Deferred Components

Deferred components is a set of features and tooling that allows developers to write dart code that can be AOT compiled into separate split dynamic libraries as well as package them together with assets into runtime downloadable modules on Android. The primary goal of deferred components is to reduce install apk size as well as reduce storage space taken up by the app by omitting features the user may not use.

This feature is currently an experimental/preview feature that is only available on Android. It takes advantage of Android and Google Play Store's dynamic feature modules to deliver the deferred components packaged as Android modules. This does not impact other platforms, which continue to build as normal with all deferred components and assets included at initial install time.

Though modules can be defer loaded, the entire application must be completely built and uploaded as a single Android App Bundle (`.aab`). Dispatching partial updates without re-uploading new Android App Bundles for the entire application (eg, code push) is not supported.

This feature only does deferred loading in release and profile mode. In debug mode, all deferred components are present at launch and will instantly load.

A step by step guide on how to integrate deferred components with your app can be found on the Flutter.dev documentation.

APK size gallery case study

Flutter Gallery implements deferred components for the crane study. Here, we will examine the initial install size gains in a fully deferred Flutter gallery branch (branch not kept up to date with master) where we implement deferred components for all studies and demos (not just crane). In this example, we have moved all the splittable assets and fonts into deferred components. Compared to a non-deferred components application, the initial installed APK file size breakdown is as follows:

Deferred components:

- base-arm64_v8a.apk - 12,325,372 bytes
- base-master.apk - 37,889,309 bytes
- Total deferred components initial installation size: 50,214,681 bytes

Non-deferred components (regular app):

- base-arm64_v8a.apk - 12,521,900 bytes
- base-master.apk - 80,605,796 bytes
- Total regular initial installation size: 93,127,696 bytes

Here, we can see a ~200kB decrease in compiled dart code size (base-arm64_v8a.apk) and a ~43mB decrease in assets (base-master.apk) on initial download. Overall, there is a 46% reduction in initial installation size. The dart code, assets, and google fonts are instead moved into separate components downloaded at runtime only when needed:

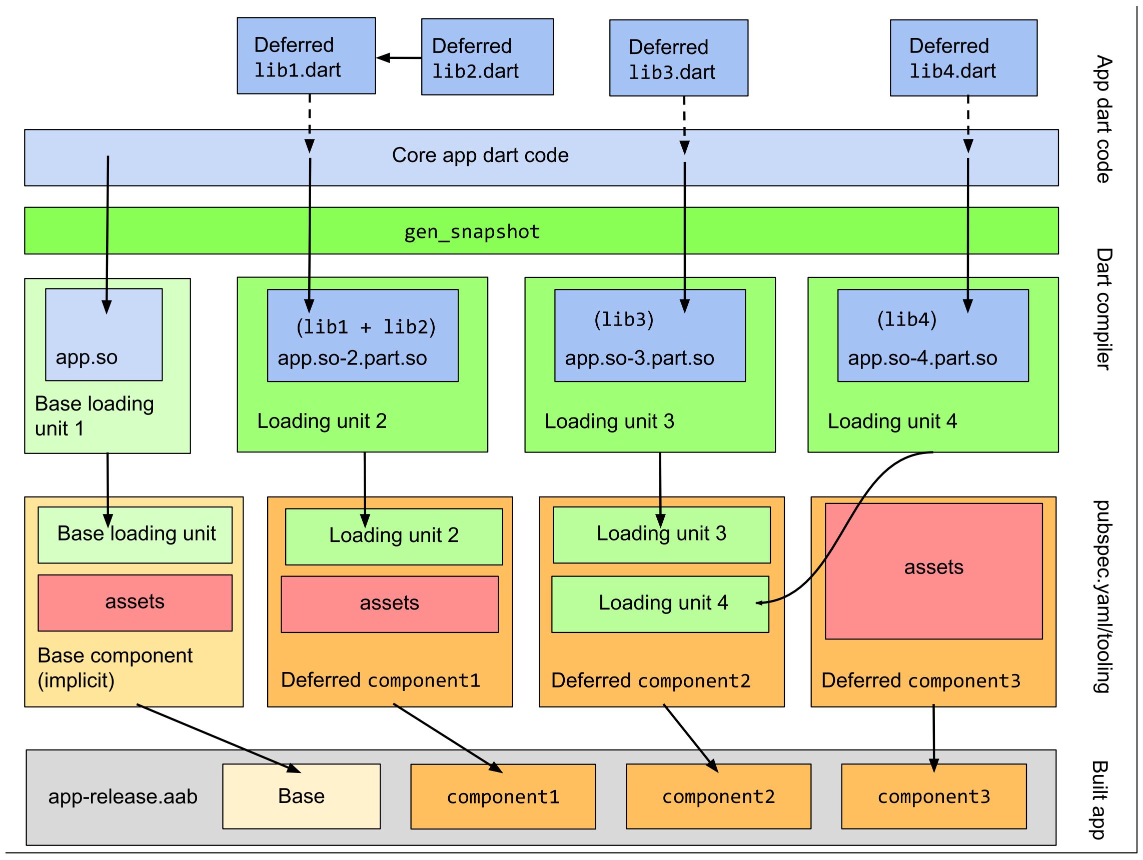
- Crane
- Fortnightly
- Rally
- Shrine
- Cupertino
- Material
- Reference

