Adopting native assets for cross-platform FFI plugins

Simon Binder

2025-06-25

Adopting ~~native~~ code assets for cross-platform FFI plugins

Simon Binder

2025-06-25

Introduction

Calling native libraries from Flutter

Dart

Also compiles to machine code!

Native libraries

OBJECTBOX

SQLite

sigrok

libsodium

Compiles to machine code

This should be so easy!

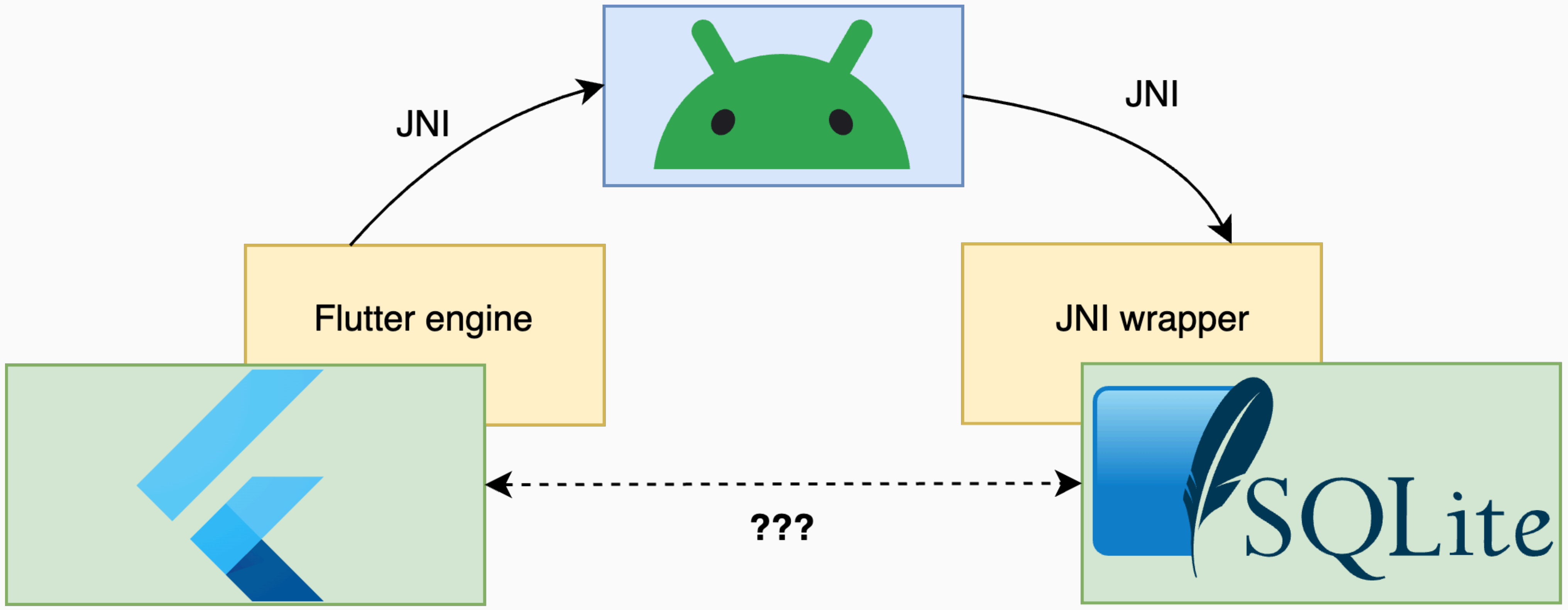
```c
// extern int cool_library_function();
// extern int useful_field;

   int main() {
     int x = cool_library_function();
     useful_field = x * 2;
   }
```

Expected behavior for Dart too:

- Fast
- Portable
- Simple
- Doesn't get in the way:
  - Synchronous
  - Shared memory

What options do we have?

