Firebase for Flutter

Last update: 2/26/2018

Summary	Flutter is a mobile app development SDK for developing high-fidelity apps across iOS and Android. Firebase is Google's mobile platform for deploying and hosting mobile apps. In this codelab, you'll learn how to enable Firebase features in your Flutter app. You'll learn how to implement Firebase authentication and analytics, and how to share data with other users via a realtime database. You'll use Flutter's platform APIs to access data on a mobile device, then upload the data to Google Cloud Storage.
URL	flutter-firebase
Category	Firebase
Status	Draft
Environment	web, kiosk, io2017, gdd17, firebase17, devfest-lon
Feedback Link	https://github.com/flutter/flutter/issues
Analytics Account	

Overview

Duration: 1:00

In this codelab, you'll learn how to enable Firebase features in an existing simple Flutter app, Friendlychat. You'll also enhance the app to send and receive images stored on a mobile device, in addition to text.

The steps for building the Flutter version of Friendlychat are described in an earlier codelab, Building Beautiful UIs with Flutter. Both codelabs use the same software environment: the Flutter SDK, a set of development tools for iOS and another set for Android, and an IntelliJ IDE. If your environment is already set up for Flutter development, you can skip to Open Friendlychat in IntelliJ.

If you're familiar with the Firebase codelabs, the Flutter codelabs demonstrate how to build a similar chat client also named Friendlychat, using Flutter. With Flutter, you can build Friendlychat for iOS and Android at the same time, from a single code base, and iterate quickly using developer tools like hot reload.

Set up your Flutter environment

Duration: 15:00
Environment: web

[[**import** Flutter Set Up_Web]]
[[**import** Flutter Set Up_Kiosk]]

Set up your Flutter environment

Duration: 1:00
Environment: kiosk

[[**import** Flutter Set Up_Kiosk]]

Get the sample code

Duration: 1:00

This codelab begins with the final version of the Friendlychat Flutter app. If you have not completed the Building Beautiful UIs with Flutter codelab, you need to download the sample code to follow this codelab.

If you want to view the samples as reference or start the codelab at a specific section, clone the `flutter/friendlychat-steps` repo and open the `full_steps` folder. We have created snapshots for you for each step, one snapshot per directory. Each step builds on the preceding step.

The steps are organized into these folders:

- **step_0_offline**—The starting code that you'll build upon in this codelab.
- **step_1_set_up_firebase**—Edit configuration files to set up Firebase integration. This is the starting code for section 6: 'Sign users into your app'.
- **step_2_sign_in**—Add a plugin and logic to authenticate users with Google Sign-In. This is the starting code for section 7: 'Track user activity'.
- **step_3_analytics**—Gather metrics on the number of login events and chat messages sent. This is the starting code for section 8: 'Authenticate users'.
- **step_4_auth**—Add security and restrictions for the contents of your realtime database. This is the starting code for section 9: 'Enable data syncing'.

- **step_5_data_syncing**—Store chat messages in a realtime database. This is the starting code for section 9: 'Add image support'.
- **step_6_image_support**—Access images stored on the device.

Open Friendlychat in IntelliJ

Duration: 2:00

Open the starting code for this codelab:

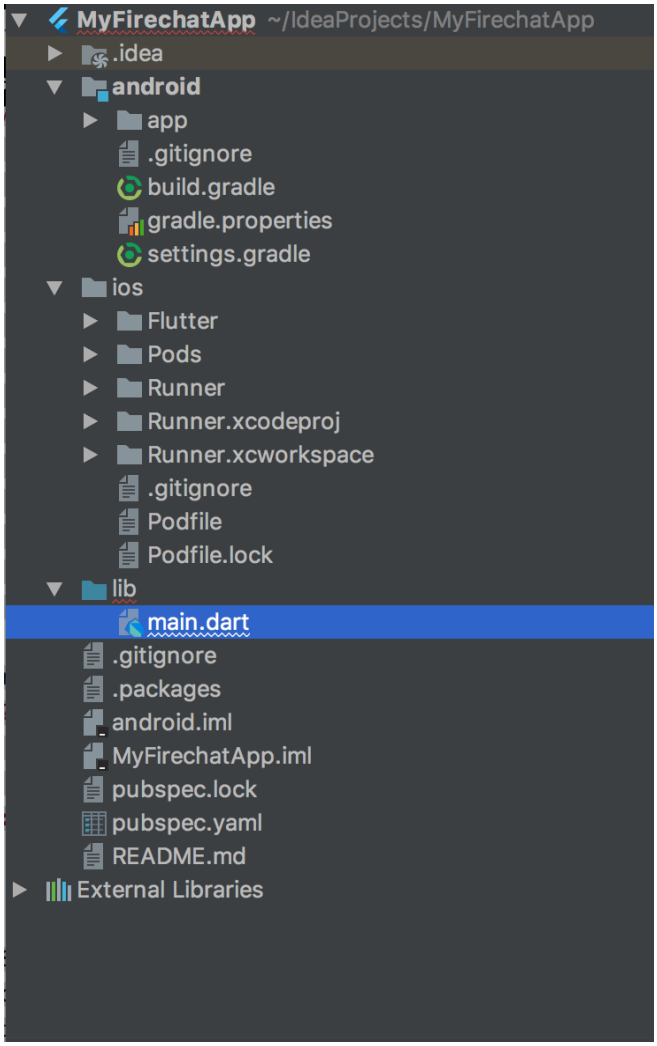
1. Launch the IDE.
2. Select **Open** in the Welcome screen or **File>Open** and navigate to the `friendlychat-steps/full_steps/step_0_offline` directory. The directory should contain a `pubspec.yaml` file.
3. Click **Open.**

Set up Firebase integration

Environment: web, kiosk

In this step, you'll configure your Flutter project for Firebase. You'll install dependencies, create a new Firebase project, and work with files in your Flutter project.

Creating a new Flutter project in IntelliJ makes a new project directory and generates the files for a basic Flutter sample app. Here's an overview of some of the important files:



`/lib/main.dart`	The main entry point for your Flutter app. This source file is written in Dart syntax. You'll revisit this file as you progress through this codelab.

`/ios/Runner/Assets.xcassets`
`/ios/Runner/Info.plist`
`/ios/Runner/Base.lproj/LaunchScreen.storyboard`
	These files are required to run your Flutter app on iOS.
	● The Asset catalog folder (`Assets.xcassets`) contains app resources such as icons for your app.
	● The Information Property List file (`Info.plist`) contains configuration information for running your app. In this step, to enable Google Sign-In you'll add a custom `Info.plist` file that Firebase generates. In a later step, you'll modify this file to let

	Friendychat access the device's camera and stored images. • The `LaunchScreen.storyboard` file controls the launch screen that users see when starting your app.
`/android/app/src/main/AndroidManifest.xml`	This file is required to run your Flutter app on Android. The manifest file describes the fundamental characteristics of the app and defines each of its components. In this step, you'll copy the package name from this file to your Firebase project. In a later step, you'll modify this file to let Friendychat access the device's camera and stored images.
`pubspec.yaml`	This file defines information about your project and its dependencies. It also contains metadata such as the Flutter app version and any third-party services or assets that your app uses. To learn more about its format, see Pubspec Format. In this step, you'll add Firebase plugin dependencies to this file.
`README.md`	This file contains documentation for your app, written in Markdown syntax. You won't need to modify this file for this codelab.

Install CocoaPods

Duration: 2:00
Environment: web

If you're developing for iOS, to use the Firebase plugins with your Flutter app you'll need some additional tools:

1. Install homebrew.
2. Open the terminal and run these commands to install CocoaPods:

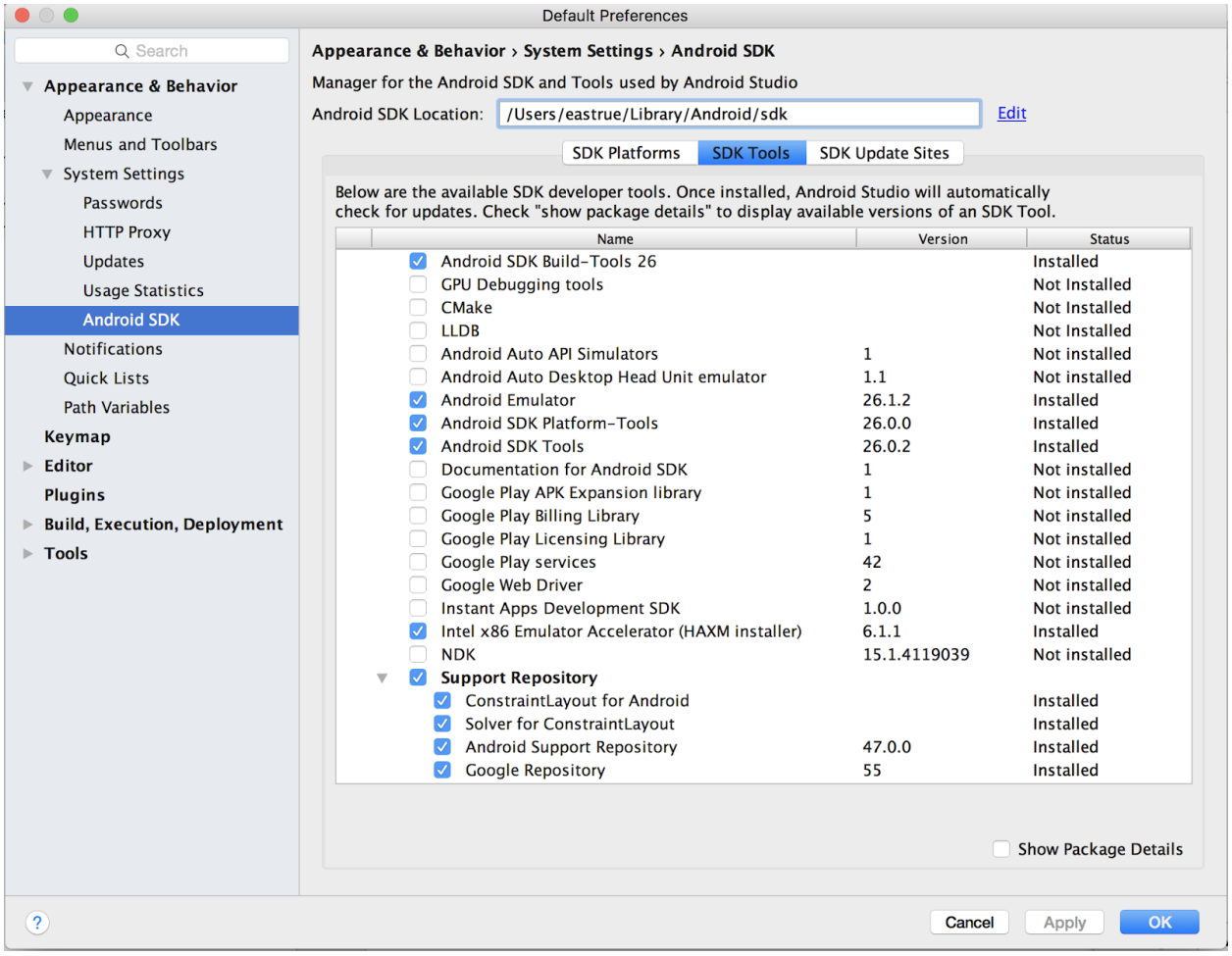
```
brew install cocoapods
pod setup
```

Install Google Repository

Duration: 2:00
Environment: web

If you're developing for Android, to use the Firebase plugins with your Flutter app you'll need Google Repository.

