JOSH

Empathize, Empower & Envision

A STUDY OF PERFORMANCE & LOOKS

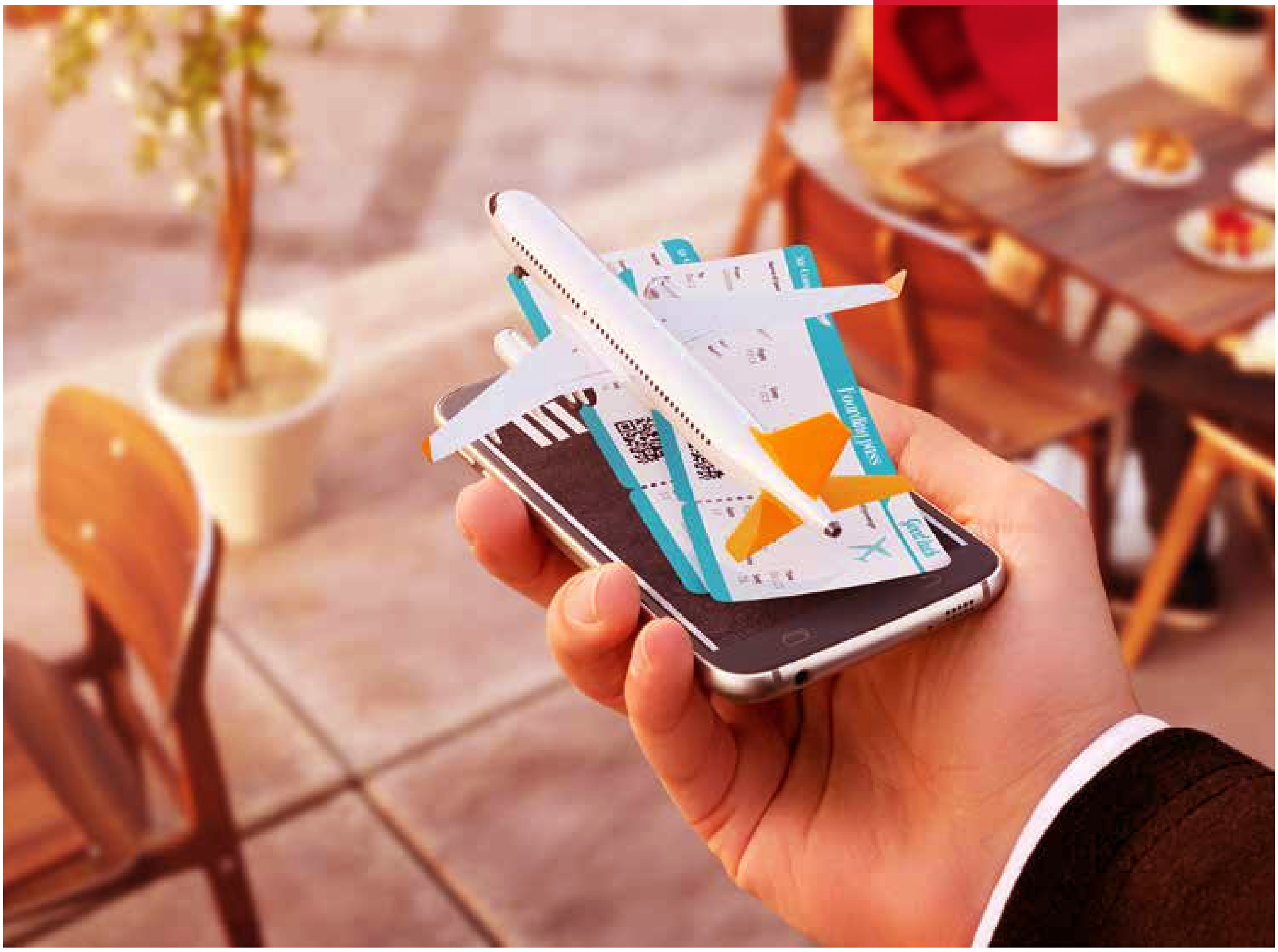
BETWEEN FLUTTER + FIREBASE APPLICATIONS

Flutter

Google developed Flutter, and the first beta release was in February 2018. By May 2018, Flutter made it to GitHub's top 100 repos. Google Flutter 1.0 was released in December 2018 and was stable and ready for production.

