

# Flutter - зависимости проекта

Библиотеки во [Flutter-проекте](#) играют важную роль в разработке приложений. Они представляют собой совокупность предопределённых функций, классов и компонентов, которые могут быть использованы разработчиками для упрощения и ускорения процесса создания приложений.

## Типы библиотек

Библиотеки бывают двух типов:

- пакеты (англ. packages) — содержат только код, написанный на Dart и Flutter;
- плагины (англ. plugins) — помимо Dart и Flutter имеют платформо-специфичный код, также называемый нативным. Хоть Flutter и позволяет писать сразу под несколько платформ, в некоторых случаях может потребоваться использование специфической функциональности, которая доступна только на одной из платформ. Например, это может быть интеграция с уведомлениями операционной системы, использование специфических интерфейсов пользователя, доступ к аппаратным функциям устройства (камере, датчикам, Bluetooth и проч.) или взаимодействие с другими приложениями, доступными только на конкретной платформе.

## pub.dev

- Когда мы говорим о библиотеках во [Flutter](#), сразу вспоминается популярный ресурс — [pub.dev](#). Данный сайт — официальный репозиторий пакетов для [Dart](#) и [Flutter](#), он предоставляет разработчикам доступ к широкому спектру пакетов, которые могут использоваться для [Flutter-проектов](#). На нём размещены пакеты, созданные сообществом [Dart](#) и [Flutter](#), а также пакеты, разработанные и поддерживаемые самой командой Google.
- Страница каждого пакета содержит его название, актуальную версию, дату публикации, автора и другую информацию, которая может быть полезна при использовании пакета.



# shared\_preferences 2.2.2

Published 42 days ago • ⚙ flutter.dev (Dart 3 compatible)

SDK FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS

8.2K

Readme Changelog Example Installing Versions Scores

8259 LIKES 140 PUB POINTS 100% POPULARITY



## provider 6.1.1

Published 12 days ago • ⚙ dash-overflow.net (Dart 3 compatible)

SDK FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS

9.1K

Readme Changelog Example Installing Versions Scores

9153 LIKES 140 PUB POINTS 100% POPULARITY

English | French | Português | 简体中文 | Español | 한국어 | বাংলা | 日本語 | Turkish

Build passing codecov 99% chat 248 online



## flutter\_bloc 8.1.3

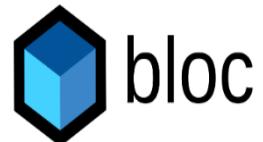
Published 6 months ago • ⚙ bloclibrary.dev (Dart 3 compatible)

SDK FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS

6.1K

Readme Changelog Example Installing Versions Scores

6152 LIKES 140 PUB POINTS 100% POPULARITY



pub v8.1.3 build passing codecov 100% stars 11k flutter website awesome flutter flutter samples  
license MIT chat 290 online bloc library



Publisher

bloclibrary.dev

Metadata

Publisher

dash-overflow.net

Metadata

A wrapper around InheritedWidget to make them easier to use and more reusable.

Repository (GitHub)

View/report issues

## Зависимости

- Для добавления зависимости в проект используется поле [dependencies](#) в файле [pubspec.yaml](#), под ним указывается название библиотеки, её источник и версия, если она обязательна для выбранного типа источника. Зависимости бывают четырёх видов с точки зрения источников.

## Зависимость из pub.dev

- Зависимость, которая была опубликована на сайте [pub.dev](#), официальном публичном репозитории библиотек для [Dart](#) и [Flutter](#).

```
dependencies:  
  dependency_name: <version>
```

## Git-зависимость

- Зависимость из Git-репозитория.

```
dependencies:  
  <dependency_name>:  
    git: https://github.com/<repo_url>  
    ## Зависимость лежит по указанному пути в корне проекта
```