- In **Android Studio > Tools > Android > SDK Manager > SDK Tools**, verify that Google Repository is installed.



z

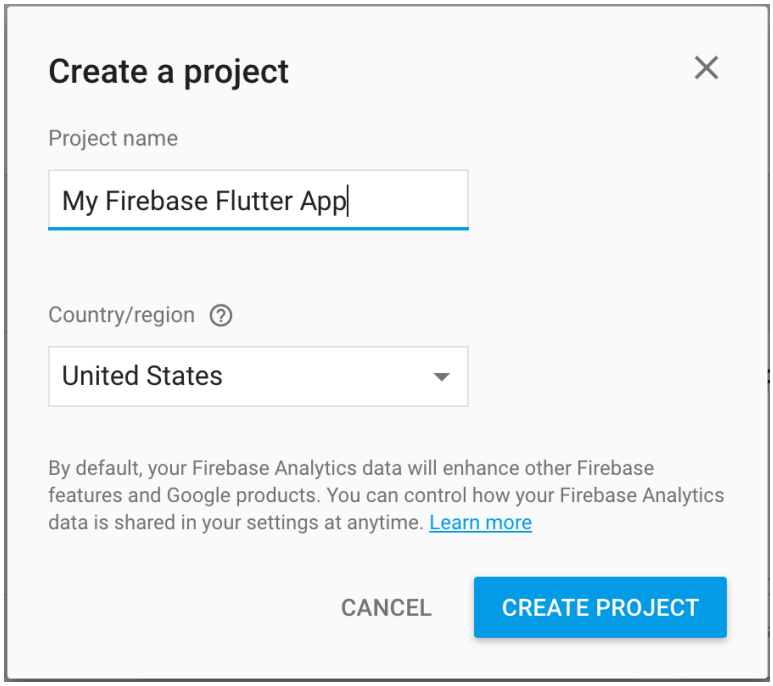
If it is not already installed, select **Google Repository > Apply**.

Create a new Firebase project

Environment: web, kiosk

The first step is to create a new Firebase project, which automatically generates a realtime database.

1. Open the Firebase console in your browser.
2. Select **Add project**.
3. Name your project 'My Firebase Flutter App' and set the Country/region, then click **Create Project.**



Add platform-specific Firebase configuration information

Environment: web, kiosk

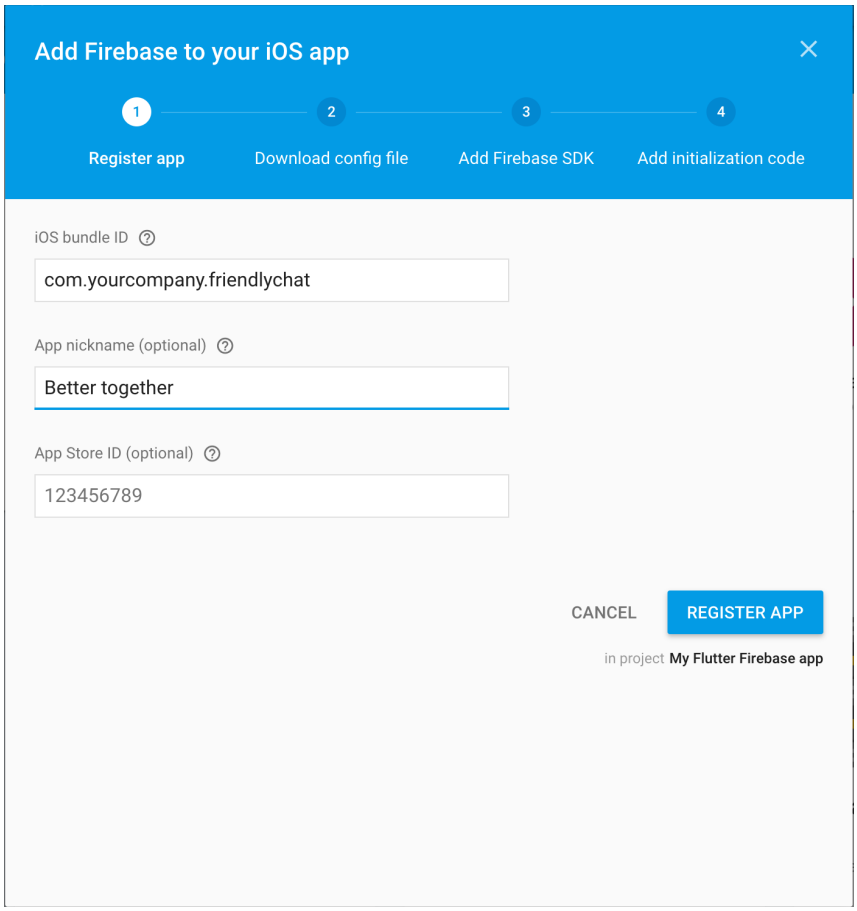
Next, to support sending and receiving chat messages between iOS and Android devices, you'll add the platform-specific configuration files generated by the Firebase console to your IntelliJ project. The Firebase wrapper for Flutter package, which you'll add later in this codelab, uses this configuration information to connect to the shared database.

iOS Configure Firebase for iOS

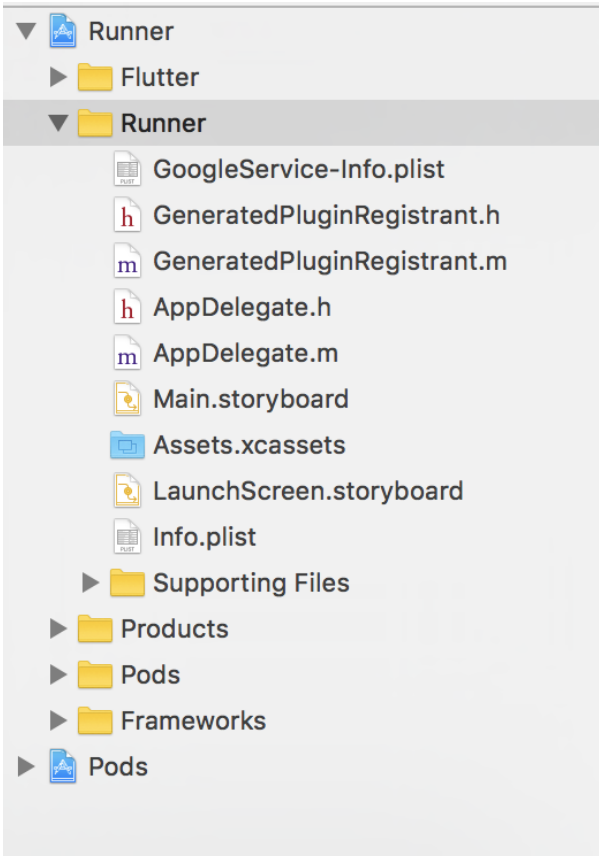
To add the Firebase configuration information for iOS to your Flutter project:

1. In Xcode, open the `Runner.xcworkspace` file by running `open ios/Runner.xcworkspace` in a terminal window.

2. Click **Runner** in the project menu at upper left to display the General tab, then copy the string value in the Bundle Identifier field. For the codelab, you'll use *com.yourcompany.friendlychat* but you can use a different string in your own app. If you're viewing the configuration file, this field corresponds to the `CFBundleIdentifier` key.
3. In the Firebase console, open your Firebase project, then click **Add Firebase to your iOS app**.
4. In the **iOS bundle ID** field, specify the string value you copied from step 2. Then click **Register App**.



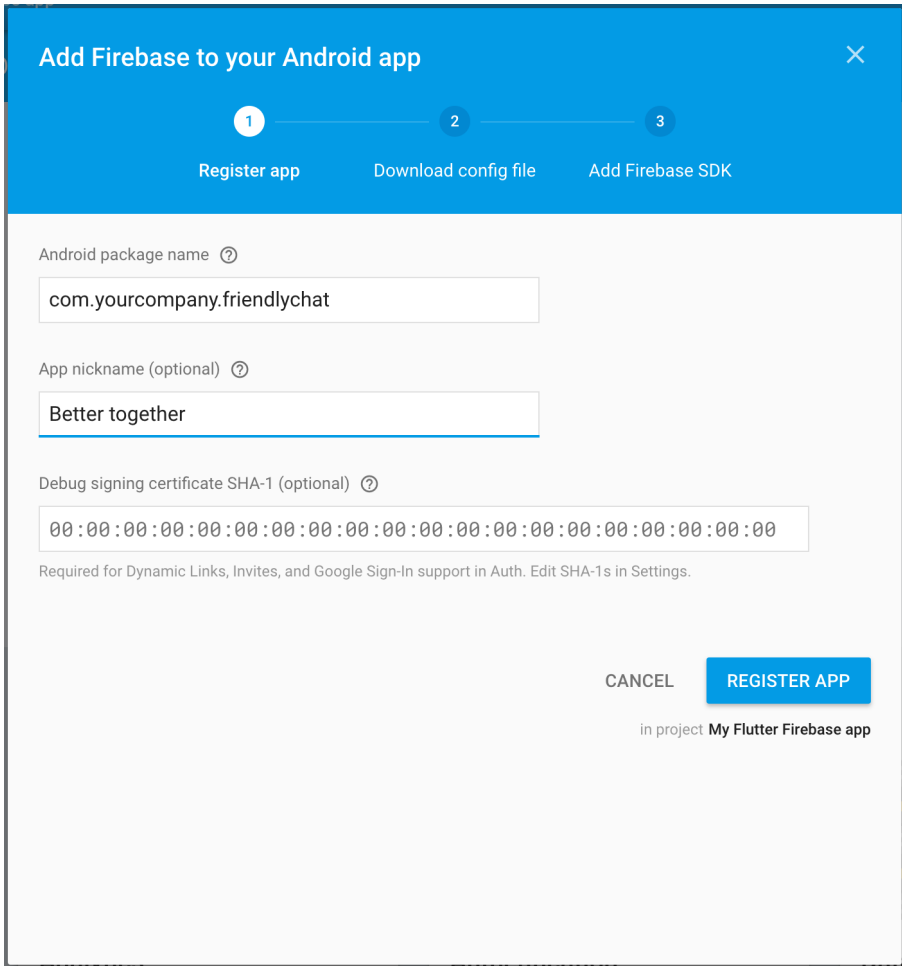
5. Firebase generates an `Info.plist` file that contains configuration details for iOS apps to connect to the Firebase service you created. Download this file to your development machine. (You do not need to complete the other steps in the wizard.)
6. Drag the generated `Info.plist` file to the `Runner` directory of your Runner project in Xcode, so the Google Sign-In framework can determine your client id. Your project's directory tree should look like this:



Configure Firebase for Android

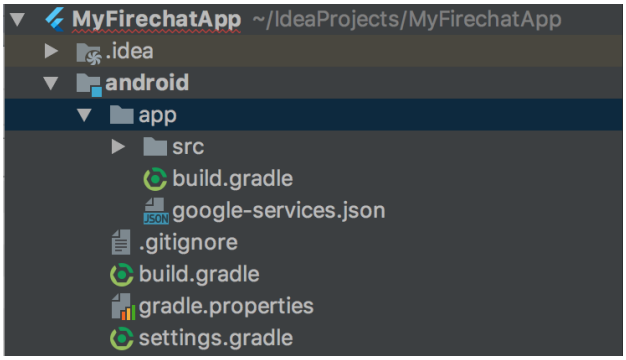
To add the Firebase configuration for Android to your Flutter project:

1. In your IntelliJ workspace, open the `AndroidManifest.xml` file in the `android/app/src/main` folder. Copy the string value for the `package` property (for this codelab, you'll use *com.yourcompany.friendlychat* but you can use a different string in your own app).
2. In the Firebase console, open your Firebase project, then click **Add Firebase to your Android app**. (If you've already configured Firebase for iOS, select **Add another app > Android** instead.)
3. In the **Android package name** field, specify the string value you copied from step 1.