Flutter has been exponentially increasing in popularity and use over the last year, as evidenced in the 200 active pull requests on GitHub (as of September 2021). Additionally, Google Trends demonstrates there is rising curiosity and awareness about Flutter.

As of March 2021, Flutter 2.0 is active and supports the following platforms: Android, iOS, web, macOS, Linux, and Windows. Here are a few features that make Flutter a growing developer favorite.



#01

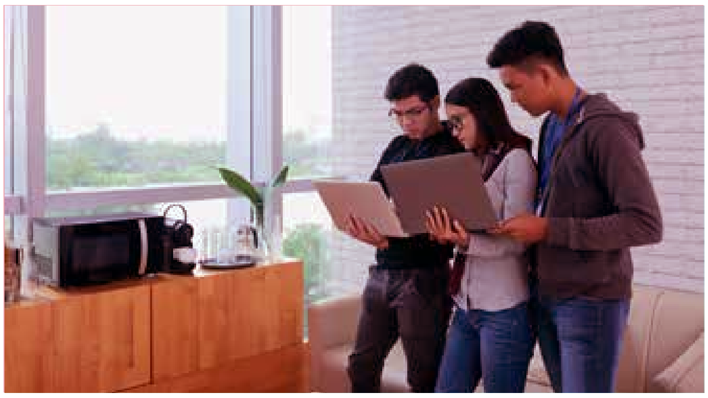
Hot Reload

Developers can view changes to the code in real-time without losing the current application state.

#02

Single Codebase

Developers can reuse the native codebase across multiple platforms with minimal modifications. Apps that target mobile, web, desktop, and embedded devices from a single codebase can be quickly built.



#03

Testing & QA

The reduced diversity of codes speeds up the testing and quality assurance process.

#04

Compatibility for Advanced UI

Skia, the internal graphics engine, is used in Mozilla Firefox, Google Chrome, and Sublime Text 3. With internal Apple Design System elements and Material UI components, Flutter creates structural & stylistic elements to the layout for a seamless UI/UX across multiple platforms.

Additionally, there are several ready-to-use and customizable widgets [Material and Cupertino]. This proves useful when building a Minimum Viable Product (MVP).

What does this add up to?

Faster Time-to-Market

- 2D-based UI
- Eliminates interacting with a native application counterpart
- Saves time in development

Smooth UI/UX Across Platforms

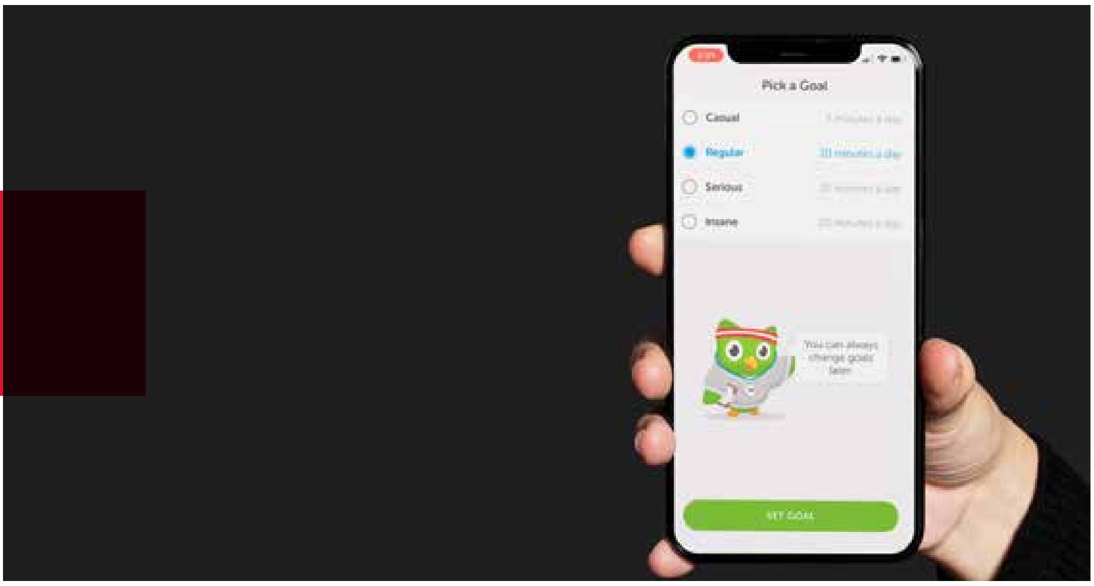
- Declarative API for building UI
- Improves performance of the application
- Ease of visual adjustments
- Both the UI code, app logic, and the UI are shared
- Dependence on platform-specific components eliminated
- UI is rendered by populating elements of the application UI on canvas

