PUBLICLY SHARED


**Author:** Aaron Clarke
**Created:** 9/28/20
**Last Updated:**11/10/20

# Description

Late 2019, we started working on a solution that made communication between the host-platform and Flutter safer and easier for Flutter Plugins and Add-to-app. The solution that we ended up developing, Pigeon, has since launched and is being used successfully in a handful of projects.

When we started work on Pigeon we scoped down the project to be the smallest amount of effort that would give us the biggest return on investment in order to inform and bootstrap a larger one. Now that Pigeon has been successful in its first phase this document lays out the potential future investments we could make.

# Background

Here is a quick recap of what Pigeon can do today as of version 0.1.9. You can create a Pigeon file, a subset of Dart, which Pigeon will use to generate a typesafe API for sending messages from Flutter to the host-platform, or vice-versa.

```
import 'package:pigeon/pigeon.h'

class AccountId {
  int number;
}

class Account {
  AccountId id;
  string name;
  int balance;
}

@HostApi()
abstract class BankingApi {
```

```
  Account lookupAccount(AccountId account);
}
```

This would generate the following client code:

```
class BankingApi {
  BankingApi() {}
  Future<Account> lookupAccount(AccountId account) async {}
}
```

Generate the following server code (in the host-platform language):

```
abstract class BankingApi {
  static void setupBankingApi(BinaryMessenger messenger, BankingApi api) {}
  Account lookupAccount(AccountId account);
}
```

# Investments

Here is the list of potential investments we could make:

## Asynchronous Handlers

**Note 10/27/21: This has been implemented.**
Pigeon generates interfaces which users implement in order to receive messages from the other platform. Right now those interfaces are synchronous, which makes things easier and safer in most cases since asynchronous handlers can fail to return a result.  Some implementations are better suited for asynchronous results though.  Right now we ask Pigeon users to couple a @FlutterApi with a @HostApi to send and receive asynchronous messages but we could make that built into the API definitions to avoid that overhead.

The Pigeon file could look like this:

```
@HostApi()
abstract class BankingApi {
  @async
  Account lookupAccount(AccountId account);
}
```

Generates this for the client:

```
class BankingApi {
  BankingApi() {}
  Future<Account> lookupAccount(AccountId account) async {...}
}
```

Generates this for the server:

```
abstract class BankingApi {
```

```
  static void setupBankingApi(BinaryMessenger messenger, BankingApi api) {...}
  Future<Account> lookupAccount(AccountId account) async;
}
```

## Generics

**Note 10/27/21: This has been implemented in v1.0.**
Platform Channels don't support Generics currently. You are expected to use generic List objects and cast the items inside of it. We have the information available to Pigeon to support them though. We may be able to implement it in such a way that support is also added to traditional Platform Channels.

## Naked Arguments and Return Values

**Note 10/27/21: This has been implemented in v1.0.**
Right now arguments to Pigeon methods and return values must be contained in a class. You can't use the raw values. If we could it could look like this:

```
@HostApi()
abstract class BankingApi {
  int lookupAccountBalance(int accountNumber);
}
```

## Synchronous Messages

The calling mechanisms in Pigeon are bound by the trapping of Platform Channels. Since Platform Channels don't support synchronous messages (i.e. blocking the UI thread for @HostApi's until execution happens on the platform thread and blocking the platform thread for @FlutterApi's until execution is complete on the isolate), Pigeon doesn't support them either.

There are a few benefits to allowing this:

1. It allows us to interface with platform API's that demand synchronous results. For example on iOS and Android there are calls that happen when an app will be killed by the operating system which expects all work to happen synchronously.
2. It is faster. Based on the benchmarks I performed, the most costly part of sending messages between Flutter and the host-platform is performing thread hops. This eliminates half of that overhead.

A user could create a situation that creates deadlock, but that risk won't be something unique to Pigeon and won't be the first time a developer managed such a risk. It will be opt-in to mitigate this further, it could look like this:

```
@HostApi()
abstract class BankingApi {
  @sync
  int lookupAccountBalance(int accountNumber);
}
```

## Advanced Threading

We could also support the user specifying what thread they want to execute a handler on for HostApi's. It could look like this:

```
@HostApi()
abstract class BankingApi {
  @thread("io")
  Account lookupAccount(AccountId account);
}
```

The builtin threads could be supported: platform, ui, io, raster, etc. Specifying a different name could generate its own thread. This feature could be used in conjunction with Synchronous Messages in order to get any threading setup.