The total installed size with all components installed is slightly larger than the non-deferred app, but this increase is on the order of a few kb due to overhead in the dart shared libraries from ELF headers and cross-loading-unit calls.

Deferred components app structure

The deferred Dart libraries are interpreted by `gen_snapshot` (Dart's compiler) to produce "loading units", each of which are output as a split AOT shared library (`.so` file) when building in profile or release mode. A loading unit is the smallest set of libraries that are imported exclusively with the deferred keyword by the main code and can be split off of the base libraries.

The following diagram shows an example app structure and a "lifecycle" of how deferred dart libraries are compiled into loading units and packaged into a `.aab` file.

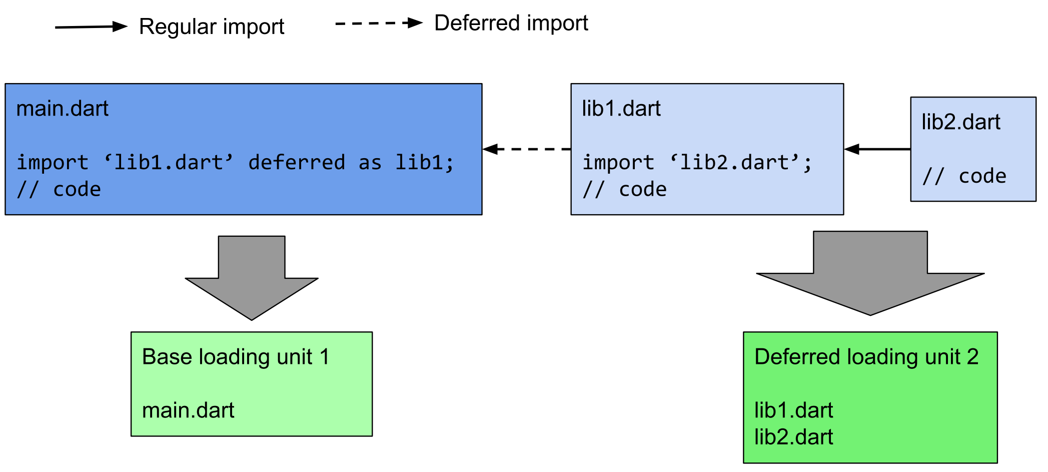


This example app has the following properties:

- Four Dart libraries, with Dart libraries `lib1` and `lib2` dependent on each other. `lib1`, `lib3`, and `lib4` are imported in the flutter app's main code as deferred.
- Four loading units, with one being the base loading unit with id 1 and loading unit 2 containing both `lib1` and `lib2`. Loading units 3 and 4 contain `lib3` and `lib4` respectively.
- Three defined deferred components, plus an implicit base component. `component1` contains loading unit 2 and assets. `component2` contains both loading units 3 and 4 and no assets. `component3` is an assets only component.
- `app-release.aab` is the completed build output file, and contains all three components as well as the base component.