**Помимо указания ссылки мы можем добавлять параметры `ref` и `path`:**

- ref — позволяет указывать номер коммита, название ветки или тега, чтобы тянуть зависимость не из ветки master или main, а исходя из идентификатора;
- path — позволяет указывать пути до библиотеки, если пакет лежит не в корне проекта.

```
dependencies:  
  <dependency_name>:  
    git:  
      url: https://github.com/<repo_url>  
      ref: some-branch  
      path: some-path
```

## Локальная зависимость

- Зависимости, которые лежат в локальной файловой системе, могут устанавливаться локально. Для этого нужно прописать полный или относительный путь до этой зависимости.

```
dependencies:
```

```
<dependency_name>:  
  path: <your_local_path>
```

```
dependencies:
```

```
<dependency_name>:  
  hosted: https://some-package-server.com  
  version: <version>
```

## Hosted-зависимость

- Зависимость, которая находится на [pub.dev](#), публична и доступна для всех разработчиков. Не всегда разработчики готовы делиться своими наработками, так что существует возможность загружать пакет с удалённого хранилища пакетов, который имеет такой же [API, что и pub.dev](#). Для установки таких зависимостей требуется указывать [URL в параметре hosted](#).

Такое приватное хранилище пакетов можно создать с помощью специального пакета [upriv](#).

## Версионирование зависимостей

Для определения версии пакета в [Dart](#) используется [семантическое версионирование](#) (англ. [semantic versioning](#)), которое состоит из трёх чисел, разделённых точками: [мажор.минор.патч](#) (англ. [major.minor.patch](#)). Каждое из чисел обозначает различные изменения в пакете:

- [мажор \(основная версия\)](#) — изменения в API, которые обратно несовместимы (англ. [breaking changes](#)). Такие изменения могут повлиять на существующий код, который использует предыдущую версию библиотеки, и могут потребовать изменений или модификаций в этом коде, чтобы он продолжал работать с новой версией;
- [минор \(англ. minor\)](#) — добавление новой функциональности в существующее API, которое не нарушает обратную совместимость;
- [патч \(англ. patch\)](#) — исправление ошибок в пакете, которые не влияют на API.

В [Dart](#) можно использовать операторы [версионирования](#) для указания ограничений версий зависимостей. Операторы [версионирования](#), которые также называют [констрайнтами \(англ. constraints\)](#), включают в себя следующие:

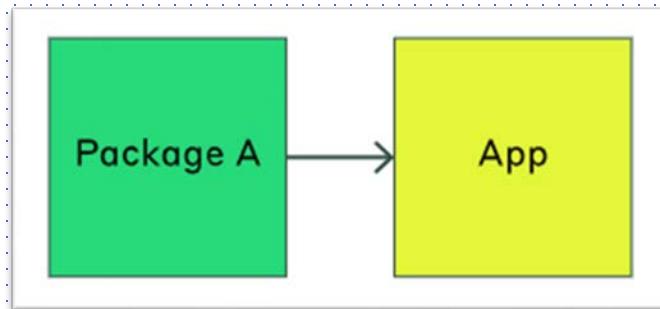
- [any](#) — используется для указания того, что проект совместим со всеми версиями зависимости. Это означает, что проект может использовать любую версию зависимости, которая доступна на момент установки;
- [a.b.c](#) — установится версия a.b.c , где a — мажор, b — минор, c — патч;
- [> a.b.c](#) — установится максимально возможная версия из тех, что выше a.b.c;
- [>= a.b.c](#) — установится максимально возможная версия из тех, что выше, или равная версии a.b.c;
- [< a.b.c](#) — установится максимально возможная версия из тех, что ниже a.b.c;
- [<= a.b.c](#) — установится максимально возможная версия из тех, что ниже, или равная версии x.y.z;
- [^ a.b.c](#) — расшифровывается как  $\geq a.b.c <(x+1).0.0 \wedge 1.2.3$  эквивалентно  $\geq 1.2.3 < 2.0.0$ . Однако для версий с мажором, равным нулю, работает по-другому:  $\wedge 0.1.2$  эквивалентно  $\geq 0.1.2 < 0.2.0$ . Если мажор равен нулю, то библиотека ещё в бета-версии и любые минорные изменения могут быть обратно несовместимы.

