Системы разработки кроссплатформенных мобильных приложений

или СРКМП

Введение

Что такое кроссплатформенность?

- способность программного обеспечения работать с двумя и более аппаратными платформами и (или) операционными системами

Плюсы кроссплатформенности:

- Единая логика приложения – логика приложения будет одинаково работать для всех платформ.
- Разработка кроссплатформенных приложений экономически эффективна
- Простое и быстрое развертывание
- Кроссплатформенные приложения покрывают более широкую аудиторию
- Кроссплатформенные приложения допускают одинаковый интерфейс и UX
- Поддержка и обновление продукта – добавление функционала или исправление ошибок сразу для всех платформ;

Минусы кроссплатформенности:

- **Кроссплатформенные приложения не являются такими гибкими, как нативные приложения**
- **Кроссплатформенные приложения не работают так же хорошо, как нативные приложения**
- **Возможное несоответствие UI в различных платформах**
- **Отправка кроссплатформенных приложений в соответствующие Магазины приложений может иметь сложности.**

Frameworks:

ionic

PhoneGap

Flutter

React Native

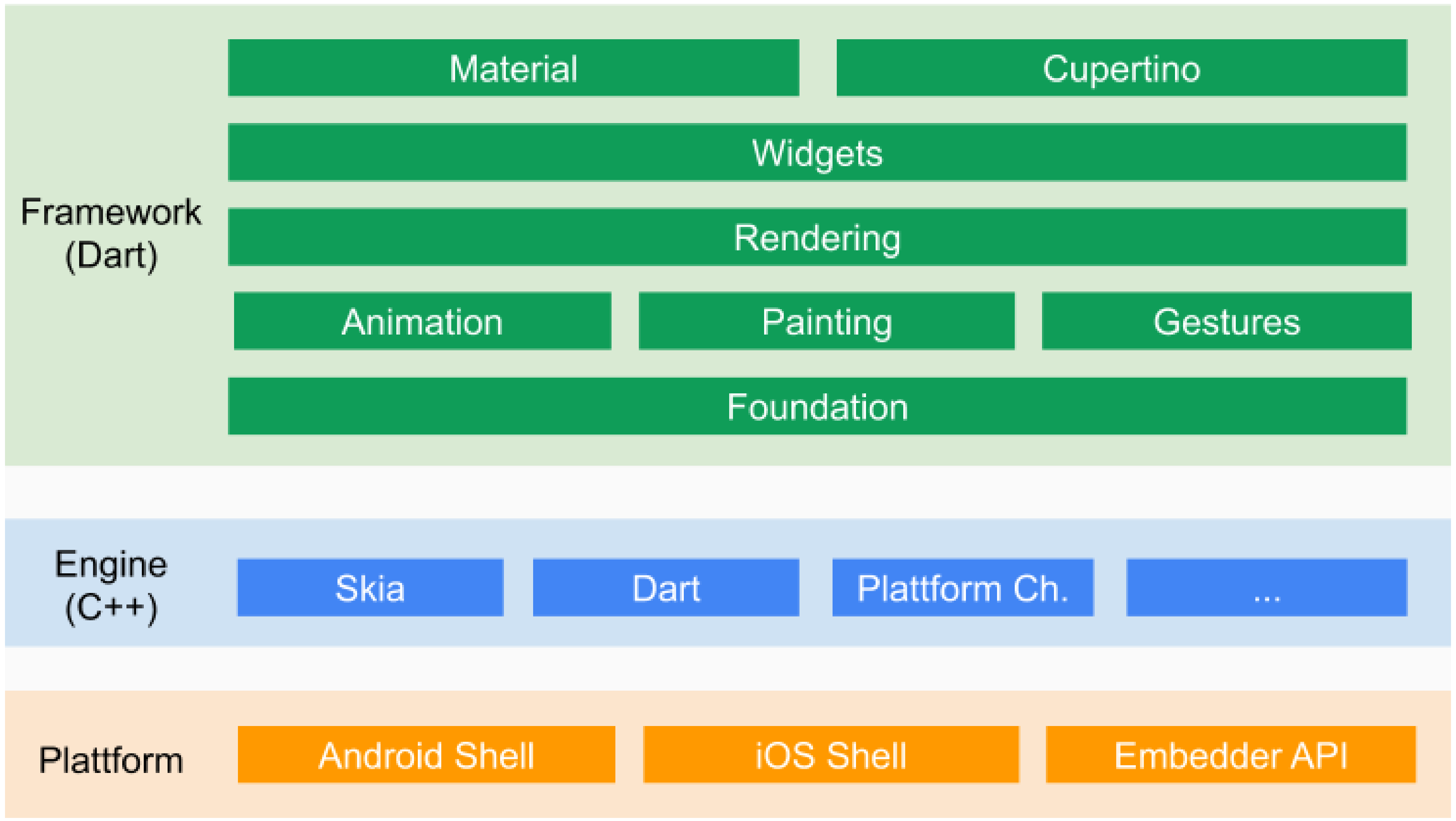
Xamarin

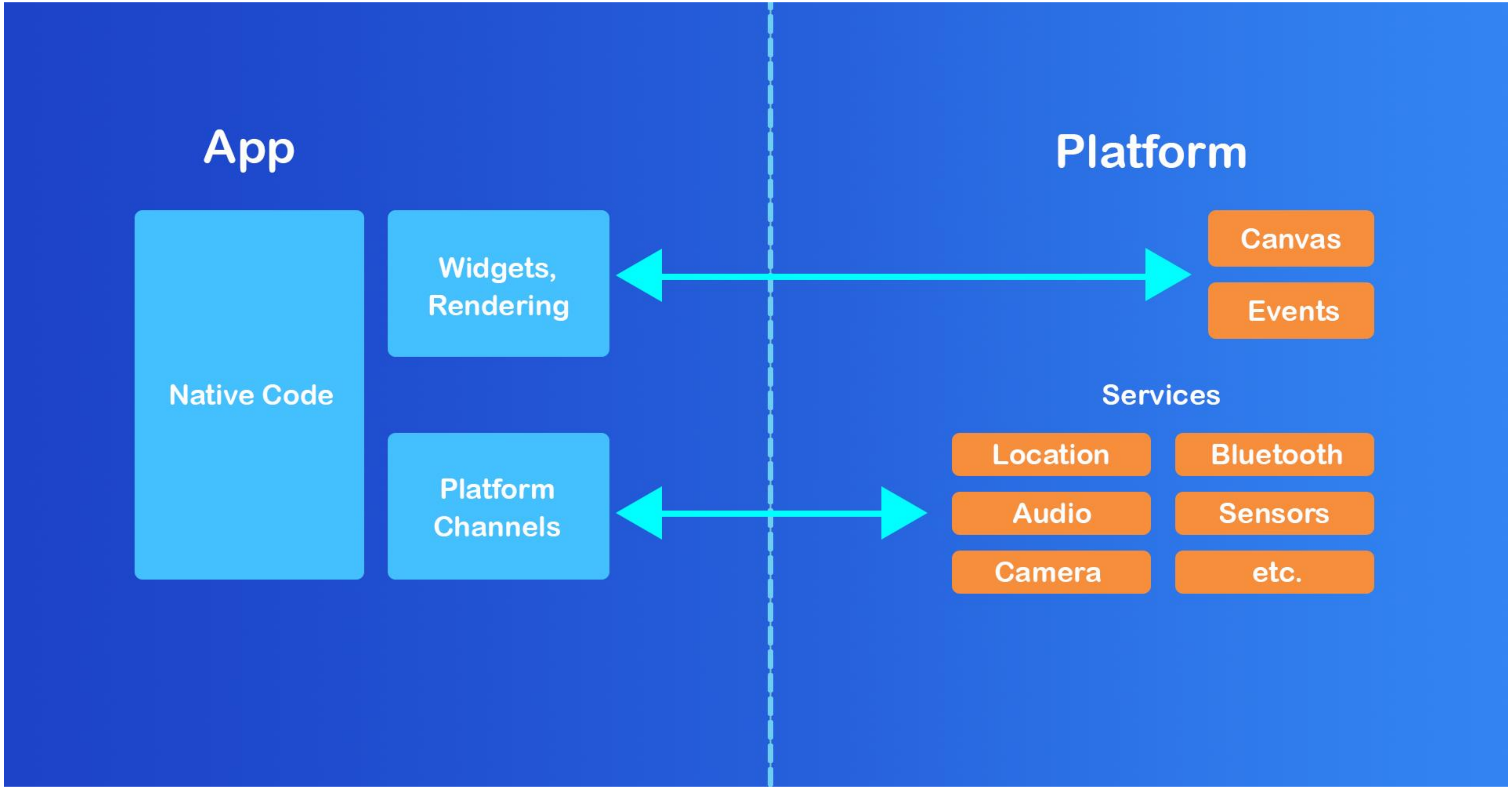
	Ionic	ReactNative	Flutter	Xamarin	PhoneGap
Owner	Ionic	Facebook	Google Inc.	Microsoft corp.	Adobe Systems Inc.
Language	HTML, JS, CSS	JS	Dart	C#	HTML, JS, CSS
Perfomance	moderate to near-native	near-native	faster then ReactNative	moderate to near-native	moderate to near-native
Used by	Market Watch, NHS, Untapped	Facebook, Instagram, Pinterest,Tesla,Walmart,Airbnb,Uber	Alibaba,AppTree, Google Ads, Tencent,Ebay, BMW	OLO, MRW, Storyo	Sworkit, TripCase