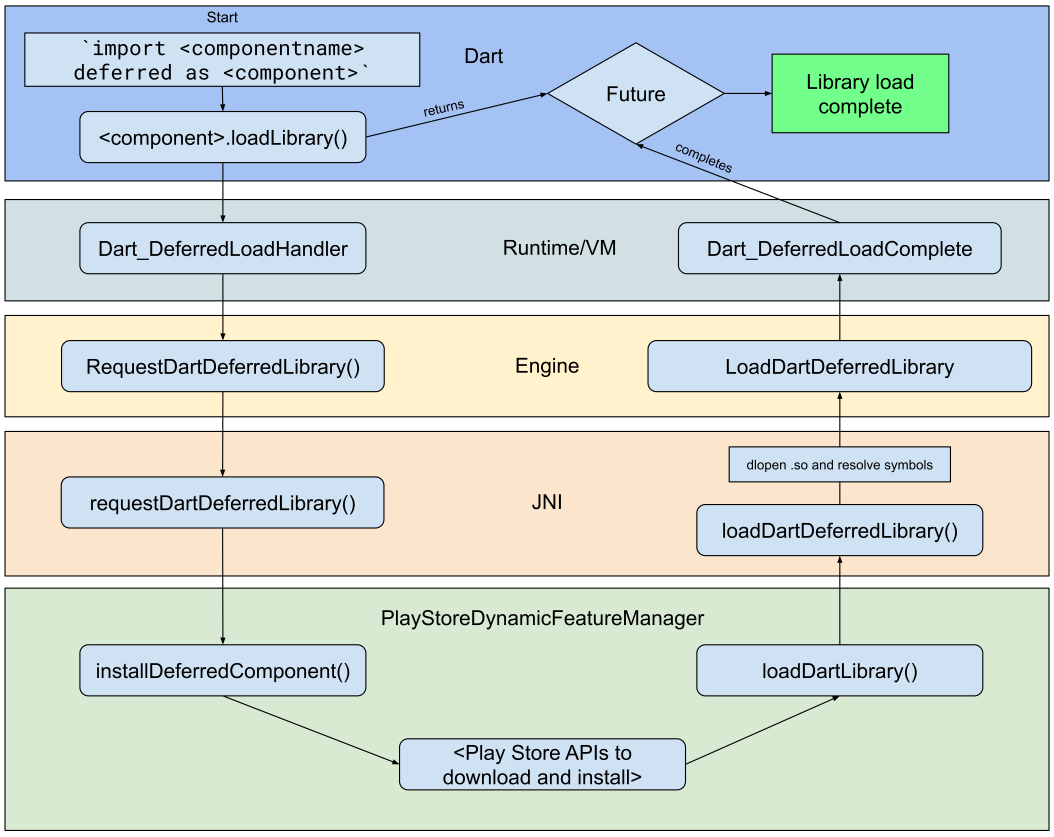
There will always be an implicit base loading unit that contains core flutter packages as well as your base application code. Any libraries that are not fully imported as deferred by your base app code will be included in the base loading unit. If no additional loading units other than base are generated, it likely means you imported your files incorrectly.

Multiple Dart libraries are compiled as a single loading unit if they import each other eagerly (non-deferred):



Lifecycle of a `loadLibrary()` call

Deferred components are primarily triggered to be downloaded, installed, and loaded via the `loadLibrary()` dart call. This call is handled differently in dart2js vs aot/native. Here, we will trace how the `loadLibrary()` call is translated into an installation of a deferred component:



Start
`import <componentname> deferred as <component>`
Dart
<component>.loadLibrary()
returns
Future
Library load complete
completes
Dart_DeferredLoadHandler
Runtime/VM
Dart_DeferredLoadComplete
RequestDartDeferredLibrary()
Engine
LoadDartDeferredLibrary
requestDartDeferredLibrary()
JNI
dlopen .so and resolve symbols
loadDartDeferredLibrary()
PlayStoreDynamicFeatureManager
installDeferredComponent()
loadDartLibrary()
<Play Store APIs to download and install>

The `loadLibrary()` dart call's native side implementation calls a `Dart_DeferredLoadHandler` callback that is set using `Dart_SetDeferredLoadHandler` in `DartIsolate::Initialize`. Dart internally retrieves the loading unit ID assigned to the library and passes it to the callback. The callback is implemented as `DartIsolate::OnDartLoadLibrary`.

The loading unit ID is then passed on through the runtime controller, engine shell and platform view until it passes into the FlutterJNI in the Android embedder. Here, the loading unit ID is passed into the `DeferredComponentsManager` 's `installDeferredComponent` where the ID is converted from an integer to a String name identifying the pubspec-defined deferred component the requested library belongs to. This conversion is handled by a AndroidManifest metadata mapping that is created and verified during the build phase.

`PlayStoreDeferredComponentManager` then invokes the play store split compat APIs to download the android module. Once the Android module is installed, the manager locates the .so files and passes the paths to the engine to `dlopen` . The engine then passes the symbols to the runtime and and dart isolate, which is able to load the symbols into the dart VM. The loading must be associated with a loading unit ID or the load will not complete the Dart Future returned by `loadLibrary()` .