4. In the **Debug signing certificate SHA-1** field, add the debug certificate fingerprint value. You can get this value by following the instructions in the Authenticating Your Client guide. Then click **Register App**.

   Firebase generates a `google-services.json` file that contains configuration details for Android apps to connect to the Firebase service you just created. Download this file to your development machine. (You do not need to complete the other steps in the wizard.)
5. Move the `google-services.json` file into the `android/app` folder. Your project's directory tree should look like this:



Integrate the FlutterFire package

Environment: web, kiosk

To add Firebase capabilities to your Flutter app, you'll use the open source FlutterFire wrapper. This package exposes a subset of the Firebase APIs to Flutter, for both Android and iOS. Support for external APIs is implemented via plugins.

Using plugins for Android development requires editing `build.gradle` files and adding the plugins to `pubspec.yaml`. For iOS development, only the `pubspec.yaml` changes are needed.

Important: For the following code to run on Android, you'll need the latest version of the Google Play services APK installed on your Android testing device.

Configure plugins for Android

To add the plugin configuration for Android to your project:

1. Add rules to your root-level build.gradle file, to include the `google-services` plugin.

android/build.gradle

```
buildscript {
    repositories {
        jcenter()
        maven {
            url "https://maven.google.com"
        }
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.3'
        classpath 'com.google.gms:google-services:3.1.0'                //new
```

```
    }
}
```

2. Add the apply plugin for Google Services **to the end** of your app-level `build.gradle` file (`/android/app/build.gradle`).

android/app/build.gradle

```
apply plugin: 'com.google.gms.google-services'                    //new
```

This should be the last line in the file.

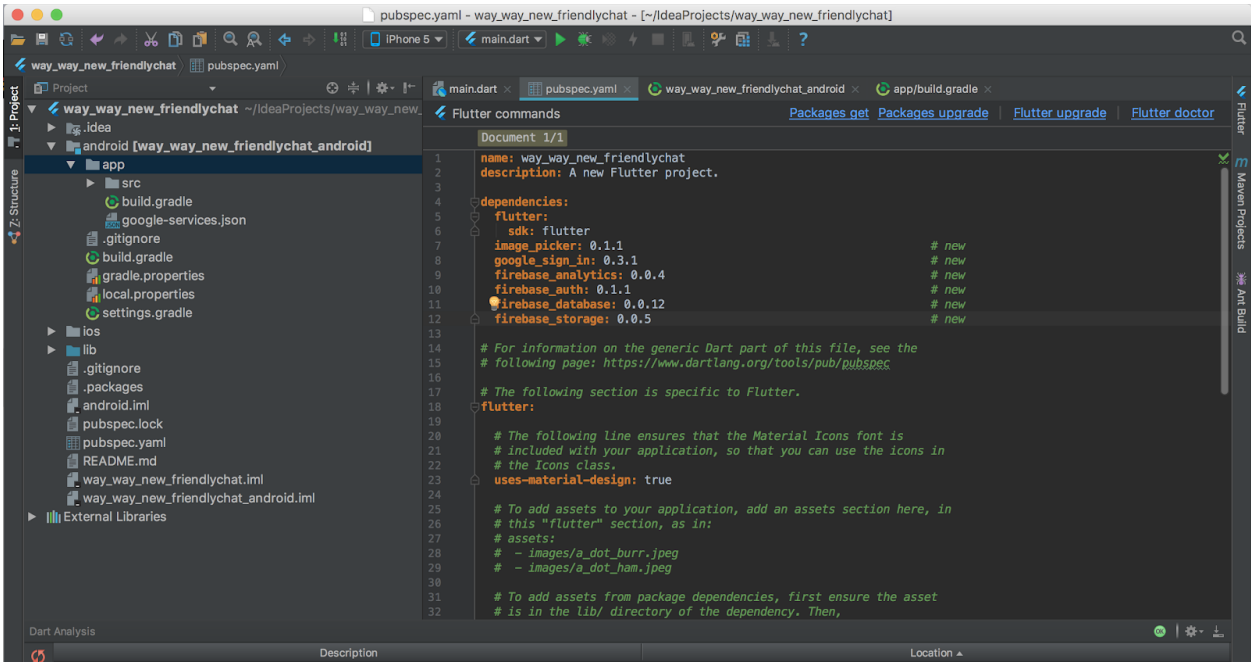
Add plugins to Friendlychat

1. Add the plugins you need for this codelab to the `pubspec.yaml` in your Flutter project file as follows:

pubspec.yaml

```
name: friendlychat
description: A new flutter project.

dependencies:
  flutter:
    sdk: flutter
  image_picker: 0.1.1                                             # new
  google_sign_in: 0.3.1                                           # new
  firebase_analytics: 0.0.4                                       # new
  firebase_auth: 0.2.0                                            # new
  firebase_database: 0.0.12                                       # new
  firebase_storage: 0.0.5                                         # new
```

After updating and saving this file, IntelliJ displays a prompt to reload `pubspec.yaml`.



2. Select **Packages get** to retrieve the additional packages and use them in your Flutter project.

Sign users into your app

Duration: 10:00

In this step, you'll use Google Sign-in to authenticate users of Friendlychat. Google Sign-In lets users securely sign in with their Google account—the same account they use with Gmail, Play, Photos and other Google services.

> **Note:** If you attempt to sign-in to an account with 2-factor auth enabled, and you're running on a simulator, sign-in may take you to an unexpected location, such as Apple Music.

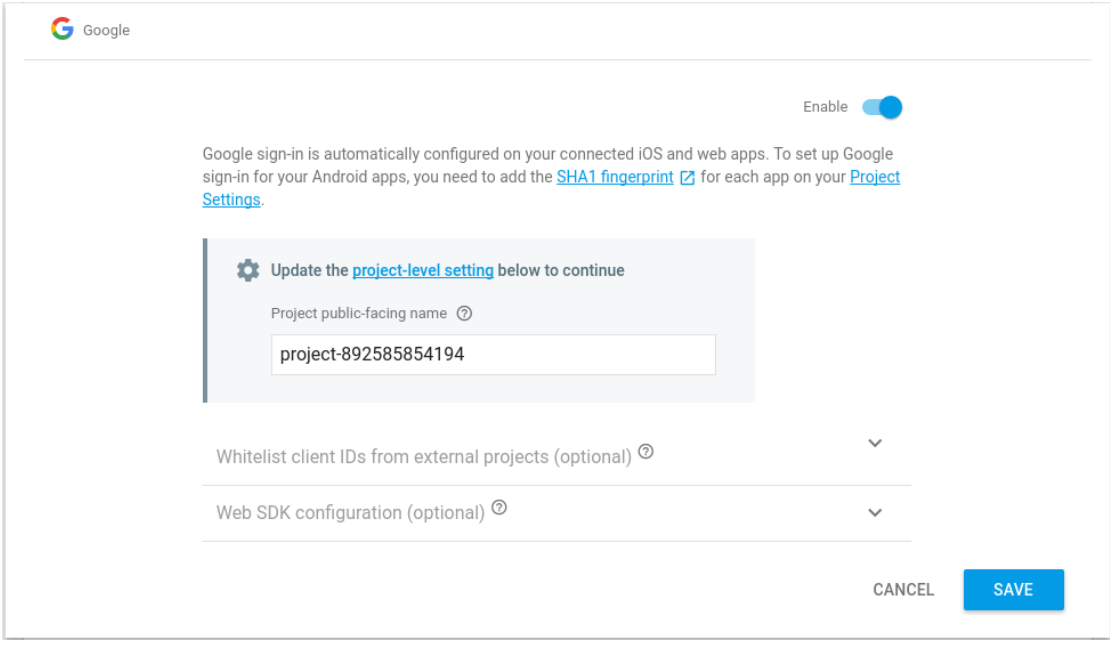
You'll also personalize the user experience, from profile and identity information associated with the user's Google account. When a user is signed in, we can personalize the chat message avatar with their profile photo. Let's see how this works in a Flutter app.

Enable Google Sign-In

The first step is to enable authentication for Google accounts.

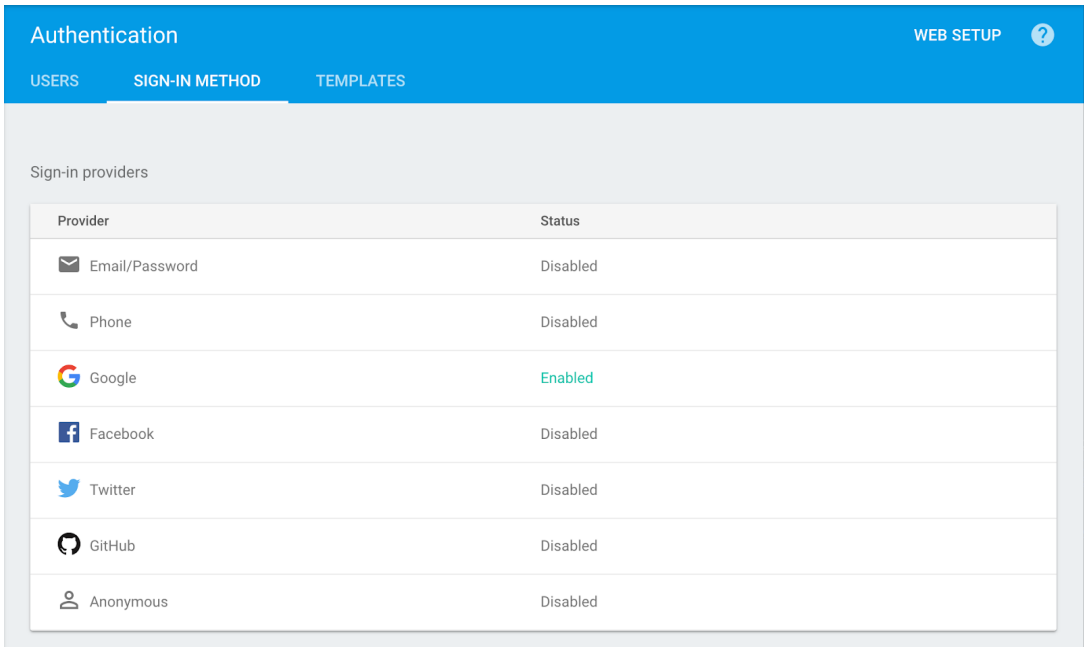
1. Open the Firebase console in your browser and select your project.

2. Navigate to **Authentication > Sign-in Method**
3. Select the **Google** provider and enable it by setting the toggle control to Enable.



4. Enter `com.yourcompany.friendlychat` (or the unique string you're using to identify this app) in the Project public-facing name field. This value matches the bundle identifier (iOS) or the package name (Android). Then click **Save**.

