Introducing Dart 3 to Plus Plugins

SUMMARY

This document is for introducing Dart 3 to Plus Plugins, primarily refactoring Plus packages to utilize new Dart 3 language features.

OBJECTIVE

Help optimize Plus Plugins using Dart 3 features such as 100% null safety, records, patterns, interoperability using JNIGen and FFIGen, class modifiers etc. Additionally, examine and scrutinize the new language features that will benefit packages by creating a new API (possible) or refactoring internal coding so that the package can take full advantage of Dart 3.

BACKGROUND

Plus Plugin so far has a lot of platform specific implementation and native calls to platforms using platform interfaces. Dart 3 supports an interoperability feature for performing native calls using JNIGen and FFIGen packages. Also Dart 3 ensures 100% null safety which helps prevent errors that result from unintentional access of variables set to null. There are several other features such as Records, pattern matching, class modifiers which potentially helps to improve code readability and build better API design for packages.

Upgrading the Plus Plugin to Dart 3 opens up new language capabilities that might assist to improve packages and perhaps establish a new API or restructure internal functionality to enable the package to take full advantage of Dart 3 features.

GLOSSARY

- **JNI -** Java Native Interface
- **FFI -** Foreign Function Interface
- **JNIGen** - It analyzes compiled JAR files or Java source code to produce an API description, which is subsequently utilized to generate Dart and C

bindings[1].

- **FFIGen** - This bindings generator is applicable for invoking C code, or code from an alternative language that compiles into C modules conforming to the C calling convention. Examples of such languages encompass Go or Rust. In experimental mode it also enables Objective-C headers for Swift APIs[2].

TABLE OF CONTENT

SUMMARY **1**
OBJECTIVE **1**
BACKGROUND **1**
Glossary 1
TABLE OF CONTENT **2**
OVERVIEW **3**
Non-goals 3
Patterns 3
Records 4
Sealed Class 6
Interoperability 8
DETAILED REFACTORING **8**
Battery_plus 8
Connectivity_plus 12
Device_info_plus 13
Network_info_plus 15
Package_info_plus 16
Sensors_plus 18
Share_plus 18
TESTING PLAN **19**
REFERENCE **19**

OVERVIEW

Non-goals

Our refactoring approach does not involve imposing Dart 3 features forcefully onto Plus Plugins. We prioritize code readability and try to optimize code effectively.

The Proposal is to inculcate Dart 3 features in Plus Plugin which make the packages enabled with null safety and improves the code readability throughout the package using records, patterns and class modifiers. Not only code readability it also ensures 100% null safety. Null safety also prevents some types of code errors, such as null pointer exceptions. It also enables our compilers and runtimes to optimize code in ways that would be impossible without null safety.

Patterns

Patterns in Dart 3 are a new feature that allow you to destructure composite data into its constituents. They can be used in a variety of places, including: switch statements, for loops, if statements, function parameters, function return values[3]. We can introduce pattern matching for removing redundant switch case break statements or directly mapping a JSON with a specific structure. Plus plugins contain multiple switch cases where patterns can be introduced to make code more concise and readable.

The below code is a switch case from the method channel implementation of `share_plus` package's platform interface. The below code can be made more readable by eradicating multiple case, return and default keywords. Here pattern matching can directly be utilized.

Before:

```
static ShareResultStatus _statusFromResult(String result) {
    switch (result) {
      case '':
        return ShareResultStatus.dismissed;
      case 'dev.fluttercommunity.plus/share/unavailable':
        return ShareResultStatus.unavailable;
      default:
        return ShareResultStatus.success;
    }
}
```

After:

```
static ShareResultStatus _statusFromResult(String result) {
  return switch (result) {
    '' => ShareResultStatus.dismissed,
    'dev.fluttercommunity.plus/share/unavailable' =>
```

```
        ShareResultStatus.unavailable,
    _ => ShareResultStatus.success,
  };
}
```

There are many places where dart 3 patterns can be utilized to improve the readability of switch cases throughout the Plus Plugins.

Records

Records in Dart 3 are a new type that allow you to bundle multiple values into a single object. They are similar to classes in that they can have fields, but they are immutable and have no constructors. Records are also structurally typed, which means that the type of a record is determined by the types of its fields[4]. Records can be used to remove extra functions for serialization, while we can return multiple typed data using the records feature.