Platform channels

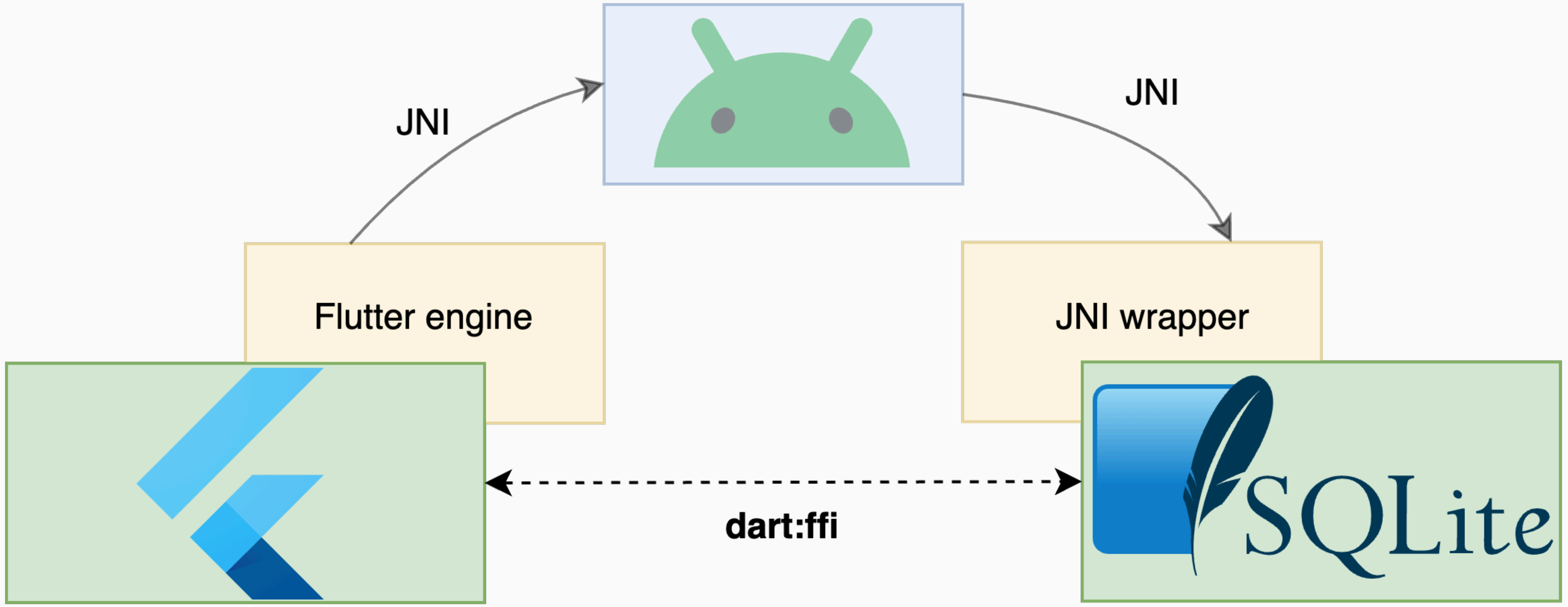


Platform channels: Not the right tool for the job!

Goal	Platform channels
Fast	Serialization overhead 🤷
Portable	Flutter-only, platform-specific code required ❌
Simple	Complex translation layer ❌
Synchronous	❌
Shared memory	❌

Can’t be used in unit/widget tests, limited isolate support, …

dart:ffi



A quick look at `dart:ffi`

- Obtain native library
- Lookup symbol dynamically
- Convert to Dart
- **Call / load / store**

dart

```dart
import 'dart:ffi';

void main() {
  final lib = DynamicLibrary.open('my_library.dylib');
  final function = lib.lookupFunction<Int Function(), int Function()>(
    'cool_library_function',
  );
  final field = lib.lookup<Int>('useful_field');

  var x = function();
  field.value = x * 2;
}
```

Benefits of `dart:ffi`

Goal	Platform channels	`dart:ffi`
Fast	🤷‍♂️	Low overhead ✓
Portable	❌	All native platforms ✓
Simple	❌	Not quite ❌
Synchronous	❌	✓
Shared memory	❌	✓

Obtaining the libraries

- We can **access** methods from native libraries
  - (after we've loaded them)
- We can **load** libraries
  - (if they're available)
- We cannot make them available!
- Platform-specific workarounds
  - Docker: Add `Dockerfile` steps
  - CLI tool: Perhaps an install command?
  - Flutter

The xxx_flutter_libs pattern

...

sqlcipher

objectbox

sqlite3_flutter_libs

powersync

isar

...