Consistent Performance

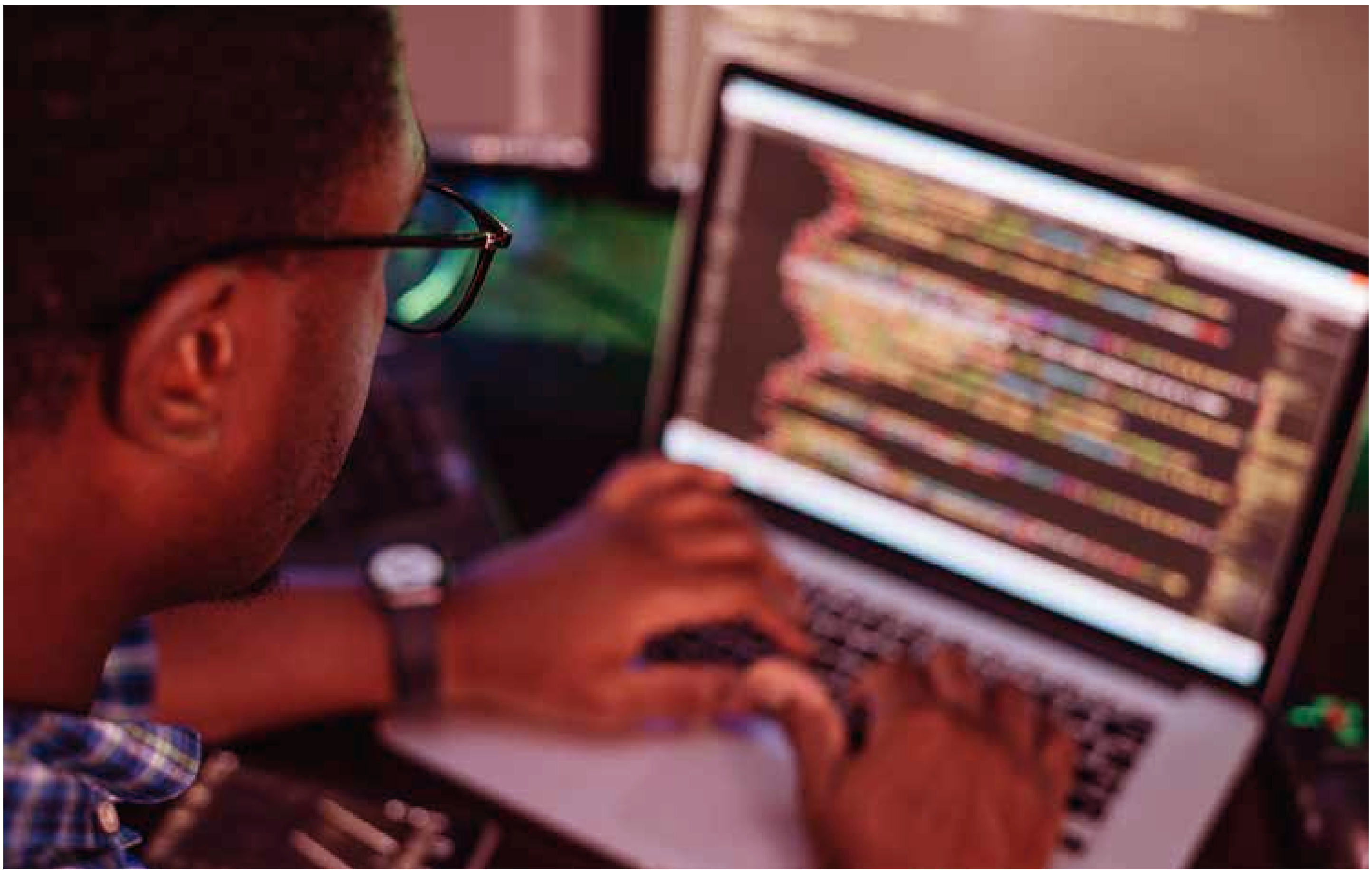
- Complex UI animation is possible
- No intermediate code representations or interpretation required
- Apps built directly into the machine code
- Apps will be fully compiled well beforehand