The below code is from the `package_info_plus` package describing the packageInfo object. And the function fromPlatfrom returns the packageInfo object's as return value.

Before:

```
  static Future<PackageInfo> fromPlatform() async {
    if (_fromPlatform != null) {
      return _fromPlatform!;
    }

    final platformData = await PackageInfoPlatform.instance.getAll();
    _fromPlatform = PackageInfo(
      appName: platformData.appName,
      packageName: platformData.packageName,
      version: platformData.version,
      buildNumber: platformData.buildNumber,
      buildSignature: platformData.buildSignature,
      installerStore: platformData.installerStore,
    );
    return _fromPlatform!;
  }

  @override
  String toString() {
```

```
    return 'PackageInfo(appName: $appName, buildNumber: $buildNumber,
packageName: $packageName, version: $version, buildSignature:
$buildSignature, installerStore: $installerStore)';
  }
```

Here we can introduce records and remove the serialization function toString which converts packageInfo object to string. Instead we can return a record containing both serialized string and packageInfo object.

After:

```
static (Future<PackageInfo>, String) fromPlatform() async {
    if (_fromPlatform != null) {
      return _fromPlatform!;
    }

    final platformData = await PackageInfoPlatform.instance.getAll();
    _fromPlatform = PackageInfo(
      appName: platformData.appName,
      packageName: platformData.packageName,
      version: platformData.version,
      buildNumber: platformData.buildNumber,
      buildSignature: platformData.buildSignature,
      installerStore: platformData.installerStore,
    );
    String packageInfoSerialized = 'PackageInfo(appName:
${platformData.appName}, buildNumber: ${platformData.buildNumber},
packageName: ${platformData.packageName}, version: ${platformData.version},
buildSignature: ${platformData.buildSignature}, installerStore:
${platformData.installerStore})'
    return (_fromPlatform!, packageInfoSerialized);
  }
```

Similarly, serialization is performed in the `device_info_platform_interface` for the `device_info` object which also can be refactored.

Sealed Class

Sealed class[5] can be used to identify the device specific implementation. Currently, in `device_info_plus` the user needs to check the platform and then fetch respective platforms data. Instead we can cover the device specific

implementation inside the package and identify the platform and provide the device info directly to the user.

Thus to implement this we can use a sealed class for devices and implement it to each of the devices class. Each device specific class contains device specific implementation and then with the use of patterns and switch cases we can return the device data according to the platform.

Before:

```
  Future<AndroidDeviceInfo> get androidInfo async =>
      _cachedAndroidDeviceInfo ??=
          AndroidDeviceInfo.fromMap((await _platform.deviceInfo()).data);

  Future<IosDeviceInfo> get iosInfo async => _cachedIosDeviceInfo ??=
      IosDeviceInfo.fromMap((await _platform.deviceInfo()).data);

  Future<LinuxDeviceInfo> get linuxInfo async => _cachedLinuxDeviceInfo ??=
      await _platform.deviceInfo() as LinuxDeviceInfo;

  Future<WebBrowserInfo> get webBrowserInfo async =>
      _cachedWebBrowserInfo ??= await _platform.deviceInfo() as
WebBrowserInfo;

  Future<MacOsDeviceInfo> get macOsInfo async => _cachedMacosDeviceInfo ??=
      MacOsDeviceInfo.fromMap((await _platform.deviceInfo()).data);

  Future<WindowsDeviceInfo> get windowsInfo async =>
      _cachedWindowsDeviceInfo ??=
          await _platform.deviceInfo() as WindowsDeviceInfo;



  /// Returns device information for the current platform.
  Future<BaseDeviceInfo> get deviceInfo async {
    if (kIsWeb) {
      return webBrowserInfo;
    } else {
      if (Platform.isAndroid) {
        return androidInfo;
      } else if (Platform.isIOS) {
        return iosInfo;
      } else if (Platform.isLinux) {
```

```
      return linuxInfo;
    } else if (Platform.isMacOS) {
      return macOsInfo;
    } else if (Platform.isWindows) {
      return windowsInfo;
    }
  }
```

