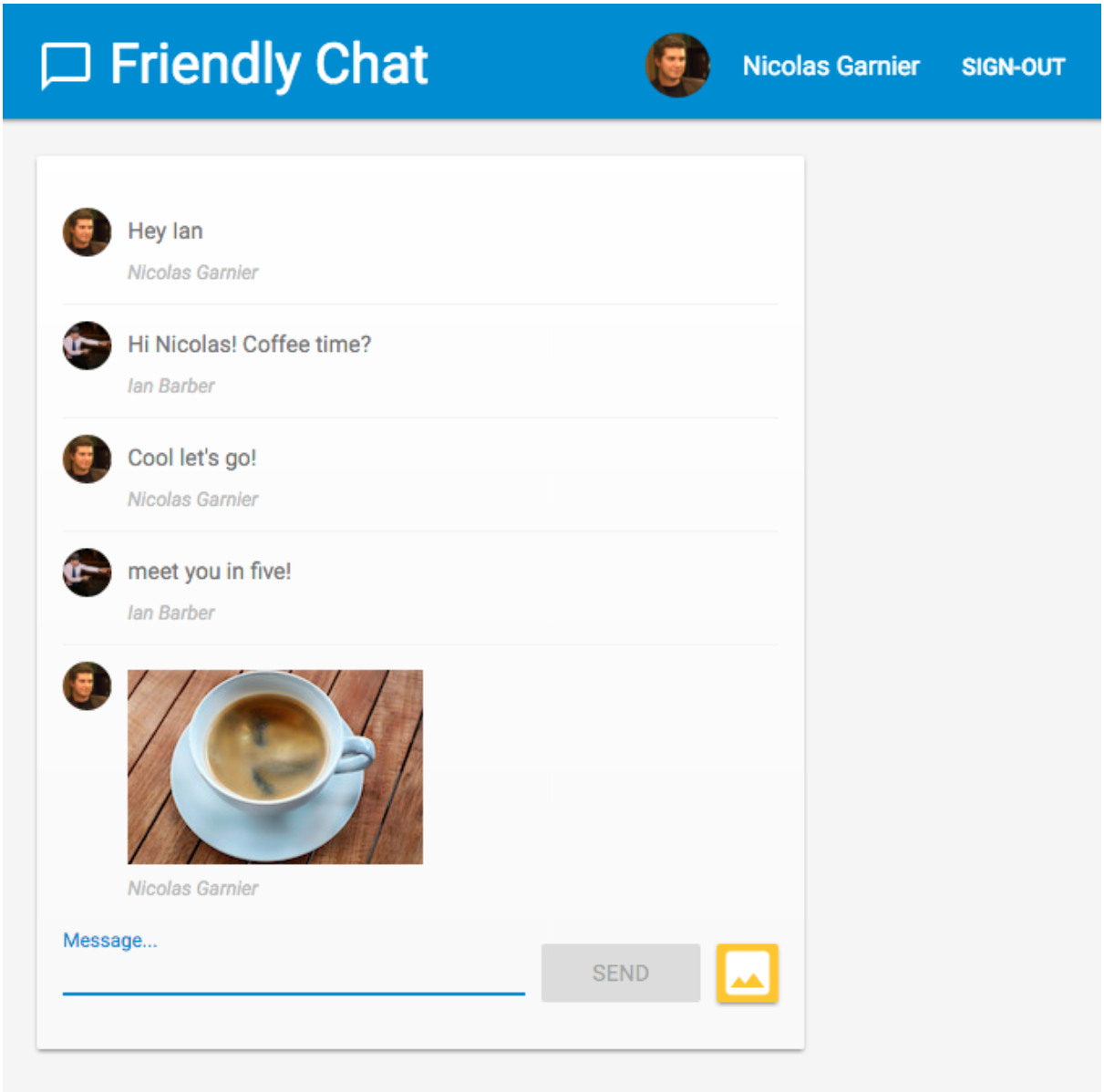
Firebase Web

GSP065

Google Cloud Self-Paced Labs

Overview

In this lab you will learn how to use Firebase to easily create web applications by implementing and deploying a chat client using Firebase products and services.



What you'll learn

In this lab, you will learn how to:

- Sync data using the Cloud Firestore and Cloud Storage for Firebase.
- Authenticate your users using Firebase Auth.
- Deploy your web app on Firebase Hosting.
- Send notifications with Firebase Cloud Messaging.

Task 1. Upgrade node version

1. In Cloud Shell, check which version of **Node** you have:

```
node --version
```

2. If the output of the previous command is `v16.1.0` or lower, run the following commands to upgrade **Node**:

```
nvm install v16.15.0

nvm use v16.15.0
```

Task 2. Get the sample code

- In Cloud Shell command line, clone the GitHub repository:

```
git clone https://github.com/firebase/friendlychat-web
```

The `friendlychat-web` repository contains sample projects for multiple platforms. This lab uses only two repositories:

- **web-start**: The starting code that you'll build upon in this lab.
- **web**: The complete code for the finished sample app.

Note: If you want to just run the finished app, you still have to create a project in the Firebase console. See the **Create a Firebase project and set up your app** section for instructions.

Task 3. View the starter application

Throughout this lab you will be modifying files in the `friendlychat-web` directory. You can use text editors that come pre-installed in Cloud Shell (like `nano` or `vim`), but this lab will use the Cloud Shell code editor.

1. To view `friendlychat-web`, click the **Open Editor** icon.

2. This opens a new browser window with the Cloud Shell tab.
3. Navigate to `friendlychat-web/web-start` in the left pane to view the application files and code in the right pane. In this lab, navigation is relative to `friendlychat-web/web-start`. The `friendlychat-web/web-start` directory contains the starting code for this lab, which consists of a fully functional Chat Web App.

Task 4. Set up your Firebase project

Now that your files are opened up in the Cloud Shell code editor, get Firebase set up. The application you build in this lab uses the whole set of Firebase products available on the web:

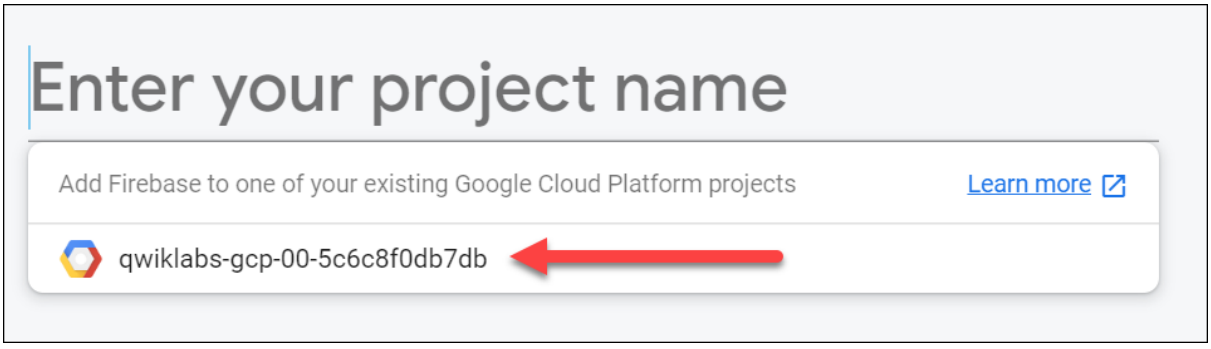
- **Firebase Authentication** to easily let your users sign-in your app.
- **Cloud Firestore** to save structured data on the cloud and get instant notification when data changes.
- **Cloud Storage for Firebase** to save files in the cloud.
- **Firebase Hosting** to host and serve your assets.
- **Firebase Cloud Messaging** to send push notifications and display browser popup notifications.

Task 5. Enable Firebase on your project

1. Open the Firebase console in a new tab.

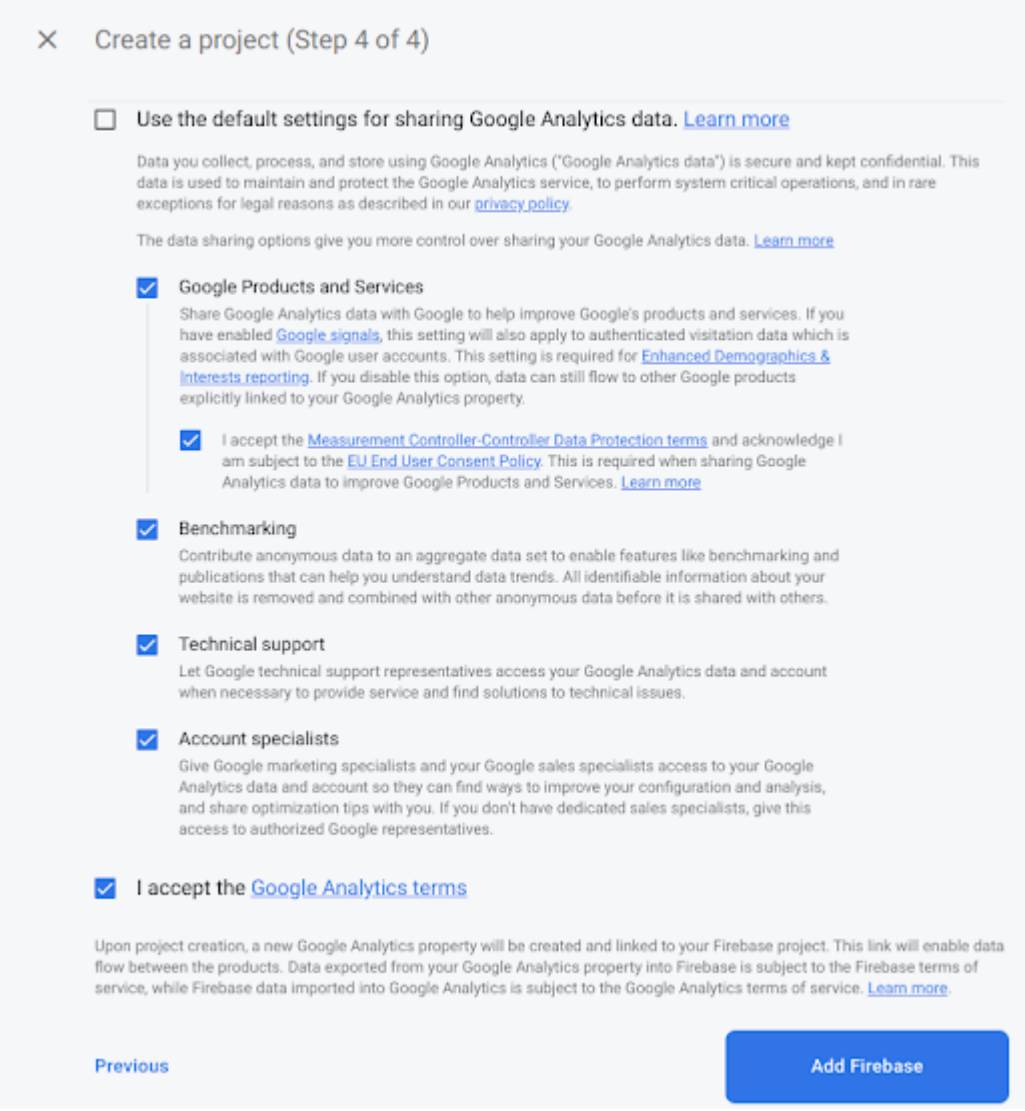
Note: You may need to sign in again using your lab username and password.

2. In the **Firebase console**, click **Add project**.
3. In the **Enter your project name** dialog, select your project name.



4. Check **I accept the Firebase terms**.
5. Check **I confirm that I will use Firebase exclusively for purposes relating to my trade, business, craft, or profession**. Click **Continue**.
6. In the **A few things to remember when adding Firebase to a Google Cloud project** dialog, click **Continue**.
7. In the **Google Analytics for your Firebase project** dialog, click **Continue**.
8. In the **Configure Google Analytics** dialog:

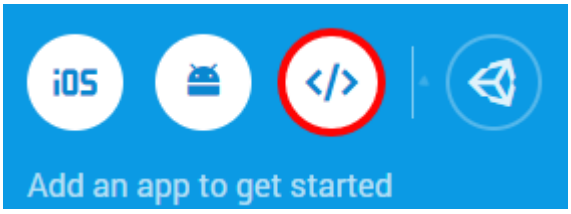
- Uncheck **Use the default settings for sharing Google Analytics data**.
- Check **Google Products and Services**.
- Check to accept the **Measurement Controller-Controller Data Protection** terms and acknowledge you are subject to the **EU End User Consent Policy**.
- Check **I accept the Google Analytics terms**.
- Click **Add Firebase**.



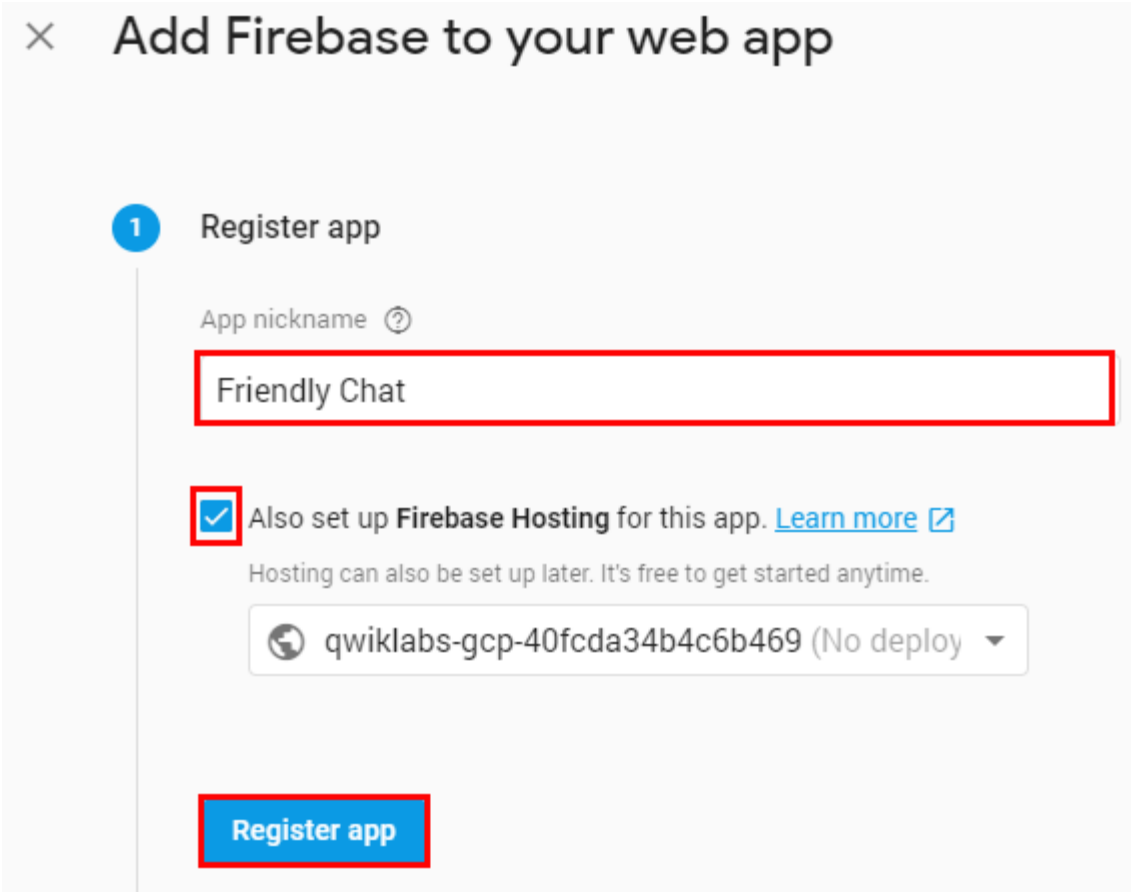
9. When your new project is ready, click **Continue**.

Task 6. Add a Firebase web app

1. Click the web icon to create a new Firebase web app.



2. Register the app with the nickname **Friendly Chat**, then check the box next to "Also set up **Firebase Hosting** for this app". Click **Register app**.

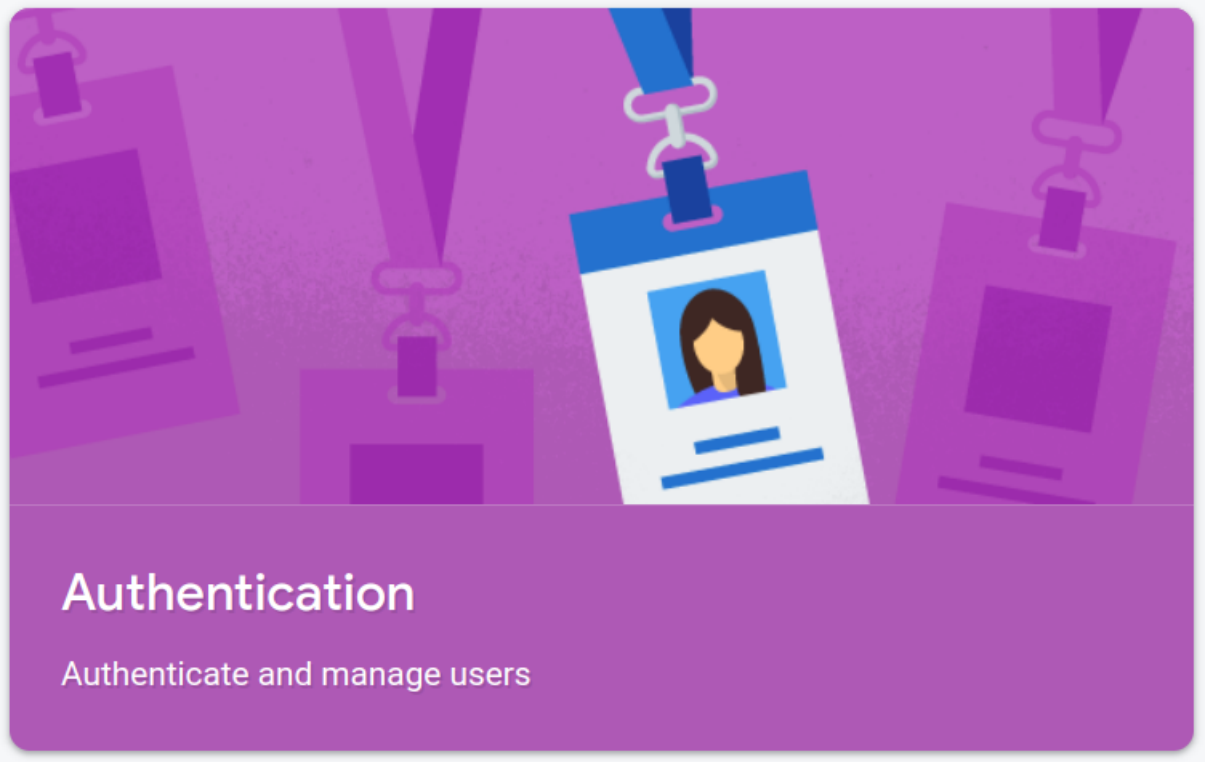


3. Click through the remaining steps. You don't need to follow the instructions now, these will be covered in later steps.

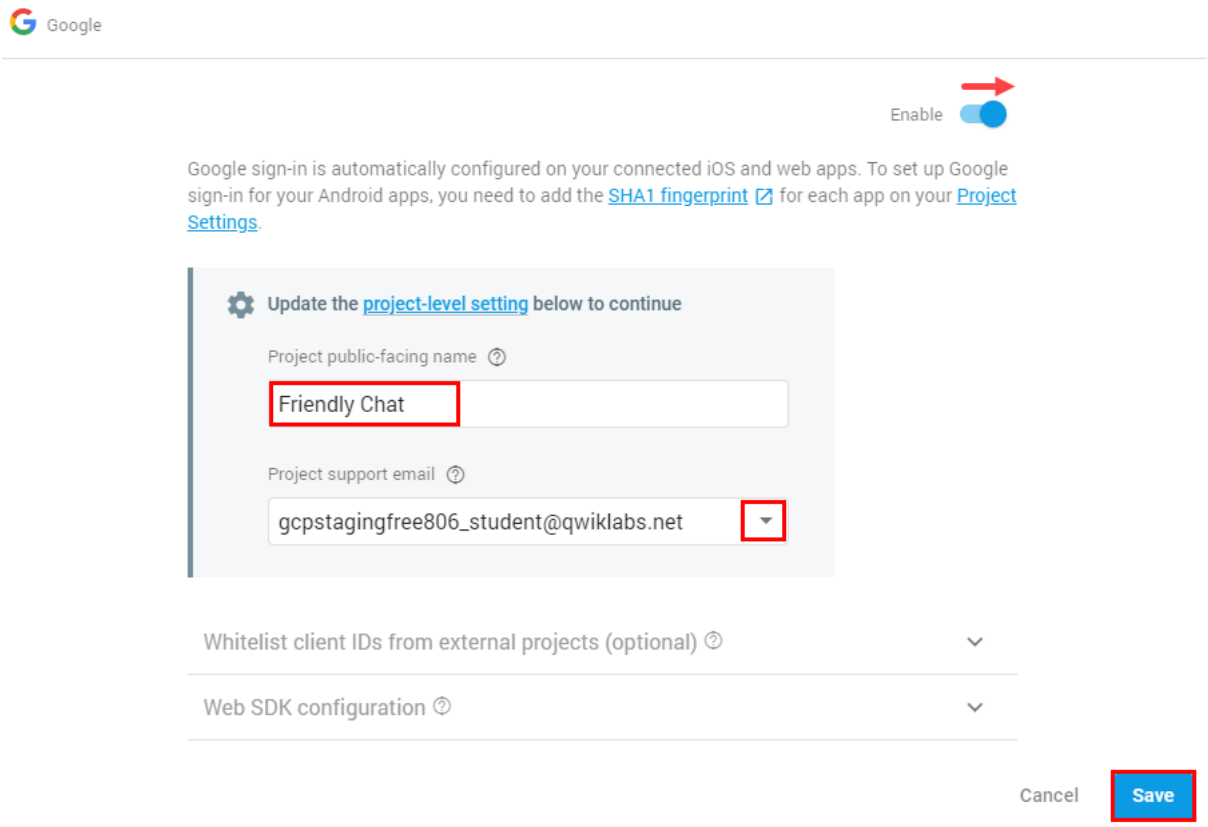
Enable Google sign-in for Firebase Authentication

Now enable Google Authentication to allow users to sign in to the web app with their Google accounts.

1. Click on the **Build** tile.
2. Navigate to **Authentication** > **Get started** > **Sign-in method**.



3. In the **Additional Providers** section, click the **Google** button.
4. Move the toggle to **Enable**.
5. Give the project the public-facing name `Friendly Chat`.
6. Click in the **project support email** field, select your lab email.
7. Then click **Save**.

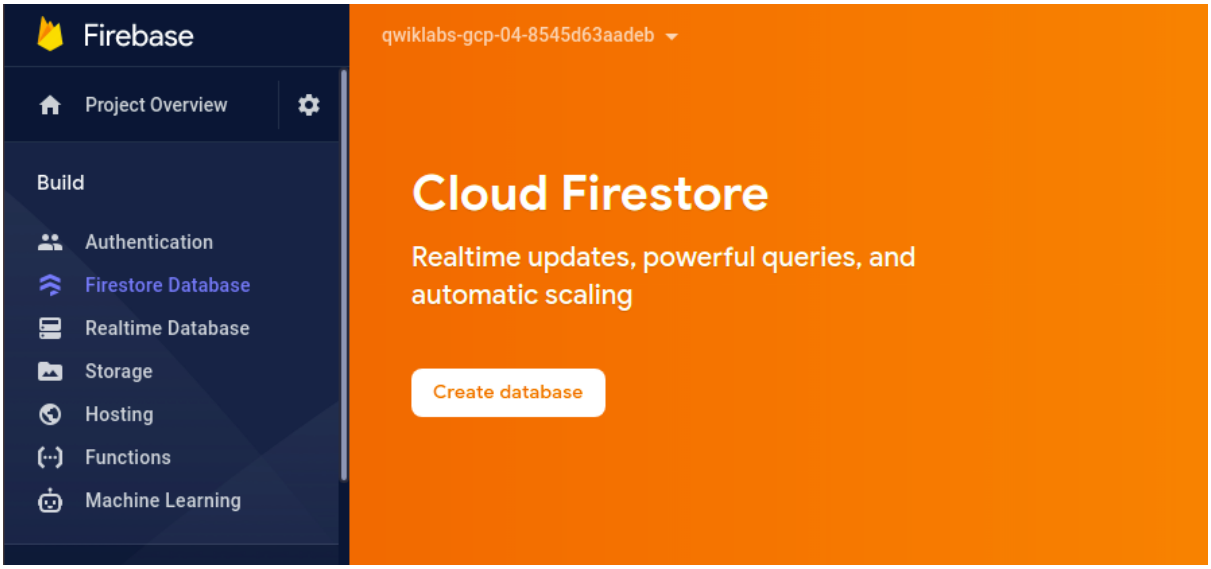


Now a user can sign into the app with their Google account credentials.

Enable Cloud Firestore

The app uses Cloud Firestore to save the chat messages and receive new chat messages.

1. To enable Cloud Firestore on your Firebase project, select **Firestore Database** from the left menu.
2. Then, click **Create database** under Cloud Firestore.



3. Select the **Start in test mode** option.
4. Then, click **Next**.
5. Leave the default Cloud Firestore location, click **Enable**.

Task 7. Install the Firebase command line interface

Cloud Shell comes with the `firebase` command line interface (CLI) already installed.

1. In Cloud Shell, check what version of Firebase is installed with the following command:

```
firebase --version
```

The output should show a version above 6.0.0.

2. Authorize the Firebase CLI:

```
firebase login --no-localhost
```

The `--no-localhost` option is used because you are on a remote shell.

3. When asked if you should **Allow Firebase to collect CLI usage and error reporting information?**, type "**Y**".
4. Copy the link in the output into a new tab.
5. Select your lab username.
6. Then, click **Allow**.
7. On **Sign in to the Firebase CLI** page, click `Yes I just ran the command` and then click `Yes, this if my session ID`.
8. Copy the verification code from the browser and enter it in the Cloud Shell prompt.
9. Still in Cloud Shell command line, navigate to `friendlychat-web` > `web-start` directory by running:

```
cd ~/friendlychat-web/web-start/
```

10. Now set up the Firebase CLI to use your Firebase Project:

```
firebase use --add
```

You'll be asked which project to add.

11. Use the arrow keys to select your Project ID, then press **Enter**.
12. When asked **What alias do you want to use for this project? (e.g. staging)**, type `staging` and press **Enter**.

Note: Giving your project an alias is helpful if you are managing multiple apps/projects. You can switch between aliases in Cloud Shell with `firebase use <alias_name>.`

Task 8. Deploy and run the starter app

Now that you have imported and configured your project you are ready to run the app for the first time.

1. In Cloud Shell run the following command:

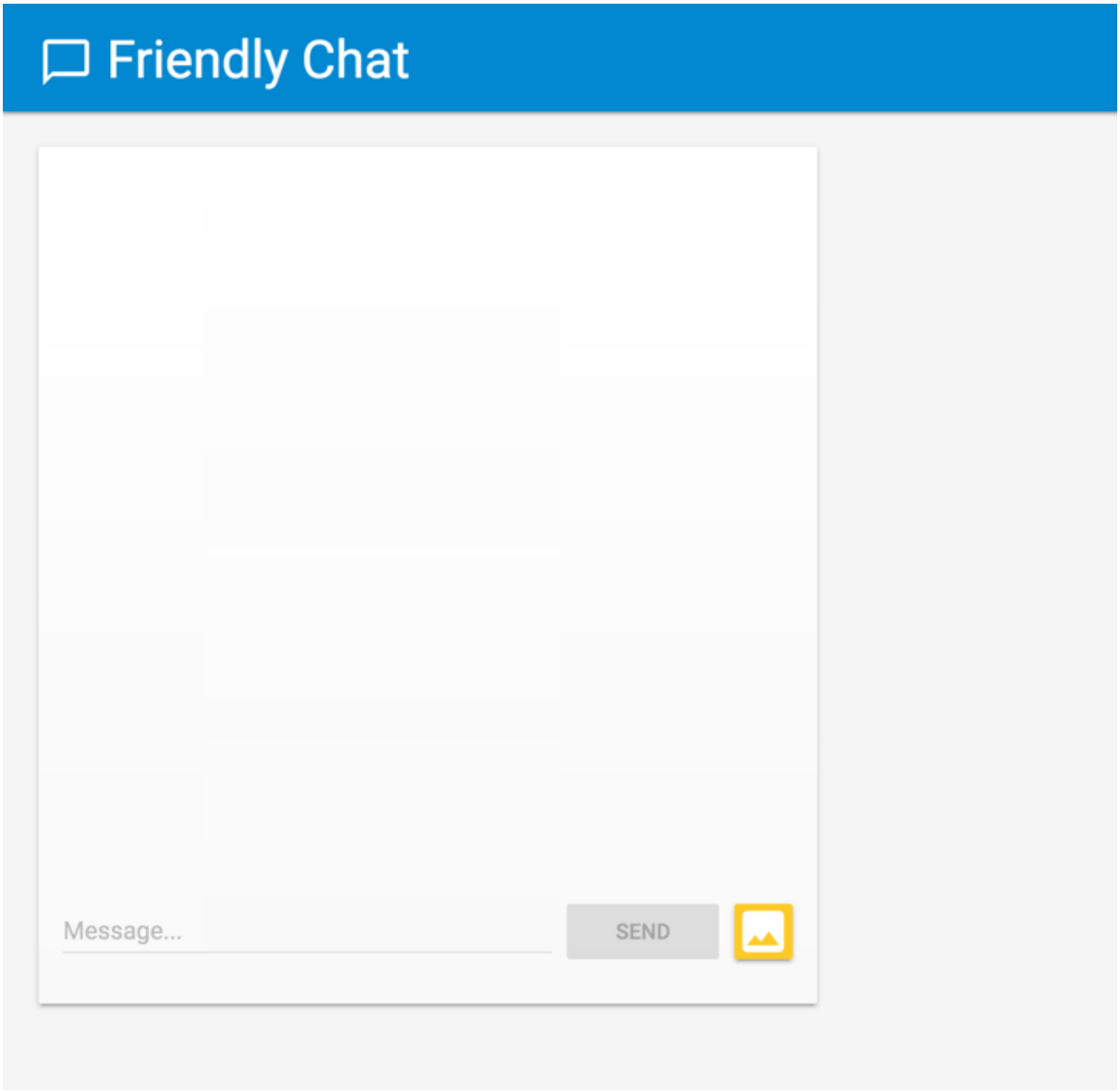
```
firebase serve --only hosting
```

Output:

```
i  hosting: Serving hosting files from: ./
✓  hosting: Local server: http://localhost:5000
```

You're using the Firebase hosting emulator to serve the application locally. The web app should now be available from `http://localhost:5000`.

2. Click the URL in your output to open it. You should see your app's (not yet!) functioning UI:



3. Stop the app by pressing **Ctrl + C**. This lab has laid out only the UI for you so far. Now you will build a realtime chat!

Task 9. Import and configure Firebase

Import the Firebase SDK

In a production environment the Firebase SDK must be imported into your application. There are multiple ways to do this that are described in the Add Firebase to your JavaScript project documentation.

You're going to get the Firebase SDK from npm and use Webpack to bundle your code. You're doing this so that Webpack can remove any unneccessary code, keeping your JS bundle size small to make sure your app loads as quickly as possible.

For this codelab, a `web-start/package.json` file that includes the Firebase SDK as a dependency has already been created, as well as imported the needed functions at the top of `web-start/src/index.js`.

During this lab, you're going to use Firebase Auth, Cloud Firestore, Cloud Storage, and Cloud Messaging, so you're importing all of their libraries. In your future apps, make sure that you're only importing the parts of Firebase that you need in order to shorten the load time of your app.

Install the Firebase SDK and start your Webpack build

1. In your Cloud Shell terminal, ensure you are in the `web-start` directory:

```
cd ~/friendlychat-web/web-start/
```

2. Run `npm install` to download the Firebase SDK:

```
npm install
```

3. Run `npm run start` to start up Webpack. Webpack will now continually rebuild your source code for the rest of the codelab.

```
npm run start
```

Configure Firebase

You also need to configure the Firebase SDK to tell it which Firebase project that you're using.

1. Go to your project setting in the Firebase console.
2. Select the tile with your project ID.
3. Scroll down to the "Your apps" card and select the **Friendly Chat** web app.
4. Under **SDK setup and configuration**, select **Config**.
5. Copy the config object snippet, then add it to `web-start/src/firebase-config.js` ensuring the object name remains `config`.

sample firebase-config.js:

```
const config = {
  apiKey: "API_KEY",
  authDomain: "PROJECT_ID.firebaseapp.com",
  databaseURL: "https://PROJECT_ID.firebaseio.com",
  projectId: "PROJECT_ID",
  storageBucket: "PROJECT_ID.appspot.com",
  messagingSenderId: "SENDER_ID",
  appId: "APP_ID",
  measurementId: "G-MEASUREMENT_ID",
};
```

Note: The above code should contain your app-specific Firebase config object, not the placeholder values!

6. Next, open `web-start/src/index.js` in your code editor.

7. At the bottom of the file, add the `initializeApp` call under the `firebaseAppConfig` variable:

```
const firebaseAppConfig = getFirebaseConfig();
initializeApp(firebaseAppConfig);
```

Note: Do not duplicate the assignation of the `firebaseAppConfig` variable.

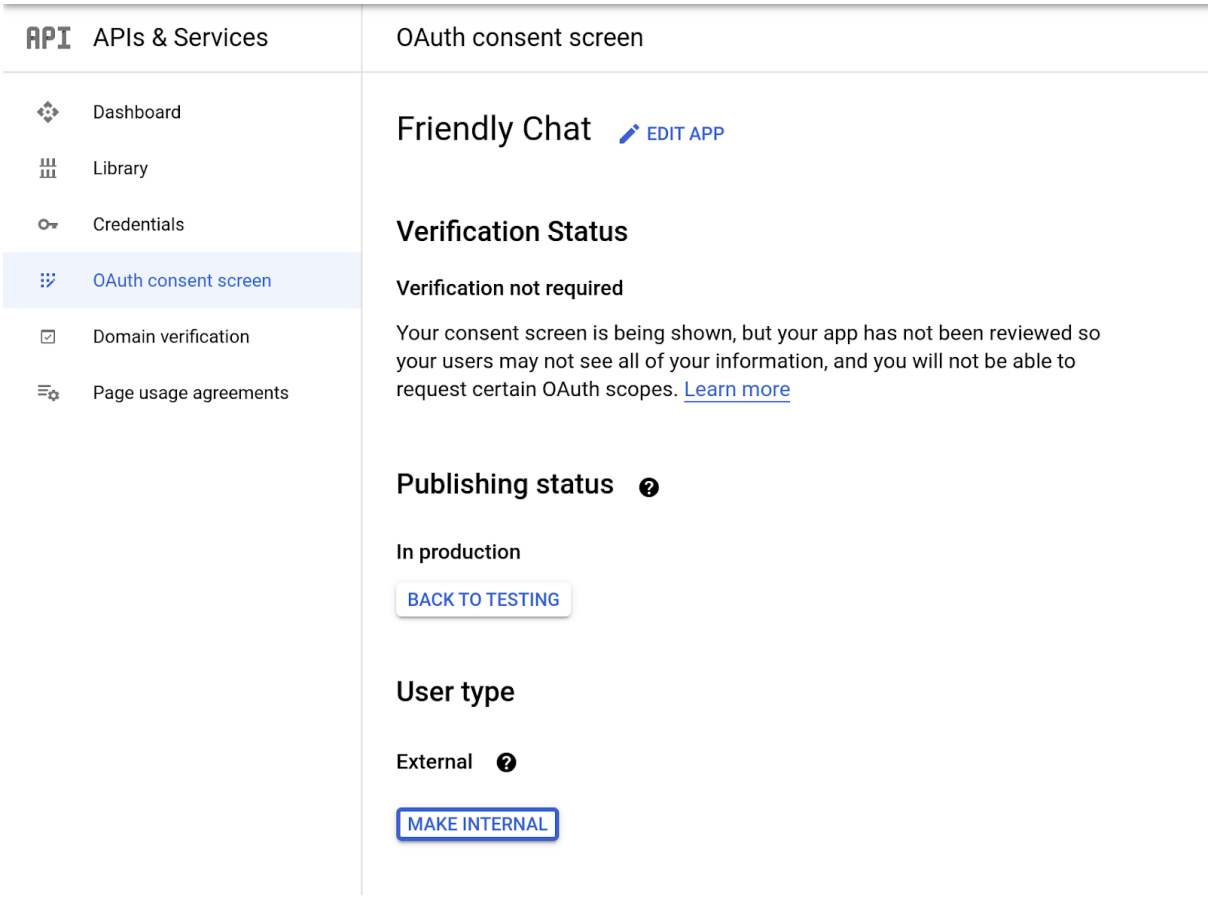
Task 10. Set up user sign in

The Firebase SDK was imported and initialized in the `index.html` file, so it's ready to use. Next, implement user sign-in using Firebase Auth.

Set up app's OAuth consent screen

The OAuth consent screen allows users to choose whether they want to grant access to their private data as well as gives them a link to your terms of service. For this lab, you'll use the "internal" setting for simplicity. Read through this page to understand what to provide for your production environment.

1. In the Cloud Console, from **Navigation menu** click on **APIs & Services** > **OAuth consent screen**.
2. For **User type**, Click **Make Internal**.



3. Then click **Confirm**.

Authenticate your users with Google Sign-In

When the user clicks the **Sign in with Google** button, the `signIn` function is triggered (this is already set up for you). Authorize Firebase to use Google as the Identity Provider. Users will sign in using a popup.

1. In the `web-start` directory, in the subdirectory `src/`, open `index.js`.
2. Find the function `signIn`.
3. Replace the entire function with the following code:

```
// Signs-in Friendly Chat.
async function signIn() {
```

```
  // Sign in Firebase using popup auth and Google as the identity
provider.
  var provider = new GoogleAuthProvider();
  await signInWithPopup(getAuth(), provider);
}
```

Copied!

content_copy

The `signOutUser` function is triggered when the user clicks the **Sign out** button.

4. Go back to the file `src/index.js`.
5. Find the function `signOutUser`.
6. Replace the entire function with the following code:

```
// Signs-out of Friendly Chat.
function signOutUser() {
  // Sign out of Firebase.
  signOut(getAuth());
}
```

Track the auth state

To update the UI accordingly, you need a way to check if the user is signed-in or signed-out. With Firebase Auth, you can register an observer on the authentication state which will be triggered each time the auth state changes.

1. Go back to the file `src/index.js`.
2. Find the function `initFirebaseAuth`.
3. Replace the entire function with the following code:

```
// Initialize firebase auth
function initFirebaseAuth() {
  // Listen to auth state changes.
  onAuthStateChanged(getAuth(), authStateObserver);
```

```
}
```

This registers the function `authStateObserver` as the observer. It triggers every time there is a change in the auth state - when the user signs in or signs out. This function updates the UI to display or hide the **Sign-in** button, the **Sign-out** button, and the signed in user's profile picture.

Display the signed in user information

In Firebase, the signed-in user's data is always available in the `firebase.auth().currentUser` object. The `authStateObserver` function will call the `getProfilePicUrl` and `getUserName` when triggered.

1. Go back to the file `src/index.js`.
2. Find the functions `getProfilePicUrl` and `getUserName`.
3. Replace both functions with the following code:

```
// Returns the signed-in user's profile Pic URL.
function getProfilePicUrl() {
  return getAuth().currentUser.photoURL ||
'/images/profile_placeholder.png';
}
// Returns the signed-in user's display name.
function getUserName() {
  return getAuth().currentUser.displayName;
}
```

If the user tries to send a messages when they are not signed in, the app should display an error message. To detect if the user is actually signed in, you will change the `isUserSignedIn` function.

4. Go back to the file `src/index.js`.
5. Find the function `isUserSignedIn`.

6. Replace the entire function with the following code:

```
// Returns true if a user is signed-in.
function isUserSignedIn() {
  return !!getAuth().currentUser;
}
```

7. **Save** `src/index.js`.

Test signing-in to the app

1. Return to your Cloud Shell terminal. It should still be running the Webpack process.
2. Open a new Cloud Shell tab by pressing the **+** button.
3. In your new tab, change to the `web-start` directory:

```
cd ~/friendlychat-web/web-start/
```

4. Check which version of **Node** you have:

```
node --version
```

5. If the output of the previous command is `v16.1.0` or lower, run the command:

```
nvm use v16.15.0
```

Copied!

content_copy

6. Redeploy your app:

```
firebase deploy --except functions --token $(gcloud auth
application-default print-access-token)
```

7. Click on the *Hosting URL* to open the application in your browser.

8. Sign in using the **Sign-In with Google** button. Remember to use your lab credentials!

After Signing in the profile pic and name of the user will be displayed:

Task 11. Write messages to Cloud Firestore

Next you'll write some data to Cloud Firestore so that you can populate the app's UI. This can be done manually in the Firebase console, but for this lab you'll do it in the app itself to demonstrate basic Cloud Firestore write.

Data model

Firestore data is split into collections, documents, fields, and subcollections. Each message of the chat is stored as a document in a top-level collection called `messages`.

Add messages to Firestore

In this section you'll add the functionality to let users write new messages to Cloud Firestore. A user clicking the **SEND** button will trigger the code snippet below. It adds a message object with the contents of the message fields to your Firestore instance in the `messages` collection.

The `add()` method adds a new document with an automatically generated ID to the collection.

1. Go back to the file `src/index.js`.
2. Find the function `saveMessage`.
3. Replace the entire function with the following code:

```
// Saves a new message to Cloud Firestore.
async function saveMessage(messageText) {
  // Add a new message entry to the Firebase database.
  try {
    await addDoc(collection(getFirestore(), 'messages'), {
      name: getUserName(),
      text: messageText,
      profilePicUrl: getProfilePicUrl(),
      timestamp: serverTimestamp()
    });
  }
  catch(error) {
    console.error('Error writing new message to Firebase Database',
error);
  }
}
```

Test sending messages

1. Redeploy your app:

```
firebase deploy --except functions --token $(gcloud auth
application-default print-access-token)
```

Copied!

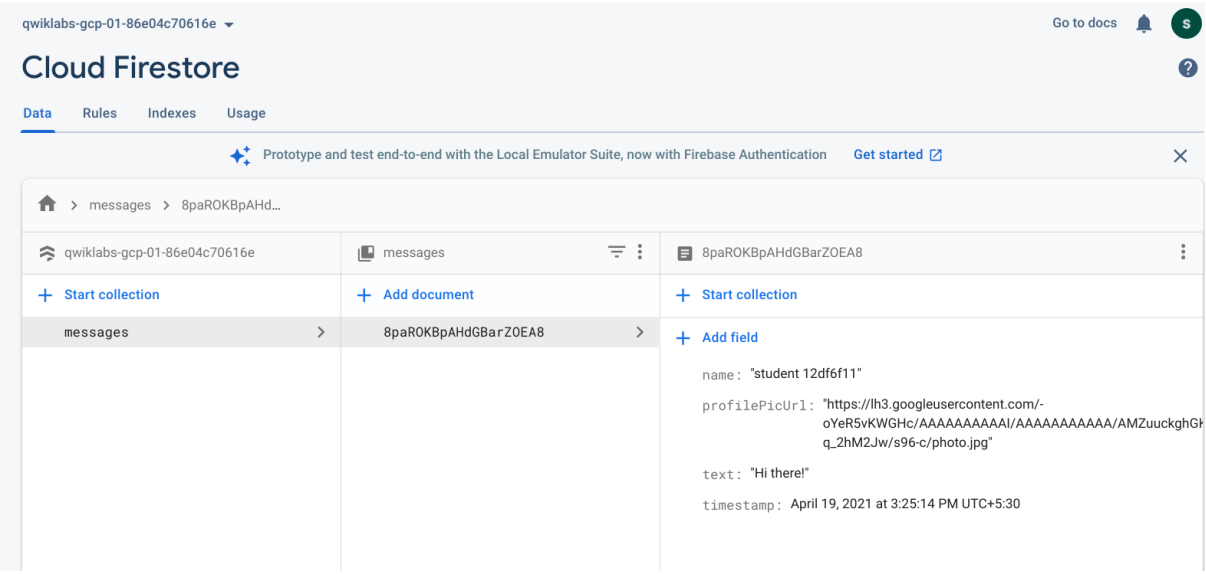
content_copy

2. Click on the *Hosting URL* to open the application in your browser, or paste the URL into a new browser (not incongnito or guest mode).
3. After signing-in, enter a message such as “Hi there!”, then click **SEND**.

This will write the message into Firestore. However, you won't yet see the data in your actual web app because you still need to implement retrieving the data (the next section of the lab).

You can see the newly added message in your Firebase Console.

4. In the Build section, click on **Firestore Database** and you should see the messages collection with your newly added message:



Task 12. Read messages

Synchronize messages

To read messages on the application, add listeners that trigger when changes are made to the data then creates a UI element that shows new messages.

You'll add code that listens for newly added messages from the app. In this code, you'll register the listener that listens for changes made to the data. You'll only display the last 12 messages of the chat to avoid displaying a very long history upon loading.

1. Go back to the file `src/index.js`.
2. Find the function `loadMessages`.
3. Replace the entire function with the following code:

```
// Loads chat messages history and listens for upcoming ones.
function loadMessages() {
  // Create the query to load the last 12 messages and listen for new
ones.
  const recentMessagesQuery = query(collection(getFirestore(),
'messages'), orderBy('timestamp', 'desc'), limit(12));
  // Start listening to the query.
  onSnapshot(recentMessagesQuery, function(snapshot) {
    snapshot.docChanges().forEach(function(change) {
      if (change.type === 'removed') {
        deleteMessage(change.doc.id);
      } else {
        var message = change.doc.data();
        displayMessage(change.doc.id, message.timestamp, message.name,
                       message.text, message.profilePicUrl,
message.imageUrl);
      }
    });
  });
```

```
}
```

To listen to messages in the database, create a query on a collection by using the `.collection` function to specify in which collection is the data you want to listen to.

Above, you're listening to the changes under the `messages` collection, which is where the messages are stored. You're also applying a limit and only listening to the last 12 messages using `.limitToLast(12)` and ordering the messages by date using `.orderBy('timestamp','desc')` to get the 12 newest messages.

The `.onSnapshot` function takes one parameter: a callback function. The callback function will be triggered when there are any changes to documents that match the query. This could be if a message gets deleted, or modified, or added.