Platform-Specific Logic Implementation

- Ready-to-use plugins for advanced OS features
- Smooth communication between platform-native code and Dart
- Easy to implement advanced features of native-app on Flutter



Firebase



FIREBASE IS A GOOGLE-BACKED APPLICATION DEVELOPMENT SOFTWARE BUILT TO CONTROL REAL-TIME, COLLABORATIVE APPLICATIONS. THIS ALLOWS ACCESS TO A SHARED DATA STRUCTURE LEADING TO ANY CHANGES TO THE DATA AUTOMATICALLY SYNCHRONIZED WITH THE FIREBASE CLOUD AND OTHER CLIENTS WITHIN MILLISECONDS.

THIS ENABLES DEVELOPERS TO DEVELOP IOS, ANDROID, AND WEB APPS WHILE PROVIDING TOOLS FOR:

- Tracking analytics
- Reporting and fixing app crashes
- Creating marketing and product experiments

What does it offer?

Google Analytics for Firebase

- Free, unlimited reporting on 500 separate events
- Displays data about user behavior in iOS and Android
- Improved decision-making
- Enhanced performance

Firebase Authentication

- Simplifies building secure authentication systems
- Enhances the sign-in and onboarding UX
- Complete identity solution [supporting email and password accounts, phone auth, as well as Google,
- Facebook, GitHub, Twitter login]

Firebase Cloud Messaging

- Cross-platform messaging tool
- Free, secure end-to-end messaging
- iOS, Android, and the web

Firebase Realtime Database

- Cloud-hosted NoSQL database
- Enables data storage
- Data sync across all users in real-time
- Offline data availability

Firebase Crashlytics

- Real-time crash reporter
- Helps developers enhance and maintain app quality
- Monitor and debug easily
- Reduces time on troubleshooting crashes

Firebase Performance Monitoring

- Insight into performance characteristics of iOS and Android apps
- Identify areas for improvement

Firebase Test Lab

- Cloud-based app-testing infrastructure
- One operation to test iOS or Android apps across devices and configurations.
- Results [videos, screenshots, logs] seen in the console

Flutter as a basic CRUD

CREATE, READ, UPDATE, DELETE [CRUD] OPERATIONS ARE FUNDAMENTAL IN ANY LANGUAGE OR FRAMEWORK. HERE, WE ARE GOING TO TAKE YOU THROUGH THE FLUTTER CRUD OPERATIONS WITH FIREBASE AS THE BACKEND DATABASE.

1. Create a new flutter application to begin.

In this example, we are using Android Studio as the IDE. You could use VScode or any other as well. These are the steps to implement an application, with the functionalities of data insert, read, delete and update.