Made by Google

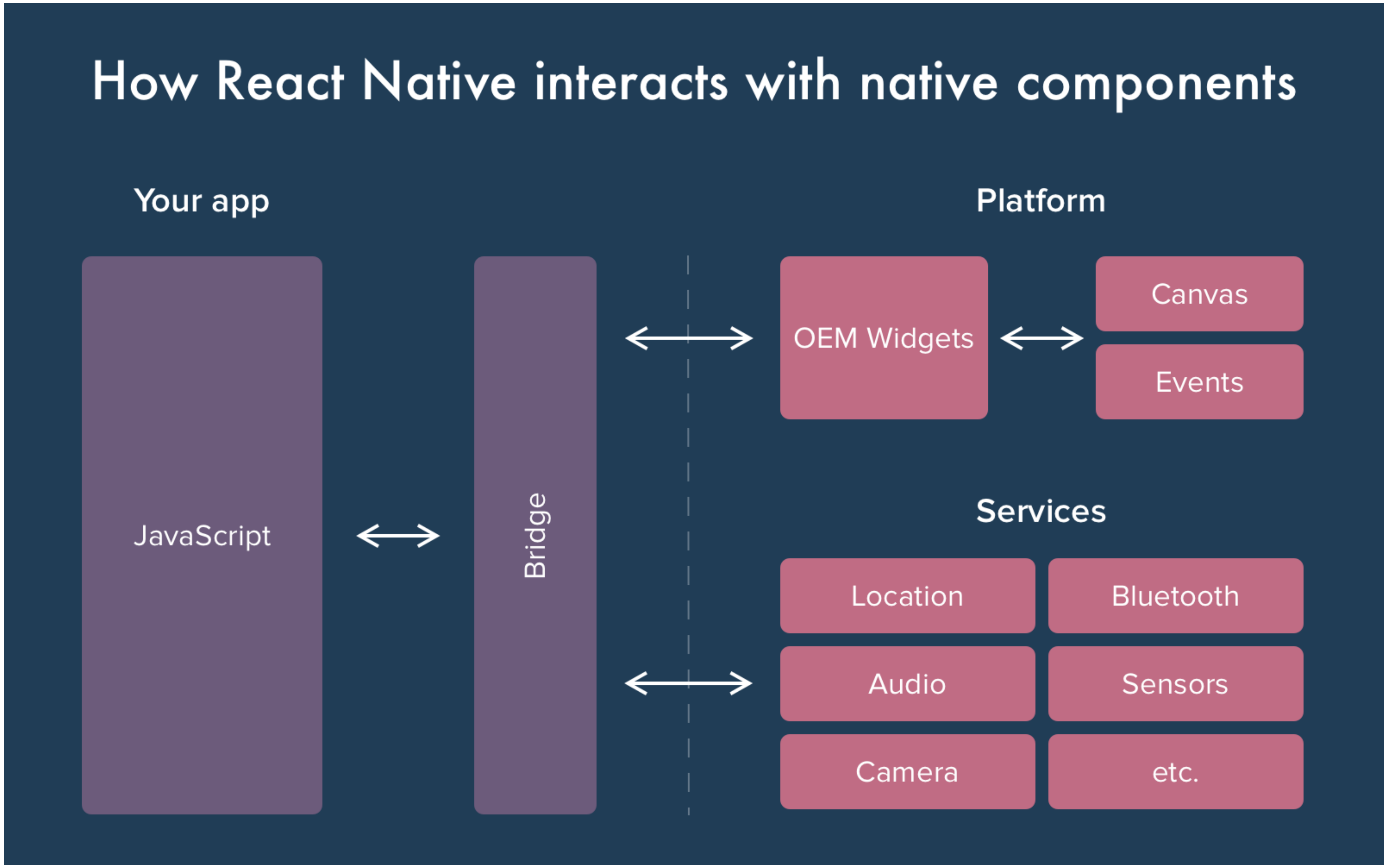
Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.