The result should look like this:



Configure Google Sign-In on iOS

To enable Google Sign-In:

1. Make sure the generated `Info.plist` file is in the `Runner` directory of your Runner project in Xcode, so the Google Sign-In framework can determine your client id.
2. Add the bundle ID and reverse URL of your client id to the main dictionary of your app's `Info.plist` file:

Info.plist

```
<!-- Add the following entries to the <dict> tag in Info.plist. -->

<key>CFBundleURLTypes</key>
  <array>
      <dict>
         <key>CFBundleTypeRole</key>
         <string>Editor</string>
         <key>CFBundleURLSchemes</key>
         <array>
         <!-- reverse url of your client id, for example: -->
<string>com.googleusercontent.apps.462578386393-kisbgopib3t6plf4dgv3s0n4ur3sv
jmo</string>
         <!-- bundle id, for example: -->
            <string>com.yourcompany.friendlychat</string>
         </array>
      </dict>
  </array>
```

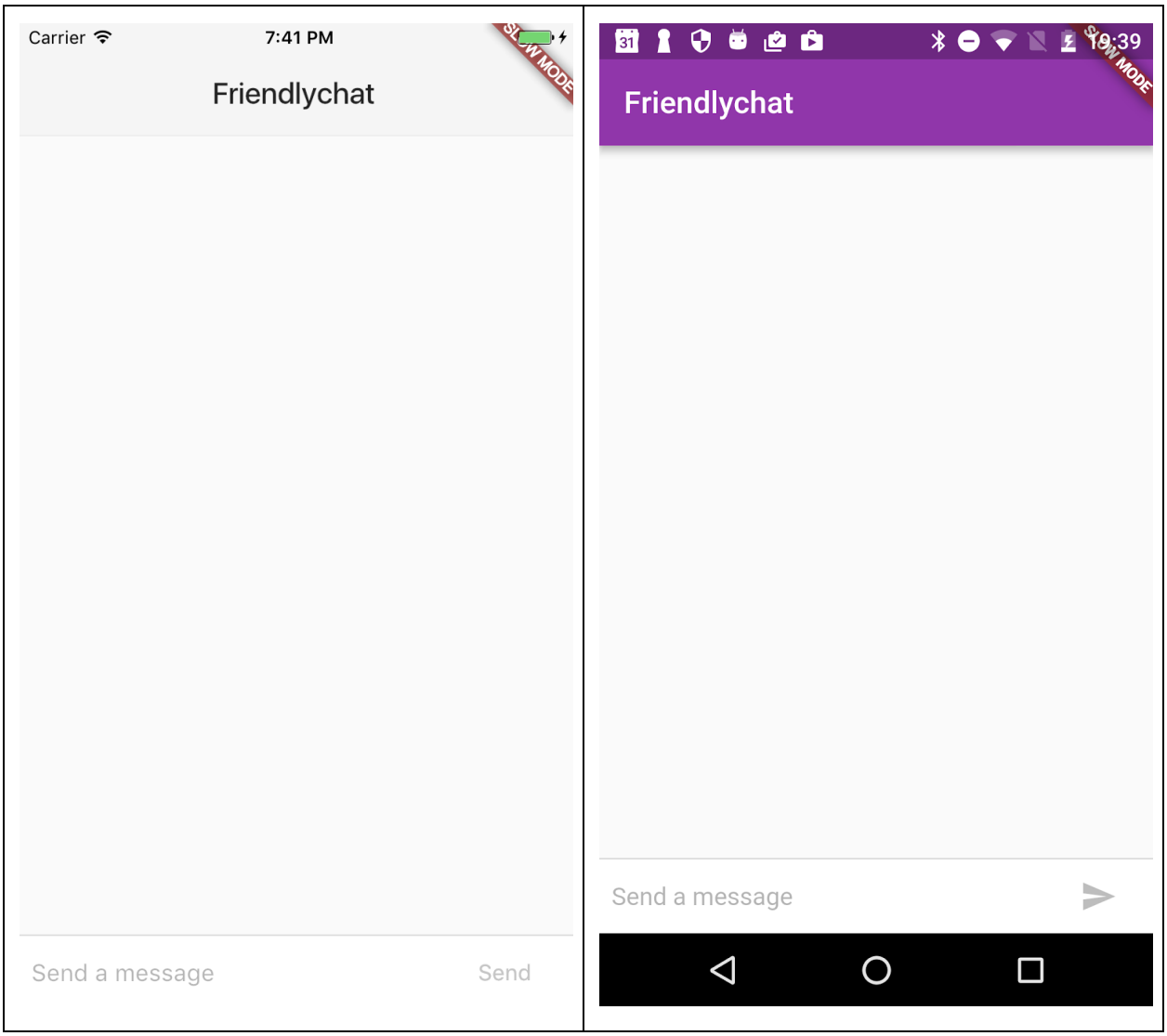
You can find the reverse URL of your client id in your `Info.plist` file. Copy the value of the `REVERSED_CLIENT_ID` string and paste to add it to `CFBundleURLSchemes`. This URL type handles the callback when a user logs into your app.

For the bundle ID, use either the literal string value or the `PRODUCT_BUNDLE_IDENTIFIER` variable.

Run Friendlychat

Hot reload the app. You should see a single screen that looks like this:

iOS	**Android**



The first time an app is started, it may take a minute or two to launch. On iOS, the extra time is required for initializing the cocoapods repo and, on Android, dependencies for maven are downloaded. After the next set of changes, the launch process will be faster.

Add Google Sign-In to Friendlychat

At the start of this codelab, your app is limited to a single user and device. When the user sends a message, the app labels it with the value of the `_name` variable and displays it on the same screen. The avatar is a simple colored circle marked with a letter.

After this codelab, multiple users will be able to share messages via a realtime database. To help scale the app you'll personalize the avatar to differentiate among users. Since you'll have

the sender's Google Sign-In credentials, you can reuse their profile photo. You'll add the database support in a later step.

To add support for Google Sign-In, you'll use the `google_sign_in` plugin. In your `main.dart` file, make sure to import the corresponding package. For details of what the plugin provides, view the Dart source code.

main.dart

```
// Add the following import statement to main.dart.

import 'package:google_sign_in/google_sign_in.dart';                    // new
```

Then, add a global variable named `googleSignIn`, as shown. Initialize it with a new `GoogleSignIn` instance, which we will use to call Google Sign-In APIs.

main.dart

```
// Add the following to main.dart.

final googleSignIn = new GoogleSignIn();                                // new
```

Sign in to Google

Before letting a user send a message, you need to give them a way to sign into their Google account. Sign-in can be a time-consuming process, so this section demonstrates how to use Dart's asynchrony support to wait for completion. The goal is to let users continue to interact with your app while waiting for the process to finish.

First, import the `dart:async` package so your Flutter app can `await` asynchronous methods, as shown.

main.dart

```
// Add the following import statement to main.dart.

import 'dart:async';                                                    // new
```

Now, define two private methods, one for signing in and another for sending the message. Add the `_ensureLoggedIn()` method to check the `currentUser` property of the `GoogleSignIn` instance.

main.dart

```
// Add the _ensureLoggedIn() method definition in ChatScreenState.

Future<Null> _ensureLoggedIn() async {
  GoogleSignInAccount user = googleSignIn.currentUser;
  if (user == null)
    user = await googleSignIn.signInSilently();
  if (user == null) {
    await googleSignIn.signIn();
  }
}
```

The previous snippet uses multiple `await` expressions to execute Google Sign-In methods in sequence. If the value of the `currentUser` property is null, your app will first execute `signInSilently()`, get the result and store it in the `user` variable. The `signInSilently` method attempts to sign in a previously authenticated user, without interaction. After this method finishes executing, if the value of `user` is still null, your app will start the sign-in process by executing the `signIn()` method.

After a user signs in, we can access the profile photo from the `GoogleSignIn` instance.

Coordinate sending messages

Submitting a message is now a two-step process (authentication and sending). We need to coordinate the work so that authentication happens first and if successful, then the user can send a message.

First, split the `_handleSubmitted` method from `ChatScreenState` into two separate methods. Instead of doing all the submitting work in `_handleSubmitted()`, you'll use it to coordinate the tasks performed by other methods, like verifying the user is logged in and sending the message.

Keep the `_textController.clear()` and `setState()`... `_isComposing` method calls in `_handleSubmitted()`. Add the `_ensureLoggedIn()`, and `_sendMessage()` calls, as shown.

main.dart

```
// Modify the _handleSubmitted() method definition in the ChatScreenState
class.

Future<Null> _handleSubmitted(String text) async {          //modified
  _textController.clear();
```

```
  setState(() {
    _isComposing = false;
  });
 await _ensureLoggedIn();                                    //new
 _sendMessage(text: text);                                   //new
}
```

Modify the signature of `_handleSubmitted()` to indicate that it's an async method that doesn't return anything. Add a call to `await _ensureLoggedIn()` to wait for the user to authenticate successfully before attempting to send a message.

Now let's define `_sendMessage()`. Add the remainder of `_handleSubmitted()` in a new private method as shown. You'll make `text` a named argument so that we can add more String arguments to `_sendMessage` later on.

main.dart

```
// Add the _sendMessage() method definition in the ChatScreenState class.

void _sendMessage({ String text }) {
  ChatMessage message = new ChatMessage(
    text: text,
    animationController: new AnimationController(
      duration: new Duration(milliseconds: 700),
      vsync: this,
    ),
  );
  setState(() {
    _messages.insert(0, message);
  });
  message.animationController.forward();
}
```

Personalize the avatar

To personalize the user's avatar, you'll replace the generic background color and first initial with the profile photo of the sender. You'll get the image using the `currentUser` property of the `GoogleSignIn` instance. The `google_sign_in.dart` plugin provides access to this property.

Set the background image by replacing the `Text` widget of the avatar with a `NetworkImage` widget. Make sure to replace the correct `Text` widget, the one with `CircleAvatar` as its parent.

main.dart

```
// Replace the Text widget in the build() method of the ChatMessage class.

class ChatMessage extends StatelessWidget {
  ChatMessage({this.text, this.animationController});
  final String text;
  final AnimationController animationController;

  Widget build(BuildContext context) {
    return new SizeTransition(
      sizeFactor: new CurvedAnimation(
          parent: animationController, curve: Curves.easeOut),
      axisAlignment: 0.0,
      child: new Container(
        margin: const EdgeInsets.symmetric(vertical: 10.0),
        child: new Row(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            new Container(
                margin: const EdgeInsets.only(right: 16.0),
                child: new CircleAvatar(
                    backgroundImage:                                    //
modified
                        new NetworkImage(googleSignIn.currentUser.photoUrl))//
modified
                ),
```

The `currentUser` property is the authentication object, and you use its `photoUrl` property to get the image for the new avatar.