Here is the sketch of the UI of the main.dart file:

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

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

class MyApp extends StatelessWidget {
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     home: MyHomePage(),
   );
 }
}

class MyHomePage extends StatefulWidget {
 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text("Flutter CRUD with Firebase"),
     ),
     body: Center(
       child: Column(mainAxisAlignment: MainAxisAlignment.center, children:
<Widget>[
         RaisedButton(
           child: Text("Create"),
         ),
         RaisedButton(
           child: Text("Read"),
         ),
         RaisedButton(
           child: Text("Update"),
         ),
         RaisedButton(
           child: Text("Delete"),
         ),
       ]),
     ),
   );
 }
}
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     home: MyHomePage(),
   );
 }
}

class MyHomePage extends StatefulWidget {
 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text("Flutter CRUD with Firebase"),
     ),
     body: Center(
       child: Column(mainAxisAlignment: MainAxisAlignment.center, children:
<Widget>[
         RaisedButton(
           child: Text("Create"),
         ),
         RaisedButton(
           child: Text("Read"),
         ),
         RaisedButton(
           child: Text("Update"),
         ),
         RaisedButton(
           child: Text("Delete"),
         ),
       ]),
     ),
   );
 }
}
```

2. Create a new firebase project

Configure the above Android project with the Firebase project

3. Add a new android app to the firebase project

a. Register the app
b. Download config file
c. Add Firebase SDK
d. Go to cloud Firestore - Test Mode
e. Create a new collection and data path to store data

4. Come back to the Android studio project and import the Firebase dependencies to pubspec.yaml file.

```
dependencies:
 flutter:
   sdk: flutter

 cloud_firestore:
 firebase_core :
```

5. Build the four CRUD functions in the main.dart.

a. Import the libraries

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_core/firebase_core.dart';
```

b. Initialize the firebase inside your main function

```
void main() async {
 WidgetsFlutterBinding.ensureInitialized();
 await Firebase.initializeApp();
 runApp(MyApp());
}
```

c. To avoid the below error message

```
[core/no-app] No Firebase App '[DEFAULT]' has been created — call Fire-
base.initializeApp()
```

d. Create and initialize a Firestore instanceImport the libraries

```
final FirebaseFirestore firestore = FirebaseFirestore.instance;
```

6. Implement the Firestore CRUD methods

a. Create Data

```
void _create() async {
 try {
   await firestore.collection('users').doc('testUser').set({
     'firstName': 'John',
     'lastName': 'Peter',
   });
 } catch (e) {
   print(e);
 }
}
```

b. Read Data

```
void _read() async {
 DocumentSnapshot documentSnapshot;
 try {
   documentSnapshot = await   firestore.collection('users').doc('testUser').
get();
   print(documentSnapshot.data());
 } catch (e) {
   print(e);
 }
}
```

c. Update Data

```
void _update() async {
 try {
   firestore.collection('users').doc('testUser').update({
     'firstName': 'Alan',
   });
 } catch (e) {
   print(e);
 }
}
```

d. Delete Data

```
void _delete() async {
 try {
   firestore.collection('users').doc('testUser').delete();
 } catch (e) {
   print(e);
 }
}
```

7. Implement the onpress command of each button. The main. dart file looks as follows

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:flutter/material.dart';

void main() async {
 WidgetsFlutterBinding.ensureInitialized();
 await Firebase.initializeApp();
 runApp(MyApp());
}

class MyApp extends StatelessWidget {
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     home: MyHomePage(),
   );
 }
}

class MyHomePage extends StatefulWidget {
 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 final FirebaseFirestore firestore = FirebaseFirestore.instance;


 void _create() async {
   try {
     await firestore.collection('users').doc('testUser').set({
       'firstName': 'John',
       'lastName': 'Peter',
     });
   } catch (e) {
     print(e);
   }
 }

 void _read() async {
   DocumentSnapshot documentSnapshot;
   try {
     documentSnapshot = await firestore.collection('users').doc('testUser').
get();
     print(documentSnapshot.data);
   } catch (e) {
     print(e);
   }
 }

 void _update() async {
   try {
     firestore.collection('users').doc('testUser').update({
       'firstName': 'Alan',
     });
   } catch (e) {
     print(e);
   }
 }

 void _delete() async {
   try {
     firestore.collection('users').doc('testUser').delete();
   } catch (e) {
     print(e);
   }
 }