Архитектура Flutter




App
Native Code
Widgets, Rendering
Platform Channels
Platform
Canvas
Events
Services
Location
Bluetooth
Audio
Sensors
Camera
etc.

Для сравнения архитектура React Native



Dart

Dart is a client-optimized language for fast apps on any platform

Made by Google

Dart

- объектно-ориентированный язык программирования общего назначения.

- позиционируется в качестве замены/альтернативы JavaScript. («Javascript has fundamental flaws…» (c) Марк Миллер)

- C-подобный синтаксис

- Dart VM

- JIT и AOT, dart2js

- Hot Reload

Установка Dart

Windows:

Вариант 1: скачать установщик с сайта https://dart.dev/get-dart

Вариант 2: если установлен пакетный менеджер Chocolate: то выполнить команду - *choco install dart-sdk*

MacOS:

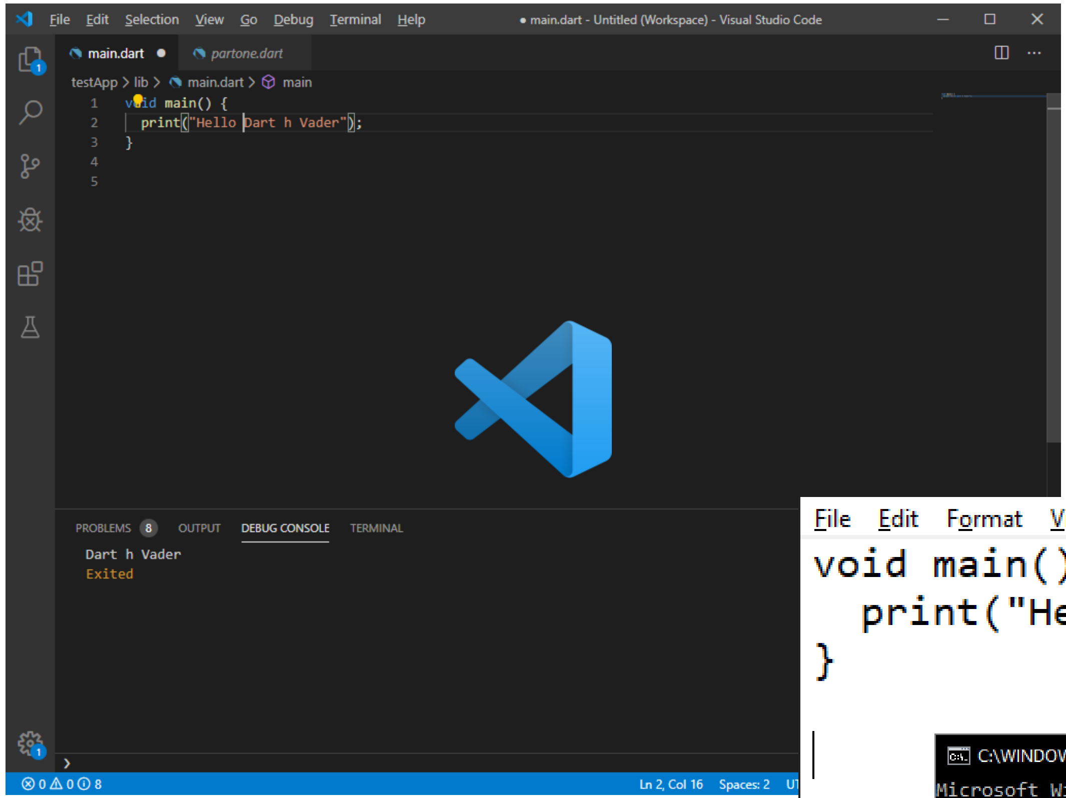
Выполнить команды в терминале:

- /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)”