After: Using Sealed class Dart 3 Feature

```
sealed class Device {}

class Android implements Device {}

class IOS implements Device {}

class Linux implements Device {}

class Web implements Device{}

class MacOS implements Device{}

class Windows implements Device{}
```

Using dart 3 pattern matching feature to map sealed classes

```
Future<BaseDeviceInfo?> deviceInfo() async {
    return match (device!) {
      Android() => await Android().getInfo(_platform, convertToMap),
      IOS() => await IOS().getInfo(_platform, convertToMap),
      Linux() => await Linux().getInfo(_platform, convertToMap),
      Web() => null,
      MacOS() => null,
      Windows() => null,
    };
  }
```

Sealed class implementation in device info plugin: [code example]

Interoperability

Dart language interoperability can be used for calling platform libraries directly from Dart, using tools and packages like FFI, JNI to access native information using Dart class. As Plus Plugin interacts with native calls frequently in all of the packages

we can use JNI and JNIGen[6] to implement Android specific kotlin and java implementation, FFI and FFIGen[2] for IOS and macOs specific Swift and objective C implementation to make use of native libraries directly from Dart classes. Here, as an example, Battery Plus Plugin uses native batterymanager package for fetching information regarding battery for android. Thus we can use JNI and JNIGen to access the native functionality of the battery manager using Dart classes.

DETAILED REFACTORING

Battery_plus

The `battery_plus` plugins consist of battery state utils which returns the enums representing the battery state such as full, charging, discharging etc. Thus the switch case being used to describe the battery state can be upgraded to a new dart 3 switch case which removes redundant case and return statements. The below code demonstrates the usage of pattern matching and switch case.

Before:

```
/// Method for parsing battery state.
BatteryState parseBatteryState(String state) {
  switch (state) {
    case 'full':
      return BatteryState.full;
    case 'charging':
      return BatteryState.charging;
    case 'discharging':
      return BatteryState.discharging;
    case 'unknown':
      return BatteryState.unknown;
    default:
      throw ArgumentError('$state is not a valid BatteryState.');
  }
}
```

After:

```
BatteryState parseBatteryState(String state) {
  return match (state) {
    'full' => BatteryState.full,
```

```
    'charging' => BatteryState.charging,
    'discharging' => BatteryState.discharging,
    'unknown' => BatteryState.unknown,
    _ => throw ArgumentError('$state is not a valid BatteryState.'),
  };
}
```

Here, we can also introduce the interoperability feature of dart 3, using JNI and JNIGen for making native calls and using native libraries. As in `battery_plus` we use the battery manager package to get battery info of android. Thus by exposing the battery manager package to JNI we can utilize the battery manager info. The below attached commit represents the live implementation for JNIGen converting kotlin code to dart class and utilizing it. For IOS specific native calls we can use FFIGen to access the IOS native libraries.

Before: Currently method channel implementation is used for Android.

```
class MethodChannelBattery extends BatteryPlatform {
  /// The method channel used to interact with the native platform.
  @visibleForTesting
  MethodChannel methodChannel =
      const MethodChannel('dev.fluttercommunity.plus/battery');

  /// The event channel used to receive BatteryState changes from the
native platform.
  @visibleForTesting
  EventChannel eventChannel =
      const EventChannel('dev.fluttercommunity.plus/charging');

  Stream<BatteryState>? _onBatteryStateChanged;

  /// Returns the current battery level in percent.
  @override
  Future<int> get batteryLevel => methodChannel
      .invokeMethod<int>('getBatteryLevel')
      .then<int>((dynamic result) => result);

  /// Returns true if the device is on battery save mode
  @override
  Future<bool> get isInBatterySaveMode => methodChannel
      .invokeMethod<bool>('isInBatterySaveMode')
      .then<bool>((dynamic result) => result);
```

```
  /// Returns the current battery state in percent.
  @override
  Future<BatteryState> get batteryState => methodChannel
      .invokeMethod<String>('getBatteryState')
      .then<BatteryState>((dynamic result) => parseBatteryState(result));

  /// Fires whenever the battery state changes.
  @override
  Stream<BatteryState> get onBatteryStateChanged {
    _onBatteryStateChanged ??= eventChannel
        .receiveBroadcastStream()
        .map((dynamic event) => parseBatteryState(event));
    return _onBatteryStateChanged!;
  }
}
```