Keep in mind that multiple loading units may be contained in a deferred component, but a `loadLibrary` call will only load the dart symbols from the specific dart library the call was made from. Each loading unit must have separate `loadLibrary` calls before use. Subsequent `loadLibrary` calls on components that are already downloaded should complete much faster, however, the loading can never happen synchronously and there will be at least one frame between call and completion.

Installation by deferred component string name

We also provide a framework-side DeferredComponent utility class that allows direct installation via deferred component name as a string.

This installation path may be used for two purposes:

- Installing asset-only deferred components that do not have any Dart code to call `loadLibrary()` on.
- Pre-downloading components to use later. However, `loadLibrary()` must still be called in order to use any dart code from the pre-downloaded component. This is useful when the exact dart library needed is not known yet.

The direct API uses platform channels to directly invoke the `installDeferredComponent` method on the `DynamicFeatureManager` and will not trigger any of the dart code packed in the component to load due to lack of a specified loading unit. Assets will be loaded. To use any dart code, `loadLibrary()` must still be called.

Uninstallation

The DeferredComponent framework utility class also provides an `uninstallDeferredComponent` method that uses platform channels to request that the OS uninstall and remove the files associated with the specified deferred component. Uninstallation is dependent on how the platform handles it and in Android's case, the removal of the files is queued and may take a long time before actually executed.

Uninstallation may only be requested directly with the string name of the component to be uninstalled. Uninstallation by loading unit id or direct call on a dart import is not yet supported.

Tooling

Deferred components must be built as Android App Bundles (`.aab`) to function. If built as debug or an apk file, dart will compile normally and produce a single `.so` file.

Deferred components makes use of the existing `$ flutter build appbundle` command and checks for the existence of the `deferred-components:` entry in the `pubspec.yaml` to decide whether to build deferred or not. When the app is opted-in to deferred components and the build mode is release, `gen_snapshot` is passed a `--loading_unit_manifest` path, which tells `gen_snapshot` to attempt to produce split AOT artifacts. This includes a base file as well as a `.so` for every deferred library in the codebase. Each of these split units is called a "loading unit" and is assigned an internal integer ID called the loading unit ID.

The build process also relies on project setup to function. Each deferred component must correspond to an Android module defined in the `android` directory of your app. The base module is build as the `app` module while each additional component should have a module with the same name as the component. The base module `AndroidManifest.xml` also needs to include the mapping between loading unit IDs and deferred components.

The `flutter build appbundle` command assists in setting up the project with a validator that guides developers through the changes needed to build properly. This validator is necessary since the exact loading units produced by `gen_snapshot` is not known until `gen_snapshot` finishes compiling. Thus, some project setup can only be done after the `gen_snapshot` step in the build appbundle process.

Since mistakenly importing a deferred file as non-deferred can cause the file to be compiled into the base loading unit, the deferred components validator also has a mechanism for preventing accidental changes to the app's final generated loading units. This check will cause the build to fail if the generated loading units do not match the results of the previous run which is cached in the `deferred_components_loading_units.yaml` file. After throwing an error upon detecting changes, the build will automatically pass this check on next run if no additional changes are made. This means that this check is not error proof as you are still free to ignore the mismatched loading unit error if the change was intended and accounted for and continue to build.

Fully deferring Flutter in add-to-app

When using add-to-app, it's possible to convert the entire Flutter module into an Android dynamic feature module to install at runtime.

Since the structures of add-to-app scenarios are highly variable, Flutter doesn't provide direct tooling to automate/validate full Flutter deferring. Instead, this guide provides details for implementing the components required for full Flutter deferring. The architecture described here is one of the many ways this can work, and is up to you to determine how best to integrate this into your apps.

Integrating full Flutter deferring is experimental. It requires custom implementations, and setup that is not fully tested due to variability in different apps. Therefore, this feature is considered a very advanced feature, and Flutter may not be able to provide guarantees or technical support for specific use cases.

Full Flutter deferral requires implementing `SplitInstallManager` in the base app module, as well as adding the dependency on `com.google.android.play:core` in `build.gradle` as an implementation. The dynamic feature module containing Flutter must depend on the base module.

The base module can no longer include any references to Flutter code. Therefore, the `:flutter` dependency in `build.gradle` should be removed.

The process described below uses the [fullscreen add-to-app example](#).