Personalize the display name

Previously, we were hard coding the username. Now that we have integrated Google Sign-In, we can remove the `_name` global variable. Instead, obtain the username from the signed-in Google user:

main.dart

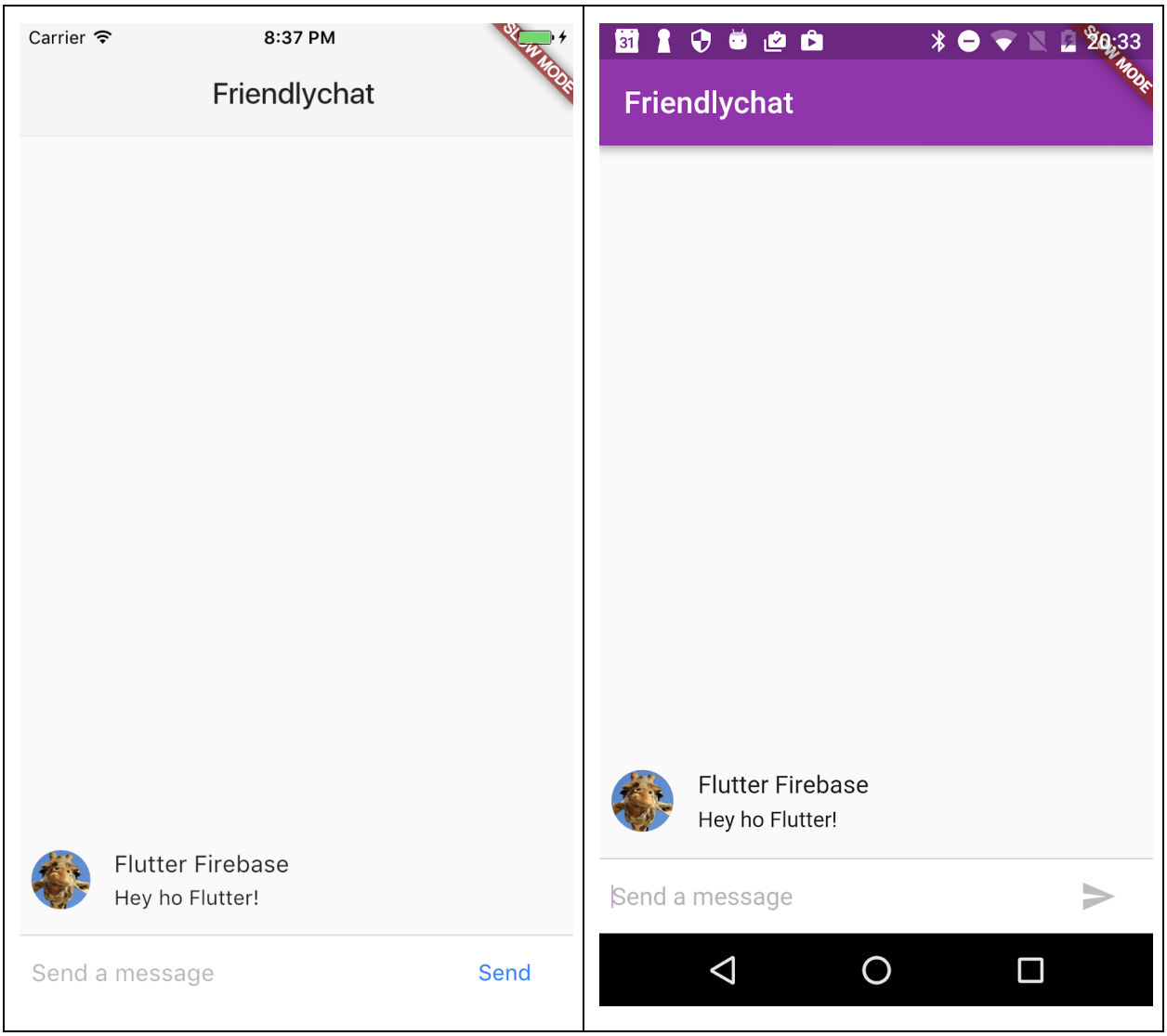
```
// Replace the _name variable in the ChatMessage class with displayName from
```

```
the GoogleSignIn instance.

child: new Container(
  margin: const EdgeInsets.symmetric(vertical: 10.0),
  child: new Row(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      new Container(
          margin: const EdgeInsets.only(right: 16.0),
          child: new CircleAvatar(
              backgroundImage:
                  new NetworkImage(googleSignIn.currentUser.photoUrl)
              )
          ),
      new Expanded(
        child: new Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            new Text(googleSignIn.currentUser.displayName,
//modified
                style: Theme.of(context).textTheme.subhead),
            new Container(
              margin: const EdgeInsets.only(top: 5.0),
              child: new Text(text),
            ),
          ],
        ),
      ),
    ],
  ),
),
```

Now when you send a message, the avatar and sender name match the profile information in your Google account.

Perform a full restart of the app and log in. You should see a single screen that looks like this:

iOS	**Android**



Track user activity

Duration: 5:00

Google Analytics for Firebase helps you understand how people use your Flutter app. In this step, you'll enable capturing predefined events. You will instrument the app to collect metrics for login events and sent messages. Once the data is captured, you'll view it in a dashboard through the Firebase console.

By default, any events you log to Google Analytics for Firebase will be aggregated, anonymized, and reported in the Firebase console within 24 hours. To see the events immediately, enable debug mode. For iOS, configure your Xcode scheme to pass the `FIRDebugEnabled` argument on launch. For Android, use `adb` to set the `debug.firebase.analytics.app` property.

To gather data on users with Google Analytics for Firebase, you'll use the `firebase_analytics` plugin. In your `main.dart` file, make sure to import the corresponding package. For details of what the plugin provides, view the Dart source code.

main.dart

```
// Add the following import statement to main.dart.

import 'package:firebase_analytics/firebase_analytics.dart';      // new
```

Now add a global variable named `analytics`, as shown in the next code snippet. Initialize it with a new `FirebaseAnalytics` instance, which you'll access through this variable.

main.dart

```
// Add the following to main.dart.

final analytics = new FirebaseAnalytics();      // new
```

Track login events

Next, you're ready to log some events in your app, in order to track and analyze them later. For this codelab, you'll track how many times users sign in with their Google account. You'll do this by logging sign-in events in the private `_ensureLoggedIn()` method in `ChatScreenState`, which gets invoked when a user sends a chat message.

main.dart

```
// Add the following to the _ensureLoggedIn() method in ChatScreenState.

Future<Null> _ensureLoggedIn() async {
  GoogleSignInAccount user = googleSignIn.currentUser;
  if (user == null)
    user = await googleSignIn.signInSilently();
  if (user == null) {
    await googleSignIn.signIn();
    analytics.logLogin();                                  //new
  }
}
```

The previous snippet calls the `logLogin()` method, which is defined by the Flutter Firebase Analytics plugin that you included in your app. This method, which takes no arguments, logs a Google Analytics for Firebase event named `login`.

Track sent messages

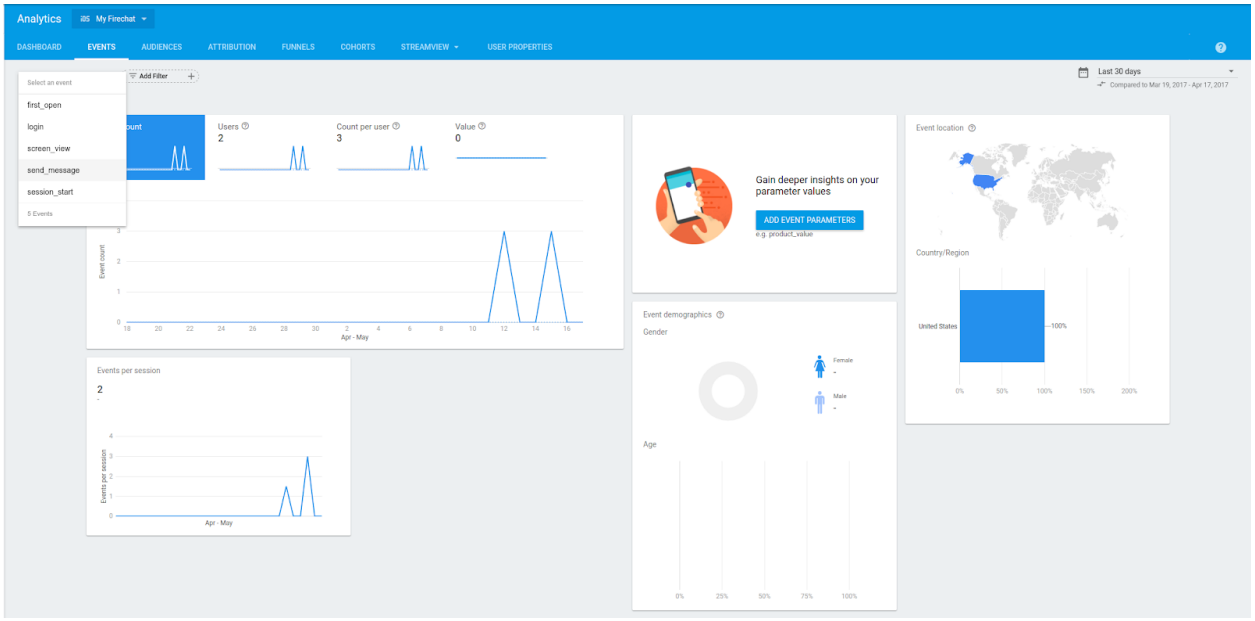
You might also want to track how many messages are sent by your app's users, as a measure of adoption and popularity. After a user installs your app, it's important to know whether they find it useful and engaging. You'll do this by logging sent message events in the private `_sendMessage()` method in `ChatScreenState`.

main.dart

```
// Add the following to the _sendMessage() method in ChatScreenState.

void _sendMessage({ String text }) {
  ChatMessage message = new ChatMessage(
    text: text,
    animationController: new AnimationController(
      duration: new Duration(milliseconds: 700),
      vsync: this,
      ),
   );
   setState(() {
     _messages.insert(0, message);
   });
   message.animationController.forward();
   analytics.logEvent(name: 'send_message');                    //new
}
```

The previous snippet calls the `logEvent()` method, which is defined by the Google Analytics for Firebase API. Access to this API is provided by the Flutter Firebase Analytics plugin, which you imported earlier. The `logEvent()` method, which takes no arguments, logs a Google Analytics for Firebase event named `send_message`.

Now when a user logs into the app or sends a message, an event is logged in the Firebase realtime database. To verify, click the Full Restart button () and send a message. In the Firebase console, select **Analytics > Events** to view the dashboard:



In the event dropdown list you should see the `login` and `send_message` events we just added, as well as the default `first_open`, `screen_view`, and `session_start` events.

Authenticate users

Duration: 10:00

Firebase Authentication lets you require your app's users to have a Google account. When a user signs in, Firebase Authentication verifies the credentials from Google Sign-In and returns a response to the app. Users who are signed in and authenticated can then connect to the Firebase Realtime Database and exhange chat messages with other users. You can apply authentication to make sure users see only messages they have access to, for example, and impose other types of restrictions on the contents of your database.

Configure Firebase Authentication

To authenticate your app's users using Firebase Authentication, you'll use the `firebase_auth` plugin. In your `main.dart` file, make sure to import the corresponding package. For details of what the plugin provides, view the Dart source code.

main.dart

```
// Add the following import statement to main.dart.

import 'package:firebase_auth/firebase_auth.dart';                    // new
```

Now add a global variable named `auth`, as shown in the next code snippet. Initialize it with a new `FirebaseAuth` instance, which you'll access through this variable.

main.dart

```
// Add the following to main.dart.

final auth = FirebaseAuth.instance;                                    // new
```

Connect Google Sign-In to Firebase

To require that only signed-in Google users can make changes to the database, add the authentication logic in the `_ensureLoggedIn()` method in `ChatScreenState`, as shown.

