Workshop IOT

Table of content

Introduction 4

Architecture: 4

Mobile Application (Flutter): 4

Backend (Laravel): 5

Firebase Cloud Messaging (FCM): 5

flutter_nearby_connections: 5

Workflow: 5

Requirements 7

System requirements for windows 7

System requirements for macOs 7

System requirements for Linux 7

Get the Flutter SDK 8

PATH For macOs or Linux: 8

Install Android Studio 8

Agree to Android Licenses 9

Install Android Studio 9

Install the Flutter and Dart plugins 9

Mac 9

Linux or Windows 9

Install VS Code 10

Install the Flutter and Dart plugins 10

Validate your setup with the Flutter Doctor 10

Test drive 11

Create the app 11

Run the app 11

Hot reload 12

Flutter Nearby Connections *12*

Introduction 12

Flutter 12

Use this package as a library 12

Depend on it 12

Import it 13

Android 13

IOS 13

Noted 13

In summary *15*

Introduction

The objective of this comprehensive workshop is to construct a robust contact tracing mobile application from the ground up. Our journey begins with setting up the development environment, followed by harnessing the power of Flutter as our primary framework for crafting the mobile app.

To detect nearby devices, we'll leverage the infrastructure Wi-Fi networks along with peer-to-peer Wi-Fi and Bluetooth Personal Area Networks (PAN). Flutter, equipped with a suite of essential packages, will serve as our toolkit to achieve this goal seamlessly. Specifically, we'll harness the capabilities of flutter_nearby_connections, enabling the communication of unique, momentary pairing codes between devices.

The server infrastructure plays a pivotal role in facilitating message exchanges among devices that identify the same pairing code. When a device detects a nearby counterpart, it transmits the pairing code to the Nearby Messages server for validation. This process includes a check for any pending messages to deliver within the application's current set of subscriptions. Notably, the Nearby Messages functionality operates without authentication, eliminating the necessity for a Google Account, prioritizing user accessibility and ease of use.

Architecture:

Mobile Application (Flutter):

1. UDID Integration:

   - Utilize device-specific UDID to uniquely identify each device without requiring user registration.

2. QR Code Integration:

   - Allow users to scan and store their COVID test certification via QR codes within the app.

3. Positive Test Reporting:

   - Provide a feature enabling users to flag themselves as COVID positive within the app.

4. Contact Tracing & Proximity Tracking:

   - Implement Bluetooth or GPS-based proximity tracking to log interactions without associating them with specific users.

Backend (Laravel):

1. API for Data Transmission:

   - Develop APIs to receive and store UDID data, QR code information, and COVID positive flags securely in the database.

2. Data Processing & Alert Generation:

   - Utilize algorithms to process contact logs, detect positive cases, and generate alerts for potentially exposed users.

3. Handling Positive Reports:

   - Design a mechanism to receive and process positive COVID reports from flagged devices, identify contacts, and notify affected users.

Firebase Cloud Messaging (FCM):

1. Push Notifications:

   - Integrate FCM to send push notifications to potentially exposed users, informing them about possible COVID exposure due to contact with a flagged device.

flutter_nearby_connections:

1. Proximity Interaction:

   - Implement flutter_nearby_connections for proximity-based communication between devices.

Workflow:

1. UDID Usage:

   - Replace user registration with UDID for device identification and management.

2. QR Code Storage:

   - Enable users to scan and store their COVID test certifications as QR codes within the app for easy retrieval.

3. Positive Test Reporting:

   - Implement a feature allowing users to self-report positive COVID tests. This action flags their device within the system.

4. Contact Tracing and Logging:

- Continue tracking interactions without associating them with specific users, maintaining logs of potential exposures.

5. Data Transmission & Processing:

   - Transmit encrypted data (UDID, QR code, positive test flags) securely to the backend via APIs for processing.

6. Alert Generation & Notifications:

   - Backend algorithms analyze data to detect positive cases and generate push notifications using FCM to notify potentially exposed users about the risk.

7. Proximity Interaction (flutter_nearby_connections):

   - Employ flutter_nearby_connections for direct communication between devices in close proximity, allowing information exchange without internet connectivity.