SplitInstallManager base module "bootstrapper"

The base module must use `SplitInstallManager` to install the Flutter dynamic feature. For example, here is a bare-bones implementation called `SplitUtility` that downloads a dynamic feature module named `flutter` when `installFlutterModule` is called:

```
import android.annotation.SuppressLint;
import android.content.Context;
import androidx.annotation.NonNull;
import com.google.android.play.core.splitinstall.SplitInstallException;
import com.google.android.play.core.splitinstall.SplitInstallManager;
import com.google.android.play.core.splitinstall.SplitInstallManagerFactory;
import com.google.android.play.core.splitinstall.SplitInstallRequest;
import com.google.android.play.core.splitinstall.SplitInstallSessionState;
import com.google.android.play.core.splitinstall.SplitInstallStateUpdatedListener;
import com.google.android.play.core.splitinstall.model.SplitInstallErrorCode;
import com.google.android.play.core.splitinstall.model.SplitInstallSessionStatus;

class SplitUtility {
  private @NonNull SplitInstallManager splitInstallManager;
  private @NonNull FeatureInstallStateUpdatedListener listener;

  private class FeatureInstallStateUpdatedListener implements SplitInstallStateUpdatedListener {
    @SuppressLint("DefaultLocale")
    public void onStateUpdate(SplitInstallSessionState state) {
      int sessionId = state.sessionId();
      switch (state.status()) {
        case SplitInstallSessionStatus.FAILED:
          break;
        case SplitInstallSessionStatus.INSTALLED:
          break;
        case SplitInstallSessionStatus.CANCELED:
          break;
        default:
      }
    }
  }

  SplitUtility(Context context) {
    splitInstallManager = SplitInstallManagerFactory.create(context);
    listener = new FeatureInstallStateUpdatedListener();
    splitInstallManager.registerListener(listener);
  }
  void installFlutterModule() {
    SplitInstallRequest request =
    SplitInstallRequest.newBuilder().addModule("flutter").build();

    splitInstallManager
      // Submits the request to install the module through the
      // asynchronous startInstall() task. Your app needs to be
      // in the foreground to submit the request.
      .startInstall(request)
      // Called when the install request is sent successfully. This is different than a successful
      // install which is handled in FeatureInstallStateUpdatedListener.
      .addOnSuccessListener(
          sessionId -> {
            // store sessionId somewhere if you want to reference it again.
          })
      .addOnFailureListener(
          exception -> {
            switch (((SplitInstallException) exception).getErrorCode()) {
              case SplitInstallErrorCode.NETWORK_ERROR:
                break;
              case SplitInstallErrorCode.MODULE_UNAVAILABLE:
                break;
              default:
                break;
            }
          });
  }

  public void destroy() {
    splitInstallManager.unregisterListener(listener);
  }
}
```

The base module should install the Flutter module when appropriate. `PlayStoreDeferredComponentsManager` actually provides much of the same functionality, but it lives inside the Flutter Android embedding, and thus cannot be referenced from the base module.

Project configuration

`build.gradle` of the base module as well as the flutter module should be modified to convert it into a dynamic feature module. The default Flutter android module used is located in the `.android/Flutter` directory.

It's recommended that you change the path of the Android module as the `.android` directory may be cleared or regenerated by cleaning tasks. You can set a different module or a clone of the default one as the module root directory with `project(":flutter").projectDir = new File("<relative>.<path>.<to>.<module>")` in your main android project's `settings.gradle`.

The following configuration changes are the base changes needed to convert a fullscreen Flutter add-to-app implementation into a dynamic feature module.

Base module `build.gradle`:

- Add `dynamicFeatures = [':flutter']` to the `android` section of the base module `build.gradle`.
- Remove `implementation project(':flutter')` from the dependencies section (the dynamic feature module must depend on the base module).
- Add `implementation "com.google.android.play:core:1.8.0"` to the dependencies section.

Flutter module `build.gradle`:

- Replace `apply plugin: 'com.android.library'` with `apply plugin: 'com.android.dynamic-feature'` in the Flutter module `build.gradle`
- Add a dependencies section to the Flutter module `build.gradle`:

```
dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation project(":app")
    api 'androidx.test:core:1.2.0'
}
```

Flutter module `AndroidManifest.xml` :

- Add `xmlns:dist="http://schemas.android.com/apk/distribution"` to the `manifest` section
- Add the dynamic feature module section:

```
<dist:module
  dist:instant="false"
  dist:title="@string/title_fluttermodule">
  <dist:delivery>
     <dist:on-demand />
  </dist:delivery>
  <dist:fusing dist:include="true" />
</dist:module>
```

Any references to the Flutter Java API will need to be done within your newly dynamic feature module. You are no longer able to directly launch a `FlutterActivity` from your main activity, and must now wrap it in a new class inside your dynamic feature module.

