Using packages

Contents

Flutter supports using shared packages contributed by other developers to the Flutter and Dart ecosystems. This allows quickly building an app without having to develop everything from scratch.

What is the difference between a package and a plugin? A plugin is a *type* of package—the full designation is *plugin package*, which is generally shortened to *plugin*.

Packages
At a minimum, a Dart package is a directory containing a pubspec file. Additionally, a package can contain dependencies (listed in the pubspec), Dart libraries, apps, resources, tests, images, and examples. The pub.dev site lists many packages—developed by Google engineers and generous members of the Flutter and Dart community— that you can use in your app.

Plugins
A plugin package is a special kind of package that makes platform functionality available to the app. Plugin packages can be written for Android (using Kotlin or Java), iOS (using Swift or Objective-C), web, macOS, Windows, Linux, or any combination thereof. For example, a plugin might provide Flutter apps with the ability to use a device's camera.

Existing packages enable many use cases—for example, making network requests (`http`), custom navigation/route handling (`fluro`), integration with device APIs (`url_launcher` and `battery`), and using third-party platform SDKs like Firebase (FlutterFire).

To write a new package, see developing packages. To add assets, images or fonts, whether stored in files or packages, see Adding assets and images.

Using packages

The following section describes how to use existing published packages.

Searching for packages

Packages are published to pub.dev.

The Flutter landing page on pub.dev displays top packages that are compatible with Flutter (those that declare dependencies generally compatible with Flutter), and supports searching among all published packages.

The Flutter Favorites page on pub.dev lists the plugins and packages that have been identified as packages you should first consider using when writing your app. For more information on what it means to be a Flutter Favorite, see the Flutter Favorites program.

You can also browse the packages on pub.dev by filtering on Android plugins, iOS plugins, web plugins, or any combination thereof.

Adding a package dependency to an app

To add the package, `css_colors`, to an app:

1. Depend on it
   - Open the `pubspec.yaml` file located inside the app folder, and add `css_colors:` under `dependencies`.
2. Install it
   - From the terminal: Run `flutter pub get`.
     OR
   - From Android Studio/IntelliJ: Click **Packages get** in the action ribbon at the top of `pubspec.yaml`.
   - From VS Code: Click **Get Packages** located in right side of the action ribbon at the top of `pubspec.yaml`.
3. Import it
   - Add a corresponding `import` statement in the Dart code.
4. Stop and restart the app, if necessary

   - If the package brings platform-specific code (Kotlin/Java for Android, Swift/Objective-C for iOS), that code must be built into your app. Hot reload and hot restart only update the Dart code, so a full restart of the app might be required to avoid errors like `MissingPluginException` when using the package.

The Installing tab, available on any package page on pub.dev, is a handy reference for these steps.

For a complete example, see the css_colors example below.

Conflict resolution

Suppose you want to use `some_package` and `another_package` in an app, and both of these depend on `url_launcher`, but in different versions. That causes a potential conflict. The best way to avoid this is for package authors to use version ranges rather than specific versions when specifying dependencies.

content_copy
```
dependencies:
  url_launcher: ^5.4.0    # Good, any version >= 5.4.0 but < 6.0.0
  image_picker: '5.4.3'   # Not so good, only version 5.4.3 works.
```

If `some_package` declares the dependencies above and `another_package` declares a compatible `url_launcher` dependency like `'5.4.6'` or `^5.5.0`, pub resolves the issue automatically. Platform-specific dependencies on Gradle modules and/or CocoaPods are solved in a similar way.

Even if `some_package` and `another_package` declare incompatible versions for `url_launcher`, they might actually use `url_launcher`in compatible ways. In this situation, the conflict can be resolved by adding a dependency override declaration to the app's`pubspec.yaml` file, forcing the use of a particular version.

For example, to force the use of `url_launcher` version `5.4.0`, make the following changes to the app's `pubspec.yaml` file:

content_copy
```
dependencies:
  some_package:
  another_package:
dependency_overrides:
  url_launcher: '5.4.0'
```

If the conflicting dependency is not itself a package, but an Android-specific library like `guava`, the dependency override declaration must be added to Gradle build logic instead.

To force the use of `guava` version `28.0`, make the following changes to the app's `android/build.gradle` file:

```
configurations.all {
    resolutionStrategy {
        force 'com.google.guava:guava:28.0-android'
    }
}
```
content_copy

CocoaPods does not currently offer dependency override functionality.

Developing new packages

If no package exists for your specific use case, you can write a custom package.

Managing package dependencies and versions

To minimize the risk of version collisions, specify a version range in the `pubspec.yaml` file.

Package versions

All packages have a version number, specified in the package's `pubspec.yaml` file. The current version of a package is displayed next to its name (for example, see the `url_launcher` package), as well as a list of all prior versions (see `url_launcher` versions).