This adjusted architecture maintains privacy by avoiding user registration while utilizing UDID for device identification. It enables users to store their test certifications via QR codes and report positive COVID tests for the system to alert potentially exposed individuals without compromising personal information.

Requirements

System requirements for windows

To install and run Flutter, your development environment must meet these minimum requirements:

- **Operating Systems**: Windows 7 SP1 or later (64-bit), x86-64 based.
- **Disk Space**: 1.64 GB (does not include disk space for IDE/tools).
- **Tools**: Flutter depends on these tools being available in your environment.
    - Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
    - Git for Windows 2.x, with the **Use Git from the Windows Command Prompt** option.

If Git for Windows is already installed, make sure you can run `git` commands from the command prompt or PowerShell.

System requirements for macOs

To install and run Flutter, your development environment must meet these minimum requirements:

- Operating Systems: macOS
- **Disk Space**: 2.8 GB (does not include disk space for IDE/tools).
- **Tools**: Flutter uses `git` for installation and upgrade. We recommend installing Xcode, which includes `git`, but you can also install `git` separately.

System requirements for Linux

To install and run Flutter, your development environment must meet these minimum requirements:

- Operating Systems: Linux (64-bit)
- **Disk Space**: 600 MB (does not include disk space for IDE/tools).
- **Tools**: Flutter depends on these command-line tools being available in your environment.
    - bash
    - curl
    - file
    - `git` 2.x
    - mkdir
    - rm
    - unzip
    - which
    - xz-utils
    - zip
- **Shared libraries**: Flutter `test` command depends on this library being available in your environment.

  - `libGLU.so.1` - provided by mesa packages such as `libglu1-mesa` on Ubuntu/Debian and `mesa-libGLU` on Fedora.

Get the Flutter SDK

Download the installation bundle to get the latest stable release of the Flutter SDK:

https://storage.googleapis.com/flutter_infra_release/releases/stable/windows/flutter_windows_3.16.3-stable.zip

After downloading the SDK, you have to extract it, then you must add the folder path (flutter) to the your environment variable PATH:
For windows:

- From the Start search bar, enter 'env' and select **Edit environment variables for your account**.
- Under **User variables** check if there is an entry called **Path**:
  - If the entry exists, append the full path to `flutter\bin` using `;` as a separator from existing values.
  - If the entry doesn't exist, create a new user variable named `Path` with the full path to `flutter\bin` as its value.

PATH For macOs or Linux:

Execute the following command in your terminal:

```
export PATH="$PATH:`pwd`/flutter/bin"
```

At this point we can run the command "`flutter doctor`" to make sure that the previous configuration has been done correctly. You will notice that there are some missing dependencies (android sdk …) which is normal, therefore, the next thing to do is to install those missing dependencies:

Install Android Studio

1. Download and install Android Studio.
2. Start Android Studio, and go through the 'Android Studio Setup Wizard'. This installs the latest Android SDK, Android SDK Command-line Tools, and Android SDK Build-Tools, which are required by Flutter when developing for Android.
3. Run `flutter doctor` to confirm that Flutter has located your installation of Android Studio. If Flutter cannot locate it, run `flutter config --android-studio-dir <directory>` to set the directory that Android Studio is installed to.

Agree to Android Licenses

Before you can use Flutter, you must agree to the licenses of the Android SDK platform. This step should be done after you have installed the tools listed above.

1. Make sure that you have a version of Java 8 installed and that your `JAVA_HOME` environment variable is set to the JDK's folder.

Android Studio versions 2.2 and higher come with a JDK, so this should already be done.

2. Open an elevated console window and run the following command to begin signing licenses.

```
2.  flutter doctor --android-licenses
```

3. Review the terms of each license carefully before agreeing to them.
4. Once you are done agreeing with licenses, run `flutter doctor` again to confirm that you are ready to use Flutter.

Install Android Studio

Android Studio offers a complete, integrated IDE experience for Flutter.

- Android Studio, version 3.0 or later

Alternatively, you can also use IntelliJ:

- IntelliJ IDEA Community, version 2017.1 or later
- IntelliJ IDEA Ultimate, version 2017.1 or later

Install the Flutter and Dart plugins

The installation instructions vary by platform.