It is recommended to create a new Android Activity in your dynamic Flutter module that implements all of the behavior from your base application and base Activity. After installing the Flutter Android split, the new dynamic Flutter Activity can then be launched.

Depending on your specific apps, additional configuration may be needed to build and run your apps.

Regular deferred components integration

We have not yet tested integration with a pure Flutter app using deferred components.

Flutter's tooling doesn't yet support building add-to-app and deferred components together. However, it is technically possible to run `gen_snapshot` in split AOT mode, and then package the different `.so` files into additional dynamic feature modules. See the Custom Implementation below for additional details.

Custom Implementations

It's possible to write a custom implementation that bypasses the Android Play store. This is only recommended for advanced developers, and is primarily aimed at apps with very unique needs such as extremely large asset components, specific download behavior, or distribution in a region that does not have access to the Play Store (e.g. China).

Overview

The Flutter Embedder allows custom implementations that handle customer-unique download and unpacking of deferred components while still allowing access to the core Dart callbacks that register a loading unit with the Dart runtime. This process is far more involved than the default play store version.

To implement a custom deferred components system, the following major pieces are required:

- Android embedder implementation of `DeferredComponentManager` that handles communication between the app and the server as well as extracting the `.so` file and assets from the downloaded component.
- Tooling to package the components in a way that is compatible with your `DeferredComponentManager` and to interpret `gen_snapshot` output of loading units.
- A server to host the downloadable components. Without the Play Store acting as a distributor for Android dynamic feature modules, this must be custom.

The following sections provide a high level guide of what needs to be done in a custom implementation.

DeferredComponentManager - Android Embedder

The Embedder is responsible for downloading and installing the packaged component files. This can be done by implementing the abstract class `DeferredComponentManager` in the Android Embedder.

The entry point into this class is `installDeferredComponent` which provides both a loading unit id and the component name to help determine what to install.

`loadLibrary()` calls will pass only a loading unit id while `DeferredComponent.installDeferredComponent()` calls from the framework services package will pass only a component name to load an assets-only component.

In order to resolve a specific component from the loading unit id, it is typically necessary to store a mapping of loading unit ids to the component name. In the default implementation, we accomplish this by storing a string meta-data in the base app's `AndroidManifest.xml`, but this can be accomplished in any way desired.

You may find detailed explanations of each method in `DeferredComponentManager` in the engine source file at `shell/platform/android/io/flutter/embedding/engine/deferredcomponents/DeferredComponentManager.java`.

The default Play Store implementation is found at `shell/platform/android/io/flutter/embedding/engine/deferredcomponents/PlayStoreDeferredComponentManager.java` and can be used as a rough guide on what needs to be implemented.

To load Dart libraries, call `FlutterJNI.loadDartDeferredLibrary` with the loading unit id, and a list of paths that potentially contain the `.so` file in your `loadDartLibrary` implementation. The engine will call `dlopen` on each of the paths provided until one is successfully opened.

To load new assets, create an Android `AssetManager` that has access to the newly downloaded assets. Pass this `AssetManager` to `FlutterJNI.updateJavaAssetManager`.

The `FlutterJNI` instance is passed in via `setJNI`.

Tooling

Flutter's tooling comes with the capability to instruct `gen_snapshot` to build split AOT, and pack the `.so` files and assets into an Android dynamic feature module.

Custom implementations are typically unable to make use of this tooling. Therefore, you may have to write custom tooling to package the `.so` files and assets, so they work alongside the custom `DeferredComponentManager` implementation.

To make `gen_snapshot` generate loading units, and the `.so` shared libraries, pass the `--loading_unit_manifest=<manifestPath>` option to `gen_snapshot`.

This will create a .json file at your `manifestPath` that contains the loading units and corresponding `.so` libs generated. The `.so` file and assets can then be packed in whatever format you wish to distribute on your file server. It is also your responsibility to unpack this format in your `DeferredComponentManager` implementation.

File server

Since custom implementations typically do not use the Play store, a custom system for hosting and serving files to end users should be implemented. This part is highly variable in how it can be accomplished, and the only requirement is that it functions in tandem with the `DeferredComponentManager` implementation, so it delivers the files needed to load Dart shared libraries, and assets.