using packages such as custom_lint or by directly interacting with the Dart analyzer you can create your own set of rules. (e.g: `avoid_as`, `avoid_non_null_assertion`)



https://youtu.be/xZQJftO1C-4?si=OiVGcILjhMKkAaXb

Coding Style

Sealed Classes

```dart
sealed class Vehicle {}

class Car extends Vehicle {}

class Truck implements Vehicle {}

class Bicycle extends Vehicle {}

String getVehicleSound(Vehicle vehicle) {

  // ERROR: The switch is missing the Bicycle subtype or a default case

  return switch (vehicle) {
    Car() => 'vroom',
    Truck() => 'VROOOOMM',
  };

}
```

Type promotion

```dart
Object foo() { ... }

final bar = foo();
if (bar is String) {
  print(bar);
}

// OR

if (foo() case final String bar) {
  print(bar);
}
```

```dart
String? foo() { ... }

final bar = foo();
if (bar != null) {
  print(bar);
}

// OR

if (foo() case final bar?) {
  print(bar);
}
```

Pattern matching

```
Object myObject = ...;
switch (myObject) {
  case int():
    print('Is int: $myObject');
  case String():
    print('Is string: "$myObject"');
  default:
    print('Unrecognized type ${myObject.runtimeType}: $myObject');
}
```

Packages & Code generation

flutter_localizations

```
{
  "@@locale": "en",
  "home_title": "Home",
  "home_date": "Hello World on {date}",
  "@home_date": {
    "placeholders": {
      "date": {
        "type": "DateTime",
        "format": "yMd"
      }
    }
  }
}
```

https://docs.flutter.dev/ui/accessibility-and-localization/internationalization

```
final strings = AppLocalizations.of(context);

Column(
  children: [
    Text(strings.home_title),
    Text(strings.home_date(DateTime.now())),
  ],
);
```

flutter_gen

A code generator for your assets, fonts, colors, etc. that will make you get rid of string-based APIs.

With this, you won't ever get an `"Unable to load asset"` error.

```
Image.asset(
  Assets.images.profile
);
```

https://pub.dev/packages/flutter_gen

freezed

A code generator used to create data-classes/unions/cloning.

While methods `when` & `map` are less relevant with `sealed` classes the generation for equality operator, `hashCode` & `copyWith` is still welcome.

https://pub.dev/packages/freezed

```dart
@freezed
sealed class Union with _$Union {
  const factory Union.data(int value) = Data;
  const factory Union.loading() = Loading;
}

const union = ...;
print(
  union.when(
    data: (int value) => 'Data $value',
    loading: () => 'loading',
  ),
);
```

Other tools

Depending of the tools you are using (or willing to use), there might be a package to help you to enforce type-safety in your code:

- go_router_builder
- riverpod
- retrofit
- graphql_codegen
- injectable
- theme_tailor
- etc.

Wrap up

- Avoid bugs and crashes
- Anticipate edge cases
- More maintainable code

- More verbose code
- Long build_runner time with a lot of codegen

Thank You!

Questions?

Guillaume Roux

Mobile Engineer at Shipfix
Co-organizer of Flutter Lyon

@TesteurManiak

github.com/TesteurManiak/enforce-type-safety-in-flutter-talk