Mac

Use the following instructions for macos:

1. Start Android Studio.
2. Open plugin preferences (**Preferences** > **Plugins** as of v3.6.3.0 or later).
3. Select the Flutter plugin and click **Install**.
4. Click **Yes** when prompted to install the Dart plugin.
5. Click **Restart** when prompted.

Linux or Windows

Use the following instructions for Linux or WIndows:

1. Open plugin preferences (**File** > **Settings** > **Plugins**).

2. Select **Marketplace**, select the Flutter plugin and click **Install**.

Install VS Code

VS Code is a lightweight editor with Flutter app execution and debug support.

- VS Code, latest stable version

Install the Flutter and Dart plugins

1. Start VS Code.
2. Invoke **View > Command Palette…**.
3. Type “install”, and select **Extensions: Install Extensions**.
4. Type “flutter” in the extensions search field, select **Flutter** in the list, and click **Install**. This also installs the required Dart plugin.

Validate your setup with the Flutter Doctor

1. Invoke **View > Command Palette…**.
2. Type “doctor”, and select the **Flutter: Run Flutter Doctor**.
3. Review the output in the **OUTPUT** pane for any issues. Make sure to select Flutter from the dropdown in the different Output Options.

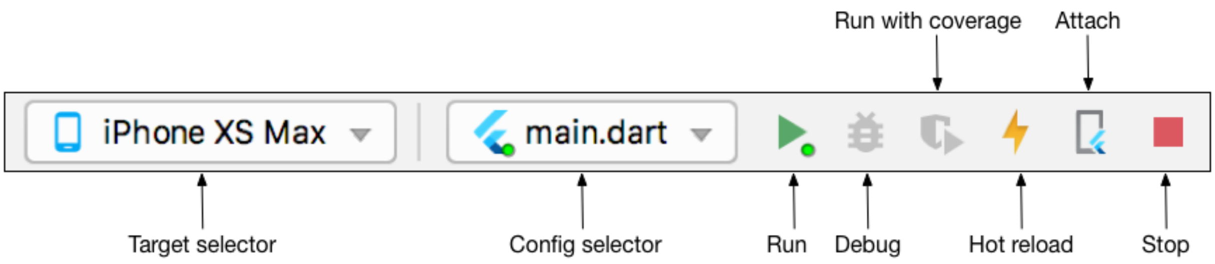
Test drive

Create the app

1. Open the IDE and select Create New Flutter Project.
2. Select **Flutter Application** as the project type. Then click **Next**.
3. Verify the Flutter SDK path specifies the SDK's location (select **Install SDK…** if the text field is blank).
4. Enter a project name (for example, `myapp`). Then click **Next**.
5. Click **Finish**.
6. Wait for Android Studio to install the SDK and create the project.

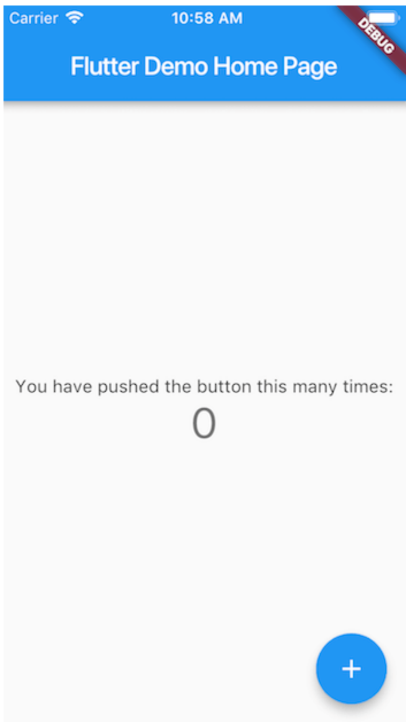
Run the app

1. Locate the main Android Studio toolbar:



2. In the **target selector**, select an Android device for running the app. If none are listed as available, select **Tools** > **AVD Manager** and create one there. For details, see Managing AVDs.
3. Click the run icon in the toolbar, or invoke the menu item **Run** > **Run**.

After the app build completes, you'll see the starter app on your device.