- brew tap dart-lang/dart

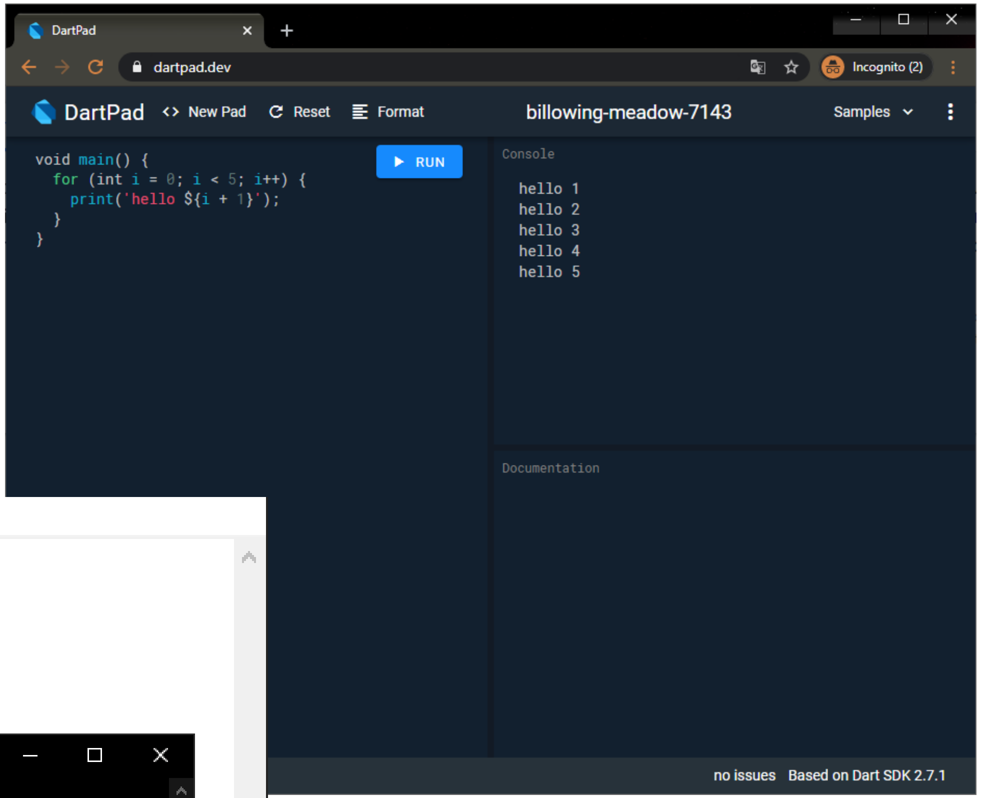
- brew install dart

IDE


File Edit Selection View Go Debug Terminal Help
● main.dart - Untitled (Workspace) - Visual Studio Code
main.dart
partone.dart
testApp > lib > main.dart > main
PROBLEMS 8
OUTPUT
DEBUG CONSOLE
TERMINAL
Ln 2, Col 16
Spaces: 2

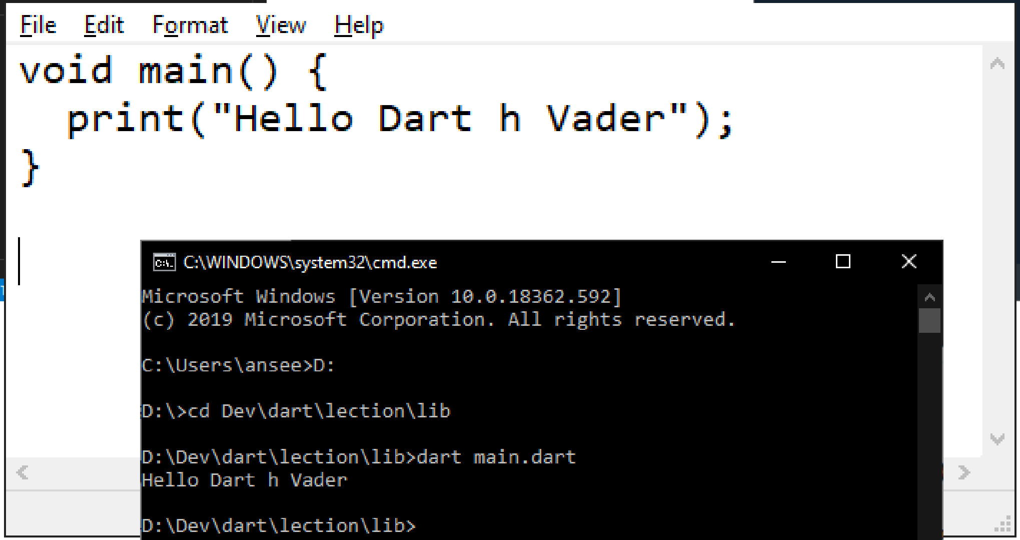
```dart
void main() {
  print("Hello Dart h Vader");
}
```

```
Dart h Vader
Exited
```