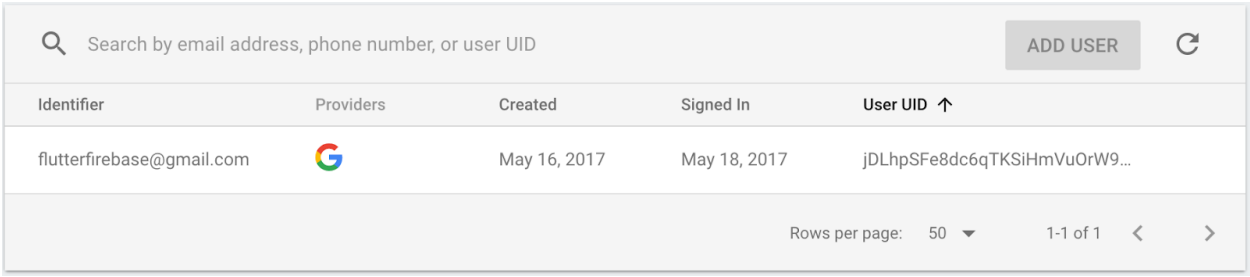
main.dart

```
// Add the following to the _ensureLoggedIn() method in ChatScreenState.

Future<Null> _ensureLoggedIn() async {
  GoogleSignInAccount user = googleSignIn.currentUser;
  if (user == null)
    user = await googleSignIn.signInSilently();
  if (user == null) {
    await googleSignIn.signIn();
    analytics.logLogin();
  }
  if (await auth.currentUser() == null) {                              //new
    GoogleSignInAuthentication credentials =                           //new
    await googleSignIn.currentUser.authentication;                     //new
    await auth.signInWithGoogle(                                       //new
      idToken: credentials.idToken,                                    //new
      accessToken: credentials.accessToken,                            //new
    );                                                                 //new
   }                                                                   //new
 }
```

The previous snippet checks whether `currentUser` is set to null. The authentication property is the user's credentials.

The `signInWithGoogle()` method takes an `idToken` and an `accessToken` as arguments. This method is provided by the Flutter Firebase Authentication plugin, which you imported earlier. It returns a new Firebase User object named `currentUser`.

If you hot reload the app () and login with a Google Account, you should now see your account information in the Authentication section of the Firebase console.



Now you're ready to require users to sign in with their Google account before they can send a message. After database rules are configured in the following step, Firebase Authentication will verify their credentials.

Enable data syncing

Duration: 15:00

In this step, you'll synchronize the app UI to a realtime database where chat messages are stored. You'll use the Firebase service to store and sync the chat message data to the cloud on a common shared realtime database. To learn more about how the Firebase Realtime Database works, see the product documentation.

To store and sync messages using Firebase Database, you'll use the `firebase_database` plugin. In your `main.dart` file, make sure to import the corresponding package. For details of what this plugin provides, view the Dart source code.

If you haven't already performed Step 6, "Sign users into your app", do that step now. You'll use Google Sign-In to authenticate users who can view the messages in Firebase Database.

You'll also use the `firebase_animated_list` plugin to enhance the list of chat messages. For details of what this plugin provides, view the Dart source code.

main.dart

```
// Add the following import statement to main.dart.

import 'package:firebase_database/firebase_database.dart';          //new
import 'package:firebase_database/ui/firebase_animated_list.dart'; //new
```

Configure security rules

1. In the Firebase console, select **Database** in the left navigation panel, then **Get Started** (for new Firebase projects).
2. On the Rules tab, change Firebase Database security rules for your project as follows:

```
{
  "rules": {
    "messages": {
      ".read": true,
      ".write": "auth != null && auth.provider == 'google'"
    }
  }
}
```

The above rules allow public read-only access to messages from the database, and Google authentication for writing messages to the database. At this point, users need to sign in before sending a message, and can view messages without being logged in.

Establish a connection to Firebase

To load the chat messages for display and submit the messages entered by the user, you must establish a connection with the Firebase realtime database. First, in your `ChatScreenState` widget, define a `DatabaseReference` variable called `reference`. Initialize this variable to get a reference to the `messages` path in your Firebase database.

main.dart

```
// Add the following code to ChatScreenState in main.dart.

final reference =
FirebaseDatabase.instance.reference().child('messages');          // new
```

Your app can now use this reference to read and write to a specific location in the database. The following code snippet shows how to modify the `_sendMessage()` method in the `ChatScreenState` class to add a new chat message to the database.

In the Friendlychat app messages were stored as an array of text values. Now that we are using a database, each message needs to be defined as a row with fields.

main.dart

```
// Modify the _sendMessage() method in ChatScreenState.

void _sendMessage({ String text }) {
  reference.push().set({                                          //new
    'text': text,                                                 //new
    'senderName': googleSignIn.currentUser.displayName,           //new
    'senderPhotoUrl': googleSignIn.currentUser.photoUrl,          //new
  });                                                             //new
  analytics.logEvent(name: 'send_message');
}
```

To write a new row for each chat message, the previous snippet calls the `push()` and `set()` methods, which are defined by the Firebase Database API. Access to this API is provided by the Flutter Firebase Database plugin, which you imported earlier. The `push()` method creates a new empty database row, and the `set()` method lets you populate it with the properties of the message (`text`, `senderName`, `senderPhotoUrl`).

Use an AnimatedList

When a message is sent or received, the original version of Friendlychat at the beginning of the codelab applies animation to slide it in vertically from the bottom of the list. The code for the UI takes a general, app-centric approach to animation, with `AnimationController` and `TickerProvider` objects managing the list of chat messages in a `ListView` widget.

In this codelab you'll use a specialized `AnimatedList` widget to achieve the same effect. This approach lets you integrate your app with the FirebaseDatabaseUI library. It enables you to perform the equivalent in your Flutter app of binding to a UITableView on iOS or a RecyclerView on Android. It also simplifies your code, with just an `animation` property required to animate the message.

Start by replacing the `ListView` widget in the `ChatScreenState` class with a new `FirebaseAnimatedList` widget, as shown.

main.dart

```
// Refactor the message list in the ChatScreenState class as follows.

body: new Column(children: <Widget>[
 new Flexible(
     child: new FirebaseAnimatedList(                             //new
         query: reference,                                        //new
         sort: (a, b) => b.key.compareTo(a.key),                  //new
```

```
          padding: new EdgeInsets.all(8.0),                          //new
          reverse: true,                                             //new
          itemBuilder: (_, DataSnapshot snapshot, Animation<double>
animation) { //new
            return new ChatMessage(                                  //new
              snapshot: snapshot,                                    //new
              animation: animation                                   //new
            );                                                       //new
          },                                                         //new
        ),                                                           //new
      ),
```

`FirebaseAnimatedList` is a custom widget provided by the Flutter Firebase Database plugin, which you imported earlier. The associated class is a wrapper around the `AnimatedList` class, enhancing it to interact with the Firebase Database.

The `query` argument to `FirebaseAnimatedList` specifies the database query that should appear in the list. In this case, you'll pass the database reference `reference`, which extends the `Query` class.

The `reverse` argument defines the beginning of the list as the bottom of the screen, near the text input. The `sort` argument specifies the order for displaying messages. To display messages at the beginning of the list (bottom of the screen) as they arrive, pass a function that compares the timestamp `key` of incoming messages.

Populate the list from the database

For the `itemBuilder` property, change the second parameter from `int index` (the position of the row being built) to a `DataSnapshot` object named `snapshot`. As the name implies, a snapshot represents the (read-only) contents of a row in the database. `FirebaseAnimatedList` will use this builder to populate list rows on demand as they scroll into view.

Finally, in the `ChatMessage` widget returned by the `build()` method of `ChatScreenState` change the `text` property to be a `snapshot`. The Flutter Firebase Database plugin defines a `snapshot` as having only a `key` and its `value`.

Until now, your app has been managing its own list of `ChatMessage` widgets and using it to populate a `ListView`. Now you'll use a `FirebaseAnimatedList`, which manages the animation controllers and automatically populates the list with the results of a Firebase Database query. You'll change the `ChatMessage` widget to construct its `CurvedAnimation` using the `Animation` object that the `FirebaseAnimatedList` passes to your app.

In the default constructor of the `ChatMessage` class, change the `AnimationController` to an `Animation` object.

main.dart

```
// Add the following to main.dart.

 class ChatMessage extends StatelessWidget {
   ChatMessage({this.snapshot, this.animation});                // modified
   final DataSnapshot snapshot;                                 // modified
   final Animation animation;                                   // modified
```

At the same time, let's retrofit the field for the message content from a text string to a `DataSnapshot`.

Remove obsolete animation code

Using `AnimatedList` syntax means modifying and removing a few lines of code from your app. Modify the CurvedAnimation object to use the new `animation` field rather than `animationController` as its parent, as shown.

main.dart

```
// Modify the following line of ChatMessage in main.dart.

class ChatMessage extends StatelessWidget {
  ChatMessage({this.snapshot, this.animation)};
  final DataSnapshot snapshot;
  final Animation animation;

  @override
  Widget build(BuildContext context) {
    return new SizeTransition(
      sizeFactor: new CurvedAnimation(
          parent: animation, curve: Curves.easeOut),              // modified
```

Remove the `TickerProviderStateMixin` widget and the List variable from the `ChatScreenState` class definition, as shown.

main.dart

```
// Modify the following line of ChatScreenState in main.dart.

class ChatScreenState extends State<ChatScreen> {               // modified
  final List<ChatMessage> _messages = <ChatMessage>[];          // remove
  final TextEditingController _textController = new
TextEditingController();
```

```
  bool _isComposing = false;
```

Also remove the `dispose()` method, which is no longer needed.

main.dart

```
// Remove the following method from ChatScreenState in main.dart.

void dispose() {                                                    // remove
  for (ChatMessage message in _messages)                            // remove
    message.animationController.dispose();                          // remove
  super.dispose();                                                  // remove
}                                                                   // remove
```

Modify the UI for messages

Now you can adapt the UI widgets that consume user profile information. The following widgets need to get this information from the Firebase Database API:

- `CircleAvatar`
- `Text` widget for the sender's name
- `Text` widget for the message content