After: Now we can use dart class(BatteryPlusAndroidPlugin) for Android

```
/// The Android implementation of BatteryPlatform.
class BatteryPlusAndroidPlugin extends BatteryPlatform {
  /// Register this dart class as the platform implementation for Android
  static void registerWith() {
    BatteryPlatform.instance = BatteryPlusAndroidPlugin();
  }

  /// Returns the current battery level in percent.
  @override
  Future<int> get batteryLevel{
    JObject activity = JObject.fromRef(Jni.getCurrentActivity());
    BatteryPlusPlugin batteryPlusPlugin = BatteryPlusPlugin(activity);
    return
Future.value(int.parse(batteryPlusPlugin.getBatteryLevel().toDartString()))
;
  }

  /// Returns the current battery state.
  @override
  Future<BatteryState> get batteryState {
    JObject activity = JObject.fromRef(Jni.getCurrentActivity());
    BatteryPlusPlugin batteryPlusPlugin = BatteryPlusPlugin(activity);
    return
Future.value(parseBatteryState(batteryPlusPlugin.getBatteryStatus().toDartS
```

```
tring()));
  }

  /// Returns true if the device is on battery save mode
  Future<bool> get isInBatterySaveMode {
    JObject activity = JObject.fromRef(Jni.getCurrentActivity());
    BatteryPlusPlugin batteryPlusPlugin = BatteryPlusPlugin(activity);
    return
Future.value(batteryPlusPlugin.isInPowerSaveMode().booleanValue());
  }

  /// Fires whenever the battery state changes.
  @override
  Stream<BatteryState> get onBatteryStateChanged {
  }
}
```

File: -

1. *packages/battery_plus/battery_plus_platform_interface/lib/src/utils.dart*

JNIGen implementation of android dart class in battery Plus Plugin [code example(Commit)].

Connectivity_plus

The `connectivity_plus` plugin interacts with web specific implementation where the effective connectivity result is mapped again to the their respective enums and similar fashion of implementation is carried out for type of connectivity. Moreover, the platform interface of `connectivity_plus` plugin deals with connectivity results which indirectly performs switch cases. As a result, we can use the dart 3 patterns and switch case feature in exactly the same manner as `battery_plus` does.

Before:

```
ConnectivityResult _effectiveTypeToConnectivityResult(String effectiveType)
{
  // Possible values:
  /*'2g'|'3g'|'4g'|'slow-2g'*/
  switch (effectiveType) {
    case 'slow-2g':
    case '2g':
```

```
    case '3g':
    case '4g':
      return ConnectivityResult.mobile;
    default:
      return ConnectivityResult.wifi;
  }
}

ConnectivityResult _typeToConnectivityResult(String type) {
  // Possible values:
/*'bluetooth'|'cellular'|'ethernet'|'mixed'|'none'|'other'|'unknown'|'wifi'
|'wimax'*/
  switch (type) {
    case 'none':
      return ConnectivityResult.none;
    case 'bluetooth':
      return ConnectivityResult.bluetooth;
    case 'cellular':
    case 'mixed':
    case 'other':
    case 'unknown':
      return ConnectivityResult.moble;
    case 'ethernet':
      return ConnectivityResult.ethernet;
    default:
      return ConnectivityResult.wifi;
  }
}
```

After:

```
ConnectivityResult _effectiveTypeToConnectivityResult(String effectiveType)
{
  // Possible values:
  /*'2g'|'3g'|'4g'|'slow-2g'*/
  return match (effectiveType) {
    'slow-2g', '2g', '3g',
    '4g' => ConnectivityResult.mobile,
    _ => ConnectivityResult.wifi,
  };
}

ConnectivityResult _typeToConnectivityResult(String type) {
  // Possible values:
```

```
/*'bluetooth'|'cellular'|'ethernet'|'mixed'|'none'|'other'|'unknown'|'wifi'
|'wimax'*/
  return match (type) {
    'none' => ConnectivityResult.none,
    'bluetooth' => ConnectivityResult.bluetooth,
    'cellular', 'mixed', 'other', 'unknown' => ConnectivityResult.mobile,
    'ethernet' => ConnectivityResult.ethernet,
    _ => ConnectivityResult.wifi,
  };
}
```