When a package is added to `pubspec.yaml`, the shorthand form `plugin1:` means that any version of the plugin1 package can be used. To ensure that the app doesn't break when a package is updated, specify a version range using one of the following formats:

- Range constraints: Specify a minimum and maximum version. For example:

  ```
  dependencies:
    url_launcher: '>=5.4.0 <6.0.0'
  ```
  content_copy

- Range constraints with *caret syntax* are similar to regular range constraints:

  ```
  dependencies:
    collection: '^5.4.0'
  ```
  content_copy

For additional details, see the package versioning guide.

Updating package dependencies

When running `flutter pub get` (**Packages get** in IntelliJ or Android Studio) for the first time after adding a package, Flutter saves the concrete package version found in the `pubspec.lock` lockfile. This ensures that you get the same version again if you, or another developer on your team, run `flutter pub get`.

To upgrade to a new version of the package, for example to use new features in that package, run `flutter pub upgrade` (**Upgrade dependencies** in IntelliJ or Android Studio) to retrieve the highest available version of the package that is allowed by the version constraint specified in `pubspec.yaml`. Note that this is a different command from `flutter upgrade` or `flutter update-packages`, which both update Flutter itself.

Dependencies on unpublished packages

Packages can be used even when not published on pub.dev. For private plugins, or for packages not ready for publishing, additional dependency options are available:

Path dependency

A Flutter app can depend on a plugin via a file system `path:` dependency. The path can be either relative or absolute. Relative paths are evaluated relative to the directory containing `pubspec.yaml`. For example, to depend on a plugin `plugin1` located in a directory next to the app, use the following syntax:

```
dependencies:
  plugin1:
    path: ../plugin1/
```
content_copy

Git dependency

You can also depend on a package stored in a Git repository. If the package is located at the root of the repo, use the following syntax:

```
dependencies:
  plugin1:
    git:
      url: git://github.com/flutter/plugin1.git
```
content_copy

Git dependency on a package in a folder

Pub assumes the package is located in the root of the Git repository. If that is not the case, specify the location with the `path`argument. For example:

```
dependencies:
  package1:
    git:
      url: git://github.com/flutter/packages.git
      path: packages/package1
```
content_copy

Finally, use the `ref` argument to pin the dependency to a specific git commit, branch, or tag. For more details, see Package dependencies.

Examples

The following examples walk through the necessary steps for using packages.

Example: Using the css_colors package

The `css_colors` package defines color constants for CSS colors, so use the constants wherever the Flutter framework expects the `Color` type.

To use this package:

1. Create a new project called `cssdemo`.

2. Open `pubspec.yaml`, and add the `css-colors` dependency:

   content_copy
   ```yaml
   dependencies:
     flutter:
       sdk: flutter
     css_colors: ^1.0.0
   ```

3. Run `flutter pub get` in the terminal, or click **Packages get** in IntelliJ or Android Studio.

4. Open `lib/main.dart` and replace its full contents with:

   content_copy
   ```dart
   import 'package:css_colors/css_colors.dart';
   import 'package:flutter/material.dart';

   void main() {
     runApp(MyApp());
   }

   class MyApp extends StatelessWidget {
     @override
     Widget build(BuildContext context) {
       return MaterialApp(
         home: DemoPage(),
       );
     }
   }

   class DemoPage extends StatelessWidget {
     @override
     Widget build(BuildContext context) {
       return Scaffold(body: Container(color: CSSColors.orange));
     }
   }
   ```

5. Run the app. The app’s background should now be orange.

Example: Using the url_launcher package to launch the browser

The url_launcher plugin package enables opening the default browser on the mobile platform to display a given URL, and is supported on Android, iOS, web, and macos. This package is a special Dart package called a *plugin package* (or *plugin*), which includes platform-specific code.

To use this plugin:

1. Create a new project called `launchdemo`.
2. Open `pubspec.yaml`, and add the `url_launcher` dependency:

   ```
   dependencies:
     flutter:
       sdk: flutter
     url_launcher: ^5.4.0
   ```
   content_copy

3. Run `flutter pub get` in the terminal, or click **Packages get** in IntelliJ or Android Studio.
4. Open `lib/main.dart` and replace its full contents with the following:

content_copy

```dart
import 'package:flutter/material.dart';
import 'package:url_launcher/url_launcher.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: DemoPage(),
    );
  }
}

class DemoPage extends StatelessWidget {
  launchURL() {
    launch('https://flutter.dev');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: launchURL,
          child: Text('Show Flutter homepage'),
        ),
      ),
    );
  }
}
```

5. Run the app (or stop and restart it, if it was already running before adding the plugin). Click **Show Flutter homepage**. You should see the default browser open on the device, displaying the homepage for flutter.dev.