The xxx_flutter_libs pattern

They're a Dart **package**, but:

- No Dart code
- build scripts (Gradle, SwiftPM, CMake, CocoaPods)
- another package to depend on
- Not available in unit tests

Benefits of `dart:ffi`?

Goal	Platform channels	`dart:ffi`
Fast	🤷	✓
Portable	❌	Platform-specific concerns ❌
Simple	❌	❌
Synchronous	❌	✓
Shared memory	❌	✓

Native Code assets

“Basically C”

```dart
import 'dart:ffi';

@Native<Int Function()>()
external int cool_library_function();

@Native<Int>()
external int useful_field;

void main() {
  var x = cool_library_function();
  useful_field = x * 2;
}
```

```c
extern int cool_library_function();


extern int useful_field;

int main() {
  int x = cool_library_function();
  useful_field = x * 2;
}
```

The distribution problem, revised

- Not just a syntax change!
- Compiler knows about all used functions
  - Preserved during all parts of the compilation process
  - Enables optimizations down the road:
    - static linking
    - native tree-shaking
- We didn’t have to load any libaries!
- So: How is the library made available?

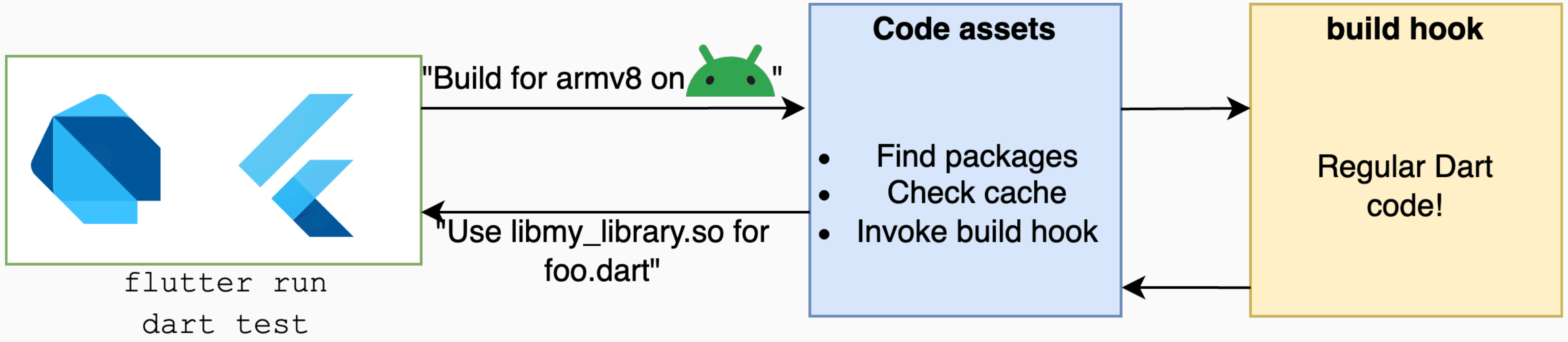
Build hooks

Build hooks

`hook/build.dart`:

```dart
import 'package:hooks/hooks.dart';

void main(List<String> args) async {
  await build(args, (input, output) async {
    // TODO: generate and declare used libraries
  });
}
```

Build hook system



- "Use libmy_library.so for foo.dart"
- my_library.c is an input

What’s in a code asset

- Cached in **`.dart_tool`**
- Attached **asset id**
  - Pointing to Dart file with **@Native** definitions
- All kinds of things (library, executable, …)
- Linking instructions
  - Tools responsible for platform-specific loading schemes!
- Created during build