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text("Flutter CRUD with Firebase"),
     ),
     body: Center(
       child: Column(mainAxisAlignment: MainAxisAlignment.center, children:
<Widget>[
         RaisedButton(
           child: Text("Create"),
           onPressed: _create,
         ),
         RaisedButton(
           child: Text("Read"),
           onPressed: _read,
         ),
         RaisedButton(
           child: Text("Update"),
           onPressed: _update,
         ),
         RaisedButton(
           child: Text("Delete"),
           onPressed: _delete,
         ),
       ]),
     ),
   );
 }
}
```

Integrating Flutter + Firebase

HERE IS A SIMPLIFIED FLOW CHART OF THE STEPS INVOLVED:

1. Install Flutter SDK on your system. Here is the official documentation to guide you.

2. Open Visual studio code and execute the following command:

```
flutter create firebase_with_flutter
```

3. This command will create a new project with the name firebase_with_flutter. Then to go to the directory of the project, execute the following:

```
cd firebase_with_flutter
code.
```

NOW, IF THE ABOVE ARE ALL EXECUTED WITH 'FLUTTER RUN' THIS WILL CREATE A NEW APPLICATION ON YOUR DEVICE.

4. Now, we integrate Firebase into the project.

a. Open the Firebase console
b. Create a new project
c. Click on the Android icon
d. Start adding information related to the project.
e. You can find the package name under the following path android\app\src\main\AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.firebase_with_flutter">
```

f. Download the Firebase Android config file google-services.json
g. Add it under android/app. [Refer **this** to know where]
h. Navigate to the android/build.gradle

```
buildscript {
 repositories {
   // Check that you have the following line (if not, add it):
   google()  // Google's Maven repository
 }
 dependencies {
   ...
   // Add this line
   classpath 'com.google.gms:google-services:4.3.3'
 }
}

allprojects {
 ...
 repositories {
   // Check that you have the following line (if not, add it):
   google()  // Google's Maven repository
   ...
 }
}
```