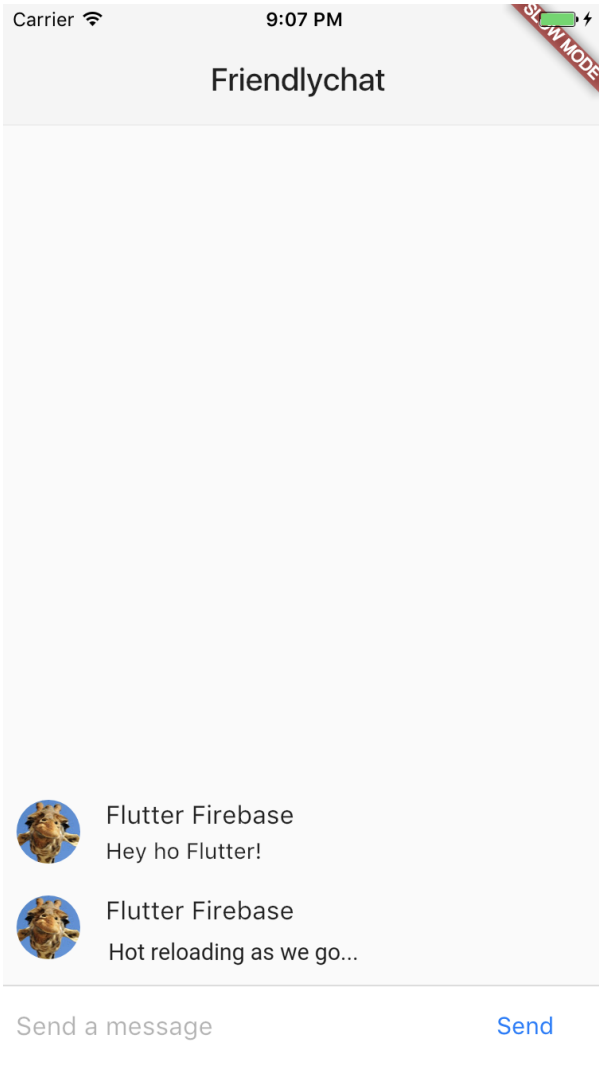
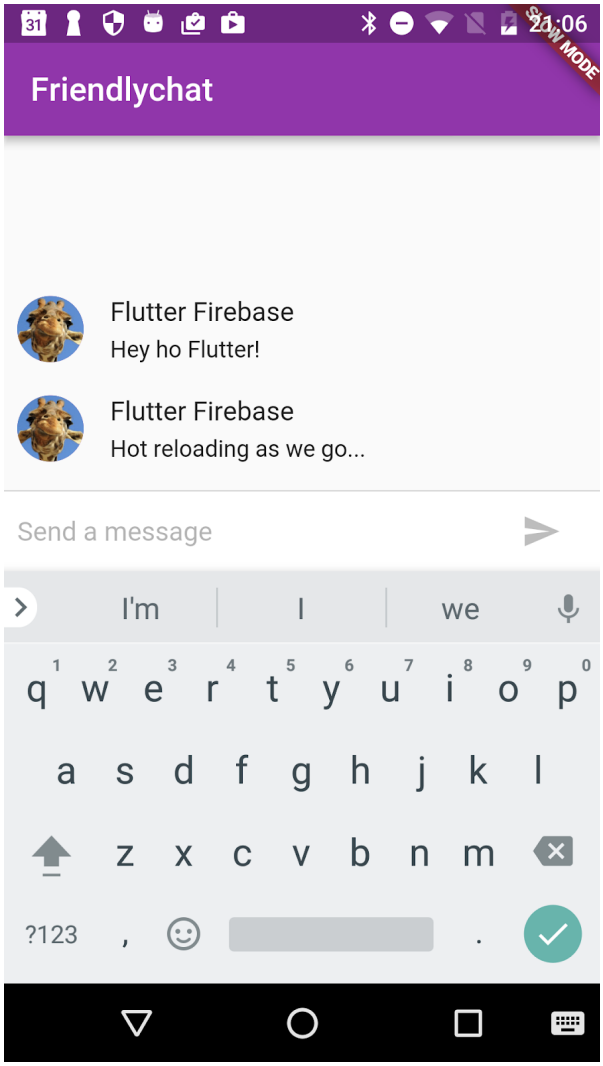
Instead of getting this information from a GoogleSignIn instance, you'll modify the widgets to get this information from the `value` field of a `DataSnapshot` object.

main.dart

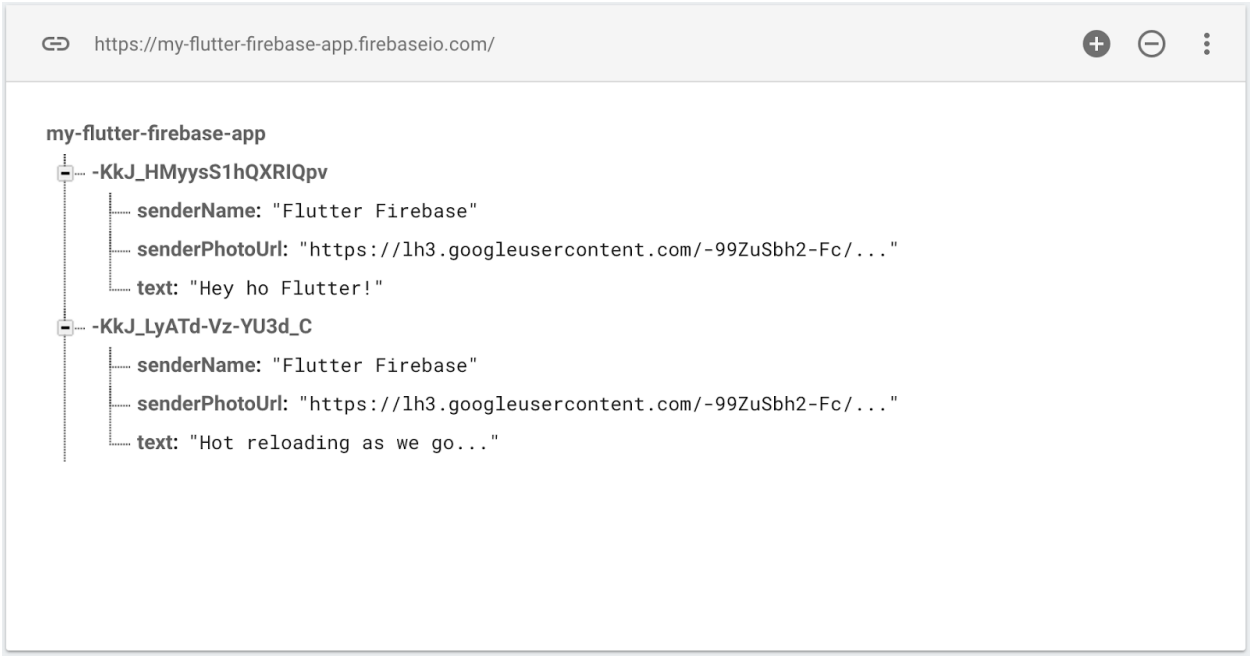
```
// Refactor the avatar and user's name in the ChatMessage class as follows.

child: new Row(
  crossAxisAlignment: CrossAxisAlignment.start,
  children: <Widget>[
    new Container(
      margin: const EdgeInsets.only(right: 16.0),
      child: new CircleAvatar(
          backgroundImage:
              new NetworkImage(snapshot.value['senderPhotoUrl'])),   //modified
    ),
    new Expanded(
      child: new Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          new Text(snapshot.value['senderName'],                     //modified
              style: Theme.of(context).textTheme.subhead),
          new Container(
            margin: const EdgeInsets.only(top: 5.0),
            child: new Text(snapshot.value['text']),                 //modified
          ),
        ],
```

```
        ),
      ),
    ],
  ),
```

If you hot reload the app (), you should see a single screen that looks like this.

iOS	Android
	

If you have two devices connected to the development machine, you should be able to send a message and see it arrive on the other device, via the Firebase Database. With a single device, you can view the messages under Database in the Firebase console:



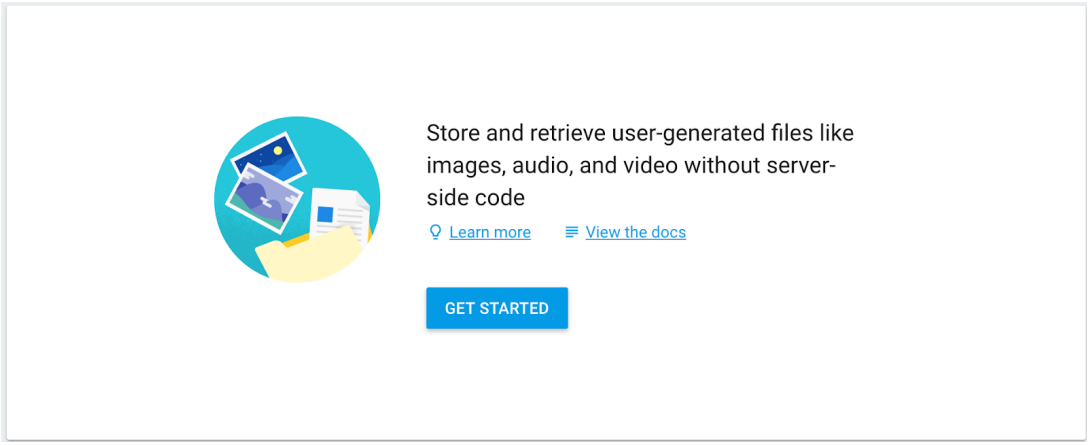
Initializing the `State` object requires restarting the app. For details on the types of changes that can be hot reloaded, see Hot Reloading Flutter Apps.

Add image support

Duration: 15:00

In this step, you'll enable users to send images in chat messages, retrieve image files from the device, and store text and image data in a Google Cloud Storage bucket. As a result of using Cloud Storage for Firebase, your app will become more robust and scalable. It will be able to handle network interruptions during uploads and downloads, store data securely, and have consistent performance as your user base expands.

To activate Google Cloud Storage for your new Firebase project, select **Storage** in the left navigation panel in the Firebase console, then **Get Started.**



You need a couple of Flutter plugins for the new image workflows: one for the Firebase SDKs, and a custom plugin for selecting images in a Flutter app. Several Dart libraries are also needed for file handling.

To upload data like text and photos from a mobile device to the cloud, you'll use the `firebase_storage` plugin. In your `main.dart` file, make sure to import the corresponding package. For details of what the plugin provides, view the Dart source code.

main.dart

```
// Add the following import statement to main.dart.

import 'package:firebase_storage/firebase_storage.dart';          // new
```

Access data on the device

To access data stored on a mobile device and use it in your Flutter app, you'll need Flutter's platform services API and the `image_picker` plugin. In your `main.dart` file, import the `image_picker` package. For details of what the plugin provides, view the Dart source code (note that it imports `services.dart` for you).

Also import the `dart:math` and `dart:io` libraries, for generating random file names and performing file operations on the device.

main.dart

```
// Add the following import statements to main.dart.

import 'package:image_picker/image_picker.dart';     // new

import 'dart:math';                                  // new
import 'dart:io';                                    // new
```

Add image support to the UI

At this point, your app lets users send and receive text messages. Now you'll add a button to the UI for composing messages, so users can access the device's camera from your app. Then you'll give the UI the ability to handle binary data.

 Enable camera and photo library access on iOS

If you're testing on a physical iOS device, add settings to give the image picker plugin access to the camera and images stored on the device.

To enable access, add the following entries to the main dictionary of the app's `Info.plist` file:

Info.plist

```
// Add the following entries to Info.plist.

 <dict>
        <key>NSCameraUsageDescription</key>                 <!-- new -->
        <string>Share images with other chat users</string>  <!-- new -->
        <key>NSPhotoLibraryUsageDescription</key>           <!-- new -->
        <string>Share images with other chat users</string>  <!-- new -->
 ...
 </dict>
```

The `<string>` elements can contain any text value. You can also add these settings in Xcode, with new rows for the `Privacy - Camera Usage Description` and `Privacy - Photo Library Usage Description` properties.

 Enable camera and media content access on Android

If you're testing on a physical Android device, add settings to give the image picker plugin access to the camera and images stored on the device.

To enable access, add the following entries to the app's `AndroidManifest.xml` file:

AndroidManifest.xml

```
// Add the following entries to AndroidManifest.xml.

 <manifest>
```

```
 ...
    <uses-permission android:name="android.permission.CAMERA"/>
<!-- new -->
    <uses-permission
android:name="android.permission.MEDIA_CONTENT_CONTROL"/> <!-- new -->
 ...
 </manifest>
```

Add a camera button

Users need a way to access images stored on the device. You'll create a button for this purpose next to the UI for composing chat messages. This part of the app UI is defined by the private method `_buildTextComposer()` in `ChatScreenState`.