Build hooks

dart

```dart
await build(args, (input, output) async {
  final config = input.config.code;

  output.assets.code.add(
    CodeAsset(
      package: 'my_library',
      name: 'foo.dart',
      linkMode: DynamicLoadingBundled(),
      file: pathToMyPrebuiltLibraryFile(config),
    ),
  );
});
```

Hooks: Inspirations

Code hooks can:

- compile C sources from assets
- download prebuilt libraries
- download and compile C sources
- use libraries from the operating system
- **let the user decide!**

User flexibility

```yaml
name: my_app
environment:
  sdk: ^3.6.0
dependencies:
  my_library:

hooks:
  my_library:
    native_version: 1.2.0
```

```dart
void main(List<String> args) {
  build(args, (input, output) async {
    final version = input.userDefines['native_version'];
    // ...
  });
}
```

`dart:ffi` with native assets

Goal	Platform channels	`dart:ffi`	Native assets
Fast	🤷‍♂️	✓	✓
Portable	❌	❌	✓
Simple	❌	❌	YES! ✓
Synchronous	❌	✓	✓
Shared memory	❌	✓	✓

Compatibility

Experimental feature

Outside of `main` channel:

```
> dart run example/main.dart
Package(s) [sqlite3] require the native assets feature to be enabled.
Enable native assets with `--enable-experiment=native-assets`.
```

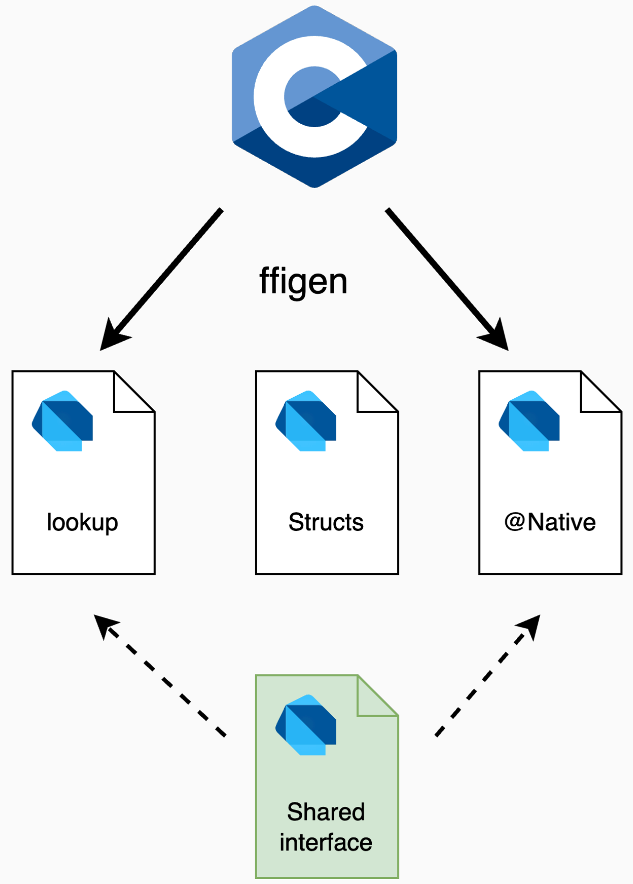
Not great for existing packages!

Adoption in existing packages

During the transition, support both:

1. The `flutter_libs` pattern.
2. Native assets (as a `native_assets` package)

Sharing code



Sharing code: The results

```dart
import 'package:sqlite3/sqlite3.dart';
import 'package:sqlite3_native_assets/sqlite3_native_assets.dart';

void main() {
  Database db = sqlite3Native.openInMemory();
  print(db.select('SELECT 1 + 1'));
}
```

Outlook

- FFI and native assets make integrating native code **amazing**.
- Not just code
- Data assets
- Auto-generated
- Of course, all platforms!

https://www.simonbinder.eu/talks/native_assets/