File: -

1. *packages/connectivity_plus/connectivity_plus/lib/src/web/utils/connectivity_result.dart*
2. *packages/connectivity_plus/connectivity_plus_platform_interface/lib/src/utils.dart*

File difference for the platform interface of connectivity Plus Plugin Code Example

Device_info_plus

`Device_info_plus` plugins fetches data for each device using the if-else syntax over the platforms. Here dart 3 patterns and switch case can be introduced to eradicate the redundant if-else conditions and match the platforms which make the code more concise and readable.

Before:

```
Future<BaseDeviceInfo> get deviceInfo async {
    if (kIsWeb) {
      return webBrowserInfo;
    } else {
      if (Platform.isAndroid) {
        return androidInfo;
      } else if (Platform.isIOS) {
        return iosInfo;
      } else if (Platform.isLinux) {
        return linuxInfo;
      } else if (Platform.isMacOS) {
        return macOsInfo;
      } else if (Platform.isWindows) {
        return windowsInfo;
      }
```

```
    }
```

After:

```
Future<BaseDeviceInfo> get deviceInfo async {
  if (kIsWeb) {
    return webBrowserInfo;
  } else {
    return switch (defaultTargetPlatform) {
      TargetPlatform.android => androidInfo,
      TargetPlatform.iOS => iosInfo,
      TargetPlatform.linux => linuxInfo,
      TargetPlatform.windows => macOsInfo,
      TargetPlatform.macOS => windowsInfo,
    };
  }
}
```

Similarly in the `device_info_plus` platform interface, tests can be simplified by using the switch case and patterns.

File: -

1. *packages/device_info_plus/device_info_plus/lib/device_info_plus.dart*
2. *packages/device_info_plus/device_info_plus_platform_interface/test/method_channel_device_info_test.dart*

Network_info_plus

The `network_info_plus` plugin also provides accessibility to location features, as a prerequisite we need to check location authorization. And for matching the location authorization status to their respective enums switch case is used. Also in the tests for creating a fake instance we set the method channel call handler using different functional cases where the new switch case can be introduced to improve code readability. Thus, we can modify the switch case to the latest dart 3 switch case and use pattern matching to match the enum type.

Before:

```
/// Convert a String to a LocationAuthorizationStatus value.
LocationAuthorizationStatus parseLocationAuthorizationStatus(String?
result) {
  switch (result) {
```

```
    case 'notDetermined':
      return LocationAuthorizationStatus.notDetermined;
    case 'restricted':
      return LocationAuthorizationStatus.restricted;
    case 'denied':
      return LocationAuthorizationStatus.denied;
    case 'authorizedAlways':
      return LocationAuthorizationStatus.authorizedAlways;
    case 'authorizedWhenInUse':
      return LocationAuthorizationStatus.authorizedWhenInUse;
    default:
      return LocationAuthorizationStatus.unknown;
  }
}
```

After:

```
LocationAuthorizationStatus parseLocationAuthorizationStatus(String?
result) {
  return match (result) {
    'notDetermined' => LocationAuthorizationStatus.notDetermined,
    'restricted' => LocationAuthorizationStatus.restricted,
    'denied' => LocationAuthorizationStatus.denied,
    'authorizedAlways' => LocationAuthorizationStatus.authorizedAlways,
    'authorizedWhenInUse' =>
LocationAuthorizationStatus.authorizedWhenInUse,
    _ => LocationAuthorizationStatus.unknown,
  };
}
```

File: -

1. *network_info_plus_platform_interface/lib/src/utils.dart*
2. *network_info_plus_platform_interface/test/method_channel_network_info_test.dart*

Package_info_plus

`Package_info_plus` plugins provides numerous information about packages, which leads to an object with information, to serialize the data toString function is used. Instead we can use records by removing the serialization function.