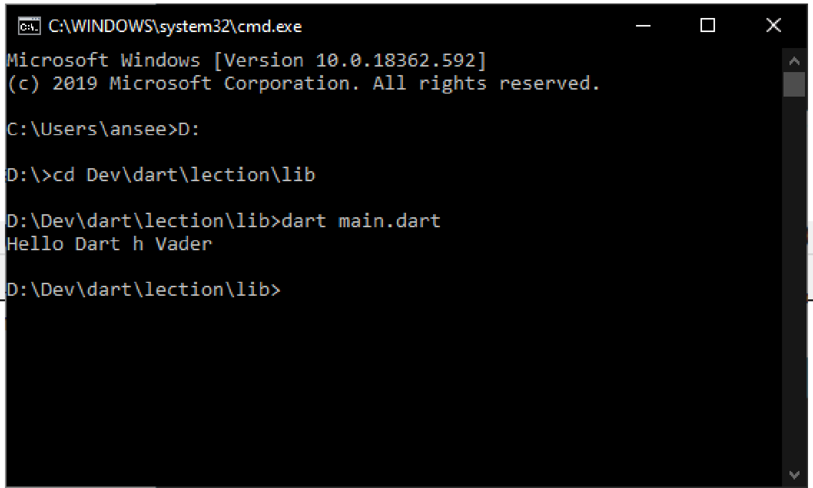


```dart
void main() {
  for (int i = 0; i < 5; i++) {
    print('hello ${i + 1}');
  }
}
```

```
hello 1
hello 2
hello 3
hello 4
hello 5
```



```dart
void main() {
  print("Hello Dart h Vader");
}
```



```
Microsoft Windows [Version 10.0.18362.592]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ansee>D:

D:\>cd Dev\dart\lection\lib

D:\Dev\dart\lection\lib>dart main.dart
Hello Dart h Vader

D:\Dev\dart\lection\lib>
```

Установка Flutter

Windows:

Скачать https://flutter.dev/docs/get-started/install/windows архив
Распаковать его в директорию не имеющую ограничений на запись
Прописать переменную среды Path, указав директорию куда распаковали архив
MacOS:
Скачать https://flutter.dev/docs/get-started/install/macos архив
И выполнить команды

```
- cd ~/development
- unzip ~/Downloads/flutter_macos_1.22.6-stable.zip
- export PATH="$PATH:`pwd`/flutter/bin"
```

IDE для Flutter

Android Studio:

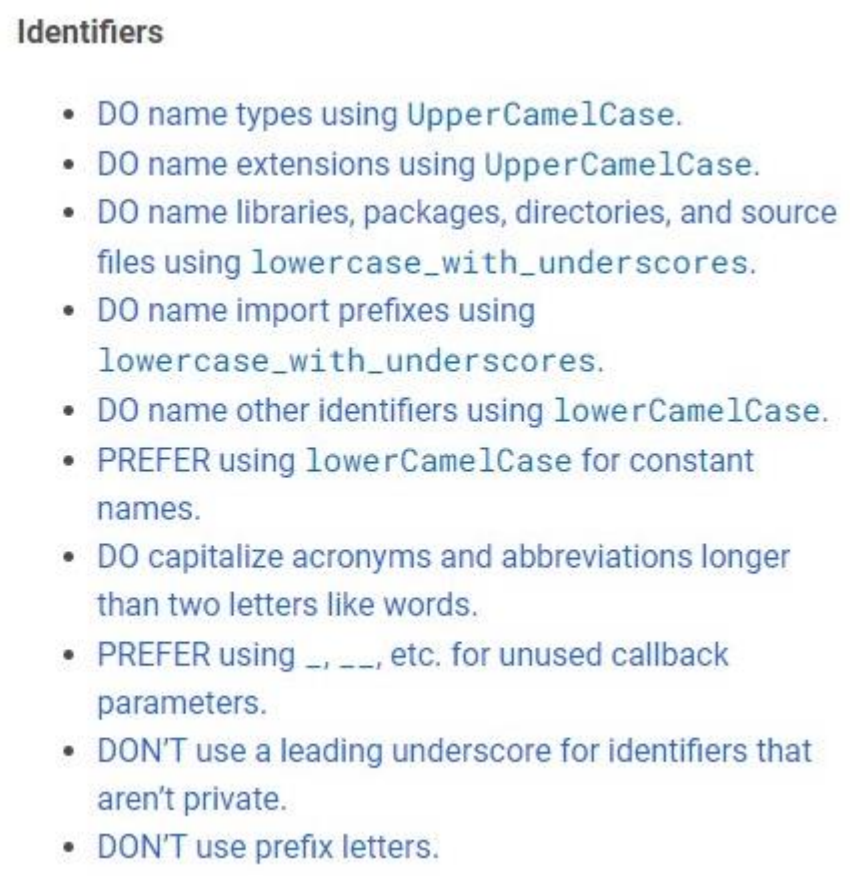
- Необходимо будет установить плагин flutter