i. Add the google maven repository and the google-services classpath:
j. Add the following inside your app android\app\build.gradle:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'com.google.gms.google-services' //this line
```

WELL DONE! YOU CAN NOW USE THE FIREBASE REALTIME DATABASE IN YOUR PROJECT!

Apps using Firebase

PicCollage
OVER 120MN DOWNLOADS.

Fabulous
ALEXA RANK, <620K GLOBALLY.

Shazam
OVER 120MN USERS EACH MONTH.

Skyscanner
OVER 50MN MONTHLY USERS.

New York Times
OVER 10MN DOWNLOADS.

Duolingo
OVER 100MN DOWNLOADS.

NPR One

Venmo

Half Brick

Trivago

The Economist

Pros & Cons

Pros of Firebase

- Expansive database capabilities
- Broad services and features
- Clear, simple documentation
- Rapid, easy integration and setup
- Free Basic Plan

Cons of Firebase

- Limited querying ability
- Restricted data migration
- Platform dependence
- Android centered
- Less support for iOS

Best use cases for Firebases

Here is a list of contexts and situations that we believe are best suited for Firebase.

#01

Sharing Data

If data needs to be shared globally with various clients/customers, Firebase is the way to go. With great data storage options, pertinent information can be easily allocated to different users.

#02

Handling < 1 million Connections

The Firestore Cloud is capable of processing 1 million concurrent connections. This would be of great help in building an app with limited sorting and filtering queries.

#03

Simple Apps

Applications that require basic options for integration with third-party tools/services without complex authentication or the processing of large volumes of information will benefit from Firebase.

#04

Real-time features

Applications where real-time features like notifications, chat, or real-time feed are required make a good use case for Firebase. Especially in cases where other parts of the code should not be altered. For instance, the streaming platform Twitch.

#05

Rapid Delivery

The simple framework leads to faster development making this a particularly viable option for building MVPs and prototypes.

#06

Smooth Integration

Tools that enhance efficiency such as Data Studio, Google Ads, BigQuery, AdMob, and Play Store can be integrated easily. Additionally, Google Analytics is automatically integrated, ensuring a seamless UX and more intelligent marketing strategies.

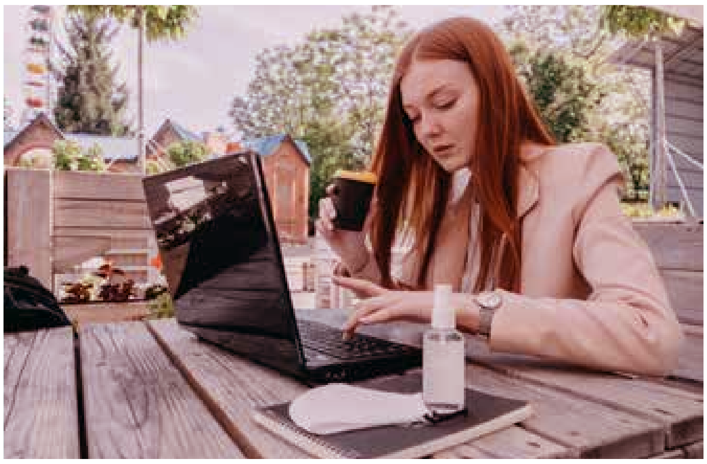
Now that we have simplified when and where to use Firebase, what about when not to use it?

When not to use Firebase

#01

Handling Complex Queries

As it works on a flat hierarchy, Firebase is not built for processing complex queries. This includes inverting the order of certain items that cannot be executed using Firebase. Additionally, concurrency might lead to inconsistent performance when offline—for instance, ERPs, Multi-workspace multi-user SaaS.



#02

Zero Data Sharing

With a limited set of security standards and rules. The focus is predominantly on data sharing across multiple platforms and users. Unless this is the requirement, Firebase is not built for zero data sharing.

#03

Microservice Integration

With data caches in memory, the processes slow down with time. This makes integrating microservices tough.

#04

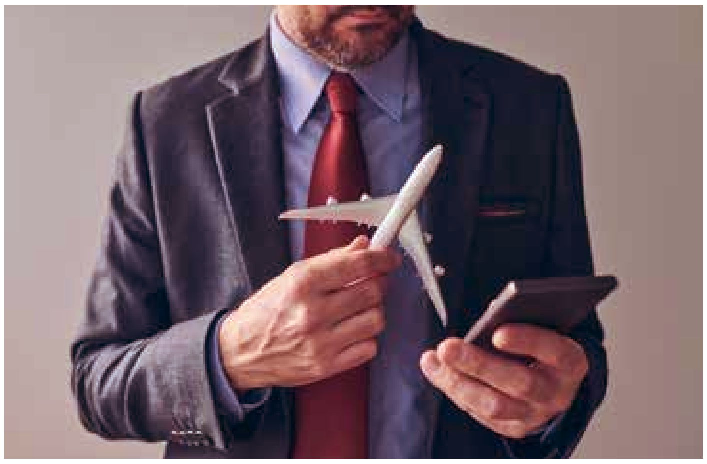
Business Intelligence Functionalities

Firebase does not have the framework to support business intelligence solutions.

#05

Rapid Delivery

There is no guarantee of data integrity with dynamic data structures similar to JSON [free of form as it is a NoSQL database]. This means information at the database level cannot be constrained while keeping the business logic at the code level. This means if something is not handled correctly, bugs will be inevitable, leading to chaotic data.



Josh Software

Josh Software is a recognized leader in Flutter app development. We specialize in helping clients get ahead by leveraging emerging software and technology. Start your Flutter development journey with us today.