О семантическом [версионировании](#) можно прочесть в официальной [документации](#). В ней же, в последнем абзаце, рассказано о библиотеках с [нулевой мажорной версией](#).

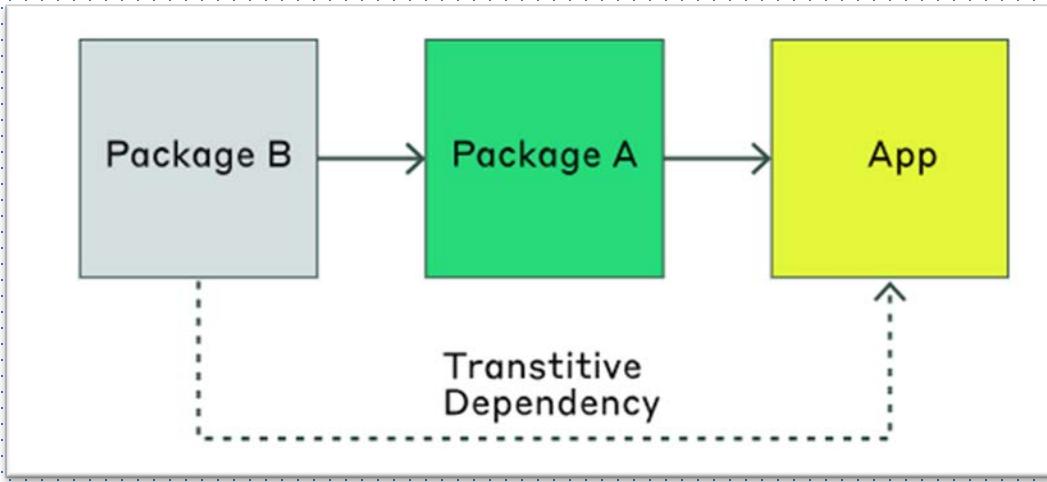
Также можно указывать диапазон версий с помощью комбинирования <, <=, >, >=. Например, конструкция  $\geq 1.12.8 \leq 3.0.0$  означает, что мы хотим установить версию от 1.12.8 включительно до 3.0.0 включительно.

Но что значит максимально возможная версия? Чтобы это понять, нужно узнать ещё об одном варианте разделения зависимостей: [прямые, транзитивные и совместно используемые](#).

Вот так выглядит прямая зависимость, мы добавляем пакет A напрямую в наше приложение.

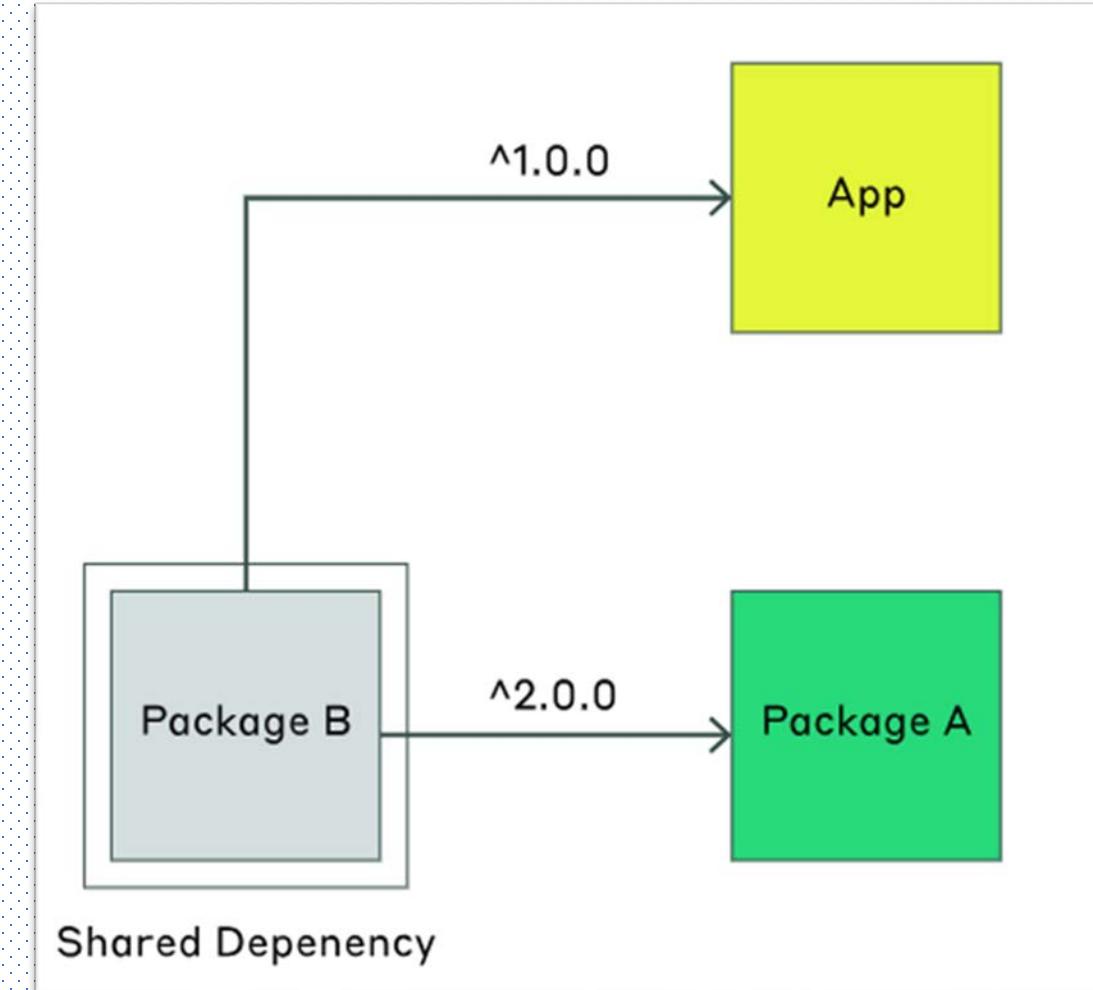


Транзитивная (англ. *transitive*) же зависимость подтягивается из пакета, который мы устанавливаем как зависимость. Это зависимость нашей зависимости.



### А общая (англ. *shared*) зависимость

— зависимость, которую подтягиваем и мы, и используемый нами пакет.



Максимально возможная версия — та, которая удовлетворяет диапазону, образующемуся при подтягивании прямых, общих и транзитивных зависимостей. Для иллюстрации выше можно заметить, что мы пытаемся установить пакет в версии [^1.0.0](#) и одновременно версии [^2.0.0](#) — так как пересечения нет, возможной для установки версии тоже не существует. Такая ситуация называется конфликтом, и при вызове команды [flutter pub get](#) в консоли появится следующее сообщение:

```
Because Package A depends on Package B ^2.0.0 and no version
So, because app depends on both Package B ^1.0.0 and Package
```

## Существует три способа решения данной проблемы:

- Поменять версию пакета на `^2.0.0` в нашем приложении.
- Установить версию пакета, которая использует версию пакета `^1.0.0`.
- Переопределить зависимости с помощью `dependency overrides`. Как следует из названия, мы перезаписываем версию библиотеки. Можно указать пакет, версию и источник в свойстве `dependency overrides`, и `pub` будет использовать эти значения вместо тех, которые указаны в `dependencies` или подтягиваются транзитивно.

В нашем случае это работает так, что мы используем пакет, который устанавливает транзитивно пакет версии `2.0.0`, но мы хотим использовать версию `1.0.0`, и для этого можно указать в файле `pubspec.yaml` следующее:

```
dependency_overrides:  
  package_B: ^1.0.0
```

С переопределением версий нужно быть аккуратными. Есть риск, что после переопределения проект перестанет компилироваться, так как библиотеке, которая использует версию выше или ниже, будет не хватать функционала в переопределённой версии. Также не стоит устанавливать в проект библиотеки неактуальных версий, ведь фреймворк и библиотеки развиваются быстро, так что могут появиться несовместимые изменения. Чем дольше не обновлять библиотеки, тем сложнее это сделать потом. Также в новых версиях могут появиться новые возможности и правки недочётов.

Помимо зависимостей, указывающихся в `pubspec.yaml` как `dependencies`, существуют `dev dependencies`. Это зависимости, которые нужны только на этапе разработки, они не попадают в сборку приложения. К ним относится `linter`, различные зависимости для кодогенерации и аннотаций, зависимости для тестов и другие зависимости, которые не используются для сборки приложения.

```
dev_dependencies:  
    flutter_lints: ^2.0.2
```

## **pubspec.lock**

- [pubspec.lock](#) — файл, который автоматически генерируется при успешном запуске команды [pub get](#) в проекте [Flutter](#) или [Dart](#). Он содержит информацию о фактических версиях зависимостей, которые были установлены для проекта, включая транзитивные зависимости.
- Файл [pubspec.lock](#) содержит информацию о конкретных версиях каждого пакета, установленного в проекте. Это гарантирует, что при повторной установке пакетов будут использованы те же версии зависимостей, что и в предыдущий раз.

```
# Generated by pub
# See https://dart.dev/tools/pub/glossary#lockfile
packages: # Тут содержится список всех зависимостей
dependency_name:
  dependency: transitive # Тут содержится тип зависимости: transitive, shared, di
  description:
    ... # Тут содержится короткое описание зависимости, для зависимости с pub.de
  source: ... # Тут содержится источник зависимости, для зависимости с pub.dev ту
  version: ... # Тут – версия зависимости, которая установилась после прогона ком
sdks: # В этой части есть констрайнты для dart и flutter
  dart: ">=2.15.0 <3.0.0" # Устанавливаем Dart версии выше или равной 2.15.0,
  flutter: ">=2.8.0" # Устанавливаем Flutter версии выше или равной 2.8.0
```

## Стоит ли хранить этот файл в Git-репозитории

- Для [приложений — да](#). Он гарантирует, что при локальном запуске на разных платформах у разработчиков не разойдутся версии библиотек. Это можно объяснить на таком примере: в команду может прийти новый разработчик, запустить команду [pub get](#) — и у него не установятся новые мажорные версии библиотек, ведь [lock-файл](#) уже существует и не поменяется.
- Однако [для библиотек](#) этот файл в репозитории следует [игнорировать](#). Библиотеки должны содержать актуальные версии своих зависимостей, чтобы разработчик библиотеки не использовал старый сгенерированный [lock-файл](#), с которым у него всё работает, а пользователи не устанавливали новые версии транзитивных библиотек, с которыми проект несовместим.

## Заключение

- Не забывайте запускать команду `flutter pub get` перед запуском проекта, чтобы установить все необходимые зависимости. Если проект не собирается по непонятным причинам, может помочь команда `flutter clean` — она очистит `.dart tools` и удалит возможно повреждённый кеш. После неё снова предстоит переустановить зависимости с помощью команды `flutter pub get`.
- Можно использовать команду `flutter pub get --offline` для установки библиотек из кеша, если в данный момент нет соединения с интернетом.
- Когда мы запускаем команду `flutter pub get`, создаётся папка `.pub cache`, она содержит закешированные библиотеки, которые мы установили. Бывает такое, что кеш слетает. К примеру, когда в него добавилась кодогенерация или мы сами непреднамеренно изменили код одной из библиотек. Ни в коем случае не стоит модифицировать файлы из кеша. Вместо этого стоит запустить `flutter pub cache clean` для полной очистки кеша и `flutter pub cache repair` для переустановки каждой библиотеки.
- В некоторых случаях возникают ошибки, связанные с платформо-специфичными библиотеками. Это может произойти, если указанные библиотеки не были установлены. Для очистки нативных зависимостей с целью их переустановки следует запустить `pod deintegrate && pod install` для iOS и `gradlew clean` для Android.