Visual Studio Code:

- Поставить расширение для flutter

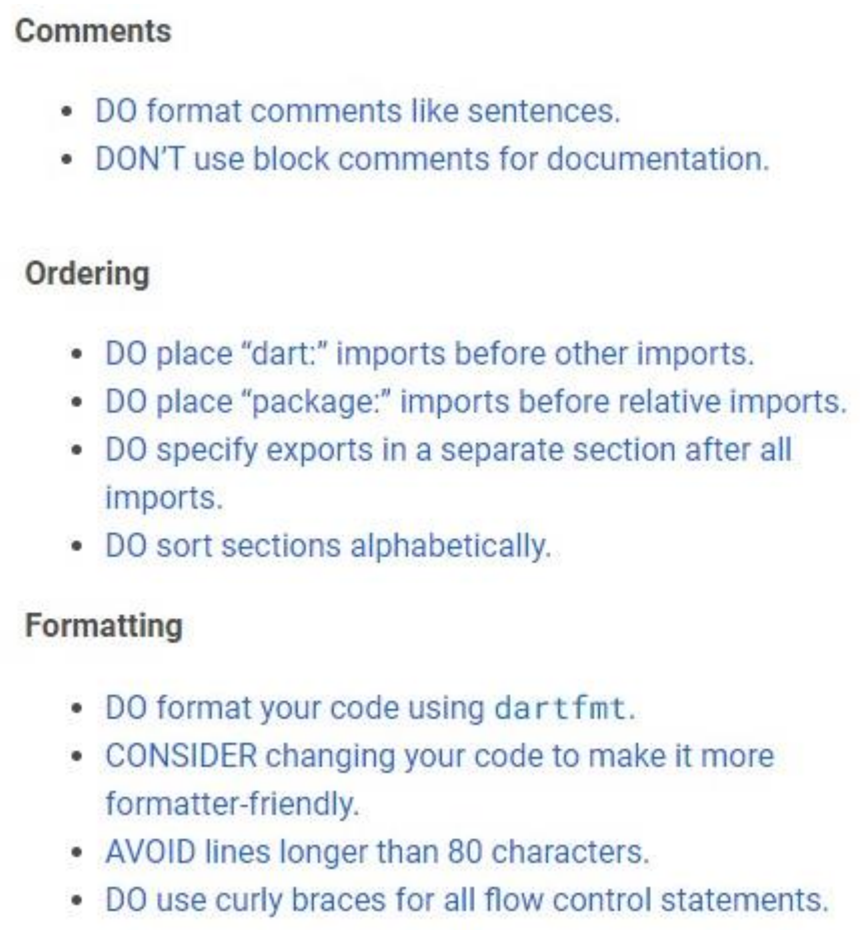
Code Style

https://dart.dev/guides/language/effective-dart



Identifiers

- DO name types using `UpperCamelCase`.
- DO name extensions using `UpperCamelCase`.
- DO name libraries, packages, directories, and source files using `lowercase_with_underscores`.
- DO name import prefixes using `lowercase_with_underscores`.
- DO name other identifiers using `lowerCamelCase`.
- PREFER using `lowerCamelCase` for constant names.
- DO capitalize acronyms and abbreviations longer than two letters like words.
- PREFER using _, __, etc. for unused callback parameters.
- DON'T use a leading underscore for identifiers that aren't private.
- DON'T use prefix letters.