Add an `IconButton` widget for accessing the camera and stored images to the `_buildTextComposer` method. The existing `Row` widget that wraps the input field and send button should be the parent. Wrap your `IconButton` widget inside a new `Container` widget; this lets you customize the margin spacing of the button so that it visually fits better next to your input field.

main.dart

```
// Add the following to the _buildTextComposer() method in ChatScreenState.

Widget _buildTextComposer() {
  return new IconTheme(
    data: new IconThemeData(color: Theme.of(context).accentColor),
    child: new Container(
        margin: const EdgeInsets.symmetric(horizontal: 8.0),
        child: new Row(children: <Widget>[
          new Container(                                          //new
            margin: new EdgeInsets.symmetric(horizontal: 4.0),    //new
            child: new IconButton(                                //new
                icon: new Icon(Icons.photo_camera),               //new
                onPressed: () {}                                  //new
            ),                                                    //new
          ),                                                      //new
         new Flexible(
           child: new TextField(
```

In the `icon` property, use the `Icons.photo_camera` constant to create a new `Icon` instance This constant lets your widget use the following icon provided by the Flutter material icons library:

Enable sending images

Now you'll modify the `onPressed` callback for the camera button. You need to get a reference to the Google Cloud Storage bucket and start uploading an image right after a user selects it.

For the button's `onPressed` property,you'll invoke an `async` function and combine it with several `await` expressions to perform the image-related tasks in a specific order. Read more about asynchronous programming support in Dart.

main.dart

```
// Add the following to the _buildTextComposer() method in ChatScreenState.

child: new IconButton(
  icon: new Icon(Icons.photo_camera),
  onPressed: () async {                                          // modified
    await _ensureLoggedIn();                                     // new
    File imageFile = await ImagePicker.pickImage();              // new
  },                                                             // new
 ),
```

Before allowing a user to select, upload, and send an image, the previous code snippet makes sure they are logged in by executing the `_ensureLoggedIn()` private method.

Then it waits for the user to select an image and gets a new File object named `imageFile` from the `pickImage()` method. This method is provided by the Flutter Image Picker plugin, which you imported earlier. It takes no arguments and returns a String value named `path` for the URL of the image file.

Next, modify `_sendMessage()` to add the `imageUrl` parameter, as shown.

main.dart

```
// Add the following to the _sendMessage() method in main.dart.

void _sendMessage({String text, String imageUrl }){          //modified
  reference.push().set({
    'text': text,
    'imageUrl': imageUrl,                                                //new
    'senderName': googleSignIn.currentUser.displayName,
    'senderPhotoUrl': googleSignIn.currentUser.photoUrl,
  });
```

```
  analytics.logEvent(name: 'send_message');
}
```

Create an instance variable for uploading the file to Google Cloud Storage. Initialize it to get a reference to the File object and pass it to the `put()` method. Give each file a unique name using `image_` as a prefix followed by a random integer, as shown.

main.dart

```
// Add the following to the _buildTextComposer() method in ChatScreenState.

child: new IconButton(
  icon: new Icon(Icons.photo_camera),
  onPressed: () async {
    await _ensureLoggedIn();
    File imageFile = await ImagePicker.pickImage();
    int random = new Random().nextInt(100000);                          //new
    StorageReference ref =                                              //new
    FirebaseStorage.instance.ref().child("image_$random.jpg");          //new
    StorageUploadTask uploadTask = ref.put(imageFile);                  //new
    Uri downloadUrl = (await uploadTask.future).downloadUrl;
//new
```

The `put()` method is defined by the Cloud Storage for Firebase API. It takes a File object as an argument and uploads it to a Google Cloud Storage bucket. The Flutter Firebase Storage plugin, which you imported earlier, provides access to this API and also defines the `StorageUploadTask` class.

After upload completes successfully, you can get the `downloadURL` for the image. Now call `_sendMessage()` and pass the `downloadURL` to send the image.

main.dart

```
// Add the following to the _buildTextComposer() method in main.dart.

child: new IconButton(
  icon: new Icon(Icons.photo_camera),
  onPressed: () async {
    await _ensureLoggedIn();
    File imageFile = await ImagePicker.pickImage();
    int random = new Random().nextInt(100000);
    StorageReference ref =
        FirebaseStorage.instance.ref().child("image_$random.jpg");
    StorageUploadTask uploadTask = ref.put(imageFile);
    Uri downloadUrl = (await uploadTask.future).downloadUrl;
    _sendMessage(imageUrl: downloadUrl.toString());                     //
```

```
new
  },
 ),
```

This URL is how the app UI lets a user who is receiving the chat message download the image from Google Cloud Storage to their device. It also lets the sender view their sent image in the chat conversation.

Display sent and received images

Each chat message, sent or received, needs to be able to display images as well as text. Next you'll enhance the `ChatMessage` class to detect when a user is sending an image and get the image using its URL.

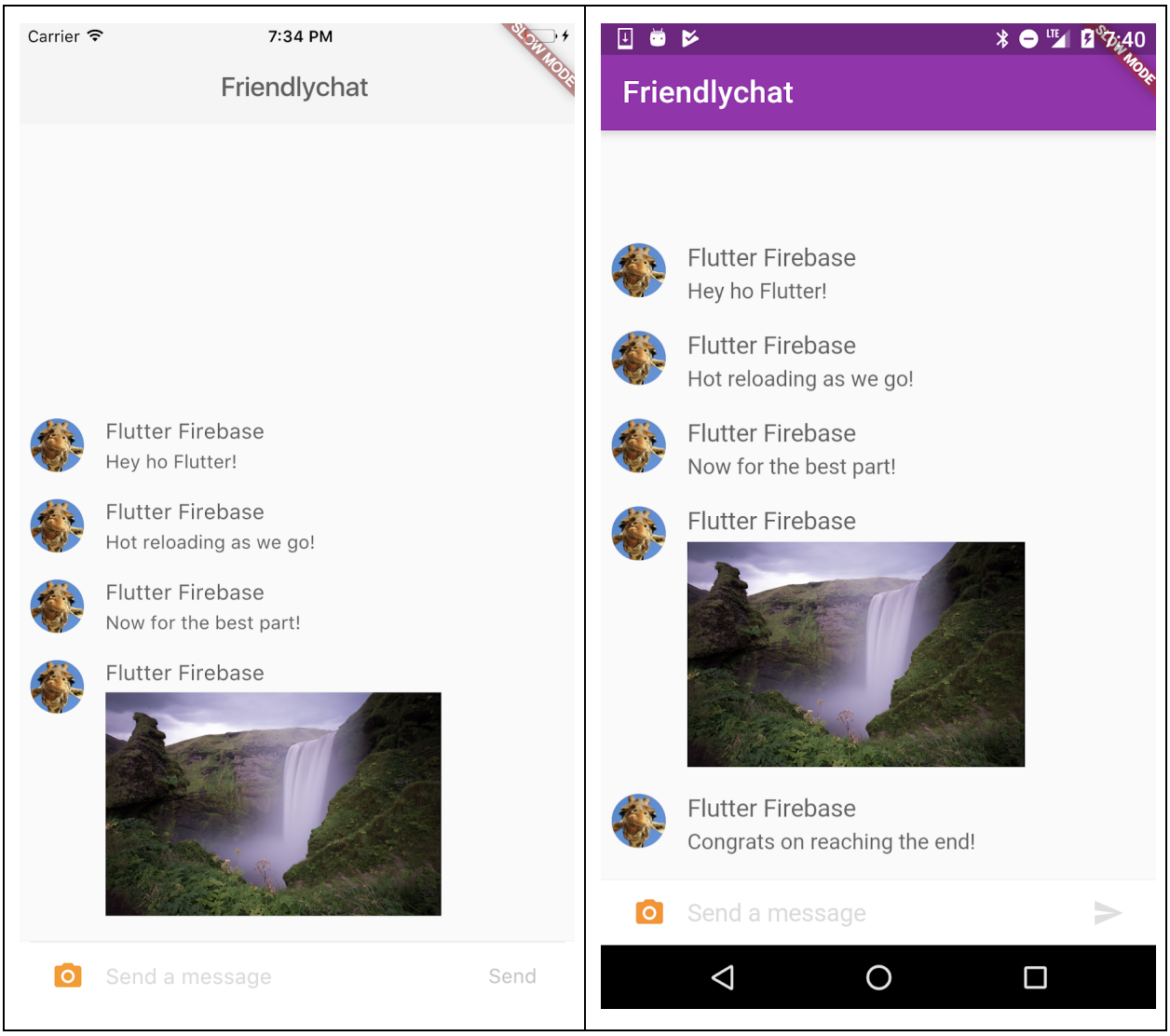
main.dart

```
// Add the following to the ChatMessage class in main.dart.

new Expanded(
  child: new Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      new Text(
          snapshot.value['senderName'],
          style: Theme.of(context).textTheme.subhead
      ),
      new Container(
        margin: const EdgeInsets.only(top: 5.0),
        child: snapshot.value['imageUrl'] != null ?
//modified
        new Image.network(                                          //new
           snapshot.value['imageUrl'],                             //new
           width: 250.0,                                            //new
        ) :                                                         //new
        new Text(snapshot.value['text']),
```

In the previous code snippet, if the value field of a database row is an `imageUrl` then your app creates an `Image` widget and populates it with the contents of the image. The width is set to a specific number of logical pixels, for consistency.

If you restart the app () , you should see a single screen that looks like this:

iOS	Android



If the message does not contain an image, your app creates a `Text` widget, as it did before you completed this step.

We changed the `Info.plist` and `AndroidManifest.xml` configuration files in this final step, which requires restarting the app rather than hot reloading.

Next steps

Duration: 2:00
Environment: web, kiosk

Congratulations!

You now know the basics of integrating Flutter apps with Firebase.

What we've covered

- ✓ How to implement Google-Sign in for a Flutter app.
- ✓ How to add Firebase Authentication and Google Analytics for your app.
- ✓ How to connect your app to Firebase Database for data syncing.
- ✓ How to use Flutter's platform services API to access data on the device.
- ✓ How to upload files to Cloud Storage for Firebase.

Ready to share your new app with friends? Check out how you can get the platform-specific binaries for your Flutter app (an IPA file for iOS and an APK file for Android). And see below for resources to continue learning about Flutter and Firebase outside of this codelab.

Additional resources

Environment: web, kiosk

For more information on Firebase, see:

- Main Firebase site (https://firebase.google.com/)
- Firebase documentation (https://firebase.google.com/docs/)
    - Firebase Realtime Database
    - Firebase Authentication
    - Google Analytics for Firebase
- Introductory Firebase codelab (for iOS/Swift) (for Android)

You may find these developer resources useful as you continue working with the Flutter framework on your own:

- Building Beautiful UIs with Flutter codelab.
- Flutter.io: The documentation site of the Flutter project.
- The Flutter API reference documentation.
- Additional Flutter sample apps, with source code.

Get the latest news about Flutter

Here are some ways to get the latest news about Flutter:

- Follow @flutterio on Twitter.
- Star us on GitHub (https://github.com/flutter/flutter).
- Join our mailing list (https://groups.google.com/forum/#!forum/flutter-dev).

Tell us how we’re doing

How likely are you to recommend trying Flutter to your friends and colleagues? (from 1=”Very unlikely” to 5=”Very likely”)

- 1
- 2
- 3
- 4
- 5