Before:

```
static Future<PackageInfo> fromPlatform() async {
    if (_fromPlatform != null) {
      return _fromPlatform!;
    }

    final platformData = await PackageInfoPlatform.instance.getAll();
    _fromPlatform = PackageInfo(
      appName: platformData.appName,
      packageName: platformData.packageName,
      version: platformData.version,
      buildNumber: platformData.buildNumber,
      buildSignature: platformData.buildSignature,
      installerStore: platformData.installerStore,
    );
    return _fromPlatform!;
  }

  @override
  String toString() {
    return 'PackageInfo(appName: $appName, buildNumber: $buildNumber,
packageName: $packageName, version: $version, buildSignature:
$buildSignature, installerStore: $installerStore)';
  }
```

After:

```
static (Future<PackageInfo>, String) fromPlatform() async {
    if (_fromPlatform != null) {
      return _fromPlatform!;
    }

    final platformData = await PackageInfoPlatform.instance.getAll();
    _fromPlatform = PackageInfo(
      appName: platformData.appName,
      packageName: platformData.packageName,
      version: platformData.version,
      buildNumber: platformData.buildNumber,
      buildSignature: platformData.buildSignature,
      installerStore: platformData.installerStore,
    );
    String packageInfoSerialized = 'PackageInfo(appName:
${platformData.appName}, buildNumber: ${platformData.buildNumber},
packageName: ${platformData.packageName}, version: ${platformData.version},
buildSignature: ${platformData.buildSignature}, installerStore:
```

```
${platformData.installerStore})'
    return (_fromPlatform!, packageInfoSerialized);
  }
```

`Package_info_plus` plugin uses platform specific implementation for getting package info. That is for android and IOS we make native calls. Thus these native calls can be replaced by using JNIGen and FFIGen dart 3 features. Using JNIGen we can access the android native libraries and using FFIGen we can access the IOS native libraries which can remove/replace the native koltin/objective C implementation of android and IOS platform. This can be implemented similarly as the `battery_plus` plugin.

Moreover, Package Info has a similar denomination such as battery plus, but using the interoperability feature of Dart 3 we can use JNIGen and FFIGen to implement native android and IOS calls. We can refactor the existing method channel call by generating dart classes using JNIGen and FFIGen for android and IOS.
File: -

1. *packages/package_info_plus/package_info_plus/lib/package_info_plus.dart*

Sensors_plus

This package uses native android and IOS calls in order to fetch sensor data. Method channels and event channels can be replaced by using JNIGen to generate dart classes while exposing the native libraries/package to dart class. Hence, similar to battery plus here we can refactor the existing method channel calls to dart class using JNI and JNIGen, FFI and FFIGen.
File: -

1. *packages/sensors_plus/sensors_plus/android/src/main/kotlin/dev/fluttercommunity/plus/sensors/SensorsPlugin.kt*

Share_plus

Newer switch cases with pattern matching can be introduced in the method channel implementation.
Before:

```
static ShareResultStatus _statusFromResult(String result) {
    switch (result) {
      case '':
        return ShareResultStatus.dismissed;
      case 'dev.fluttercommunity.plus/share/unavailable':
        return ShareResultStatus.unavailable;
      default:
        return ShareResultStatus.success;
    }
  }
```

After:

```
ShareResultStatus _statusFromResult(String result) {
  return match (result) {
    '' => ShareResultStatus.dismissed,
    'dev.fluttercommunity.plus/share/unavailable' =>
ShareResultStatus.unavailable,
    _ => ShareResultStatus.success,
  };
}
```

File: -

1. *packages/share_plus/share_plus_platform_interface/lib/method_channel/method_channel_share.dart*

TESTING PLAN

Standard unit testing protocols will be employed to validate the correct functionality of all components implemented utilizing Dart 3 features.

REFERENCE

[1] jnigen readme. GitHub. (n.d.).
https://github.com/dart-lang/jnigen/blob/main/README.md
[2] Ffigen: Dart Package. Dart packages. (2023, July 24).
https://pub.dev/packages/ffigen
[3] Patterns. Dart. (n.d.). https://dart.dev/language/patterns
[4] Records. Dart. (n.d.-b). https://dart.dev/language/records
[5] Class modifiers. Dart. (n.d.-a). https://dart.dev/language/class-modifiers#sealed
[6] Jnigen: Dart Package. Dart packages. (2023b, May 31).
https://pub.dev/packages/jnigen