Hot reload

Flutter offers a fast development cycle with *Stateful Hot Reload*, the ability to reload the code of a live running app without restarting or losing app state. Make a change to app source, tell your IDE or command-line tool that you want to hot reload, and see the change in your simulator, emulator, or device.

Flutter Nearby Connections

Introduction

Nearby Connections enables advertising, discovery, and connections between nearby devices in a fully-offline peer-to-peer manner. Connections between devices are high-bandwidth, low-latency, and fully encrypted to enable fast, secure data transfers.

A primary goal of this API is to provide a platform that is simple, reliable, and performant. Under the hood, the API uses a combination of Bluetooth, BLE, and Wifi hotspots, leveraging the strengths of each while circumventing their respective weaknesses. This effectively abstracts the vagaries of Bluetooth and Wifi across a range of Android OS versions and hardware, allowing developers to focus on the features that matter to their users.

As a convenience, users are not prompted to turn on Bluetooth or Wi-Fi — Nearby Connections enables these features as they are required, and restores the device to its prior state once the app is done using the API, ensuring a smooth user experience.

Flutter

Flutter plugin supports peer-to-peer connectivity and discovers nearby devices for Android and IOS

The flutter_nearby_connections plugin supports the discovery of services provided by nearby devices. Moreover, the flutter_nearby_connections plugin also supports communicating with those services through message-based data, streaming data, and resources (such as files). The framework uses infrastructure Wi-Fi networks, peer-to-peer Wi-Fi and Bluetooth Personal Area Networks (PAN) for the underlying transport over UDP.

Use this package as a library

Depend on it

Run this command:

With Flutter:

```
 $ flutter pub add flutter_nearby_connections
```

This will add a line like this to your package's pubspec.yaml (and run an implicit `flutter pub get`):

dependencies:

  flutter_nearby_connections: ^1.1.2

Alternatively, your editor might support or `flutter pub get`. Check the docs for your editor to learn more.

Import it

Now in your Dart code, you can use:

import 'package:flutter_nearby_connections/flutter_nearby_connections.dart';

Android

Nearby Connections API (Bluetooth & hotspot) Support Strategy: ***Strategy.P2P_CLUSTER***, ***Strategy.P2P_STAR***, ***Strategy.P2P_POINT_TO_POINT***

Wi-Fi P2P (only WIFI hotspot no internet) Support Strategy: ***Strategy.Wi_Fi_P2P***

IOS

Multipeer Connectivity

We use the NearbyConnections API, but Flutter methods are based on the concept of Multipeer Connectivity IOS.

Methods provided:

startAdvertisingPeer, startBrowsingForPeers, stopAdvertisingPeer

We separate the dependencies of the MCNearbyServiceAdvertiser, MCNearbyServiceBrowser and MCSession classes. All of the methods will be implemented in the NearbyService class.

Noted

- Android doesn't support emulator only support real devices
- On iOS 14, need to define in Info.plist

```
<key>NSBonjourServices</key>

<array>

    <string>_{YOUR_SERVICE_TYPE}._tcp</string>
</array>

<key>UIRequiresPersistentWiFi</key>

<true/>

<key>NSBluetoothAlwaysUsageDescription</key>

<string>{YOUR_DESCRIPTION}</string>
```

in this case, YOUR_SERVICE_TYPE is 'mp-connection' (you can define it)

```
nearbyService.init(

        serviceType: 'mp-connection',

        strategy: Strategy.P2P_CLUSTER,
```

In summary

By this stage, we are poised to create a mobile application that harnesses the available connection infrastructure effectively. Flutter, as a framework, stands out for its rich repository of pre-defined packages and features, empowering developers to construct a diverse range of applications seamlessly. Its versatility significantly diminishes the entry barrier for app development, streamlining the process and mitigating the costs and complexities associated with creating apps across various platforms.

Flutter's design ethos emphasizes cross-platform functionality, catering not only to Android and iOS but also extending its reach to interactive web-based applications. Additionally, while desktop support remains in beta, a stable channel snapshot is accessible, highlighting Flutter's commitment to expanding its scope and potential across diverse platforms.

If you want to discover more regarding flutter's packages, you can visit this website https://pub.dev.

To learn more about flutter, you can visit the official website https://docs.flutter.dev/ .