## Resource Tracking Handlers

Right now if a user wants to manage multiple objects with Pigeon it is their responsibility to do the multiplexing for that. For example, a video player plugin could manage multiple active video players. The API might look something like this:

```
class VideoPlayerId {
  int id;
}

class PlayMessage {
  int id;
  String filePath;
}

@HostApi()
abstract class VideoApi {
  VideoPlayerId create();
  void dispose(VideoPlayerId videoPlayer);
  void play(PlayMessage playMessage);
}
```

Notice that the user has to manage their own identity system that sends an integer back and forth between Flutter and the host-platform in order to manage a remote resource from Flutter.

One possible solution might be to allow API's as return types, such as:

```
class PlayMessage {
  String filePath;
}

@HostApi()
```

```
abstract class VideoApi {
  VideoApi create();
  void dispose();
  void play(PlayMessage playMessage);
}
```

The generated server code could look like:

```
abstract class VideoApi {
  void setup(BinaryMessenger messenger, VideoApi api) {}
  void create(int instance);
  void dispose(int instance);
  void play(int instance, PlayMessage playMessage);
}
```

## Custom Codec

Right now Pigeon is built on top of Flutter's Standard Codec which uses reflection to serialize and tagged types to deserialize. Pigeon could generate a custom Codec for each class which could eliminate that runtime cost which would make sending messages faster.

## Build System Integration

Using Pigeon right now requires the manual step of executing Pigeon to generate source code which is then checked into your repository. The problem with this is that a user could edit a Pigeon file and then forget to regenerate the source code. If we integrated with the Flutter build system, we could make sure that doesn't happen.

Dart does have Builders but they are problematic since they require the use of an alternative build system outside of the typical Dart build system. Since all calls to the Dart build system are abstracted away for Flutter users with the Flutter tool it may be possible to use them.

Alternatively, Flutter could add it's own buildstep. There could be a magic script like `prebuild.sh` or an entry in the pubspec `pre_build_script`. The entry could be as explicit as "`pigeons:`".

If we had build system integration we could potentially hide the generated source code. Some asked for this in the beginning. I'm not a fan of this since it can be confusing to be importing and using source code you can't easily inspect.

## Dart Compiler Integration

Since Pigeon is a subset of Dart, it could live next to the other Dart code in a Flutter project, in the same source files. If the Dart compiler allowed us to manipulate the Abstract Syntax Tree that is generated from the a Dart file, we could insert all the generated Pigeon code there.

Unfortunately this would only make things cleaner on the Flutter side. We still would need to generate host-platform code.

## Software Transactional Memory

Potentially we'd want to share a large amount of memory between the host and Flutter. So much memory that it would be an impediment to send it back and forth between them. We could instead create an STM that allows us to share data between the platforms and only pay the cost for copying data that is accessed. The interface to the STM could be wrapped up in the channel API.

Here's an example of what it might look like:

Pigeon:

```
@STM()
class Person {
  String name;
  int balance;
  int accountNumber;
}
```

Host:

```
- (void)loadPerson {
  PersonSTM* stm = [PersonSTM instance];
  Person* person = [[Person alloc] init];
  person.name = @"aaron";
  person.accountId = ++_count;
  person.balance = 0;
  [stm store:person];
}
```

Flutter:

```
class PersonWidget extends StatefulWidget {
  PersonWidget({Key key, this.accountNumber}) : super(key: key);
  final int accountNumber;
  @override
  _PersonState createState() => _PersonState();
}

class _PersonState extends State<PersonWidget> {
  @override
  Widget build(BuildContext context) {
    PersonSTM stm = PersonSTM.instance;
    Person person = stm.queryAccountNumber(widget.accountNumber).first;
    return Column(
      children:[
        Text('name: ${person.name}'),
        Text('balance: ${person.balance}'),
```

```
        Text('accountNumber: ${person.accountNumber}'),
        Button(text:'add doller', onPressed:(){
          person.balance += 100;
          setState((){
            stm.store(person);
          });
        });
      ],
    );
  }
}
```

## Protobuf Front-End

The Pigeon compiler is split between a front-end and a back-end.  The current front-end being driven by Dart parser.  We could alternatively create a front-end using the protoc compiler.  This would allow users to take advantage of all the Pigeon features if they are already using Protobufs in their app.