Comments

- DO format comments like sentences.
- DON'T use block comments for documentation.

Ordering

- DO place "dart:" imports before other imports.
- DO place "package:" imports before relative imports.
- DO specify exports in a separate section after all imports.
- DO sort sections alphabetically.

Formatting

- DO format your code using `dartfmt`.
- CONSIDER changing your code to make it more formatter-friendly.
- AVOID lines longer than 80 characters.
- DO use curly braces for all flow control statements.

Doc comments

- DO use /// doc comments to document members and types.
- PREFER writing doc comments for public APIs.
- CONSIDER writing a library-level doc comment.
- CONSIDER writing doc comments for private APIs.
- DO start doc comments with a single-sentence summary.
- DO separate the first sentence of a doc comment into its own paragraph.
- AVOID redundancy with the surrounding context.
- PREFER starting function or method comments with third-person verbs.
- PREFER starting variable, getter, or setter comments with noun phrases.
- PREFER starting library or type comments with noun phrases.
- CONSIDER including code samples in doc comments.
- DO use square brackets in doc comments to refer to in-scope identifiers.
- DO use prose to explain parameters, return values, and exceptions.
- DO put doc comments before metadata annotations.

Markdown

- AVOID using markdown excessively.
- AVOID using HTML for formatting.
- PREFER backtick fences for code blocks.

Writing

- PREFER brevity.
- AVOID abbreviations and acronyms unless they are obvious.
- PREFER using “this” instead of “the” to refer to a member’s instance.

Libraries

- DO use strings in `part of` directives.
- DON'T import libraries that are inside the `src` directory of another package.
- DO use relative paths when importing libraries within your own package's `lib` directory.

Booleans

- DO use `??` to convert `null` to a boolean value.

Strings

- DO use adjacent strings to concatenate string literals.
- PREFER using interpolation to compose strings and values.
- AVOID using curly braces in interpolation when not needed.

Collections

- DO use collection literals when possible.
- DON'T use `.length` to see if a collection is empty.
- CONSIDER using higher-order methods to transform a sequence.
- AVOID using `Iterable.forEach()` with a function literal.
- DON'T use `List.from()` unless you intend to change the type of the result.
- DO use `whereType()` to filter a collection by type.
- DON'T use `cast()` when a nearby operation will do.
- AVOID using `cast()`.

Names

- DO use terms consistently.
- AVOID abbreviations.
- PREFER putting the most descriptive noun last.
- CONSIDER making the code read like a sentence.
- PREFER a noun phrase for a non-boolean property or variable.
- PREFER a non-imperative verb phrase for a boolean property or variable.
- CONSIDER omitting the verb for a named boolean *parameter*.
- PREFER the "positive" name for a boolean property or variable.
- PREFER an imperative verb phrase for a function or method whose main purpose is a side effect.
- PREFER a noun phrase or non-imperative verb phrase for a function or method if returning a value is its primary purpose.
- CONSIDER an imperative verb phrase for a function or method if you want to draw attention to the work it performs.
- AVOID starting a method name with `get`.
- PREFER naming a method `to___()` if it copies the object's state to a new object.
- PREFER naming a method `as___()` if it returns a different representation backed by the original object.
- AVOID describing the parameters in the function's or method's name.
- DO follow existing mnemonic conventions when naming type parameters.

Functions

- DO use a function declaration to bind a function to a name.
- DON'T create a lambda when a tear-off will do.

Parameters

- DO use `=` to separate a named parameter from its default value.
- DON'T use an explicit default value of `null`.

Variables

- DON'T explicitly initialize variables to `null`.
- AVOID storing what you can calculate.

Members

- DON'T wrap a field in a getter and setter unnecessarily.
- PREFER using a `final` field to make a read-only property.
- CONSIDER using `=>` for simple members.
- DON'T use `this.` except to redirect to a named constructor or to avoid shadowing.
- DO initialize fields at their declaration when possible.

Constructors

- DO use initializing formals when possible.
- DON'T type annotate initializing formals.
- DO use `;` instead of `{}` for empty constructor bodies.
- DON'T use `new`.
- DON'T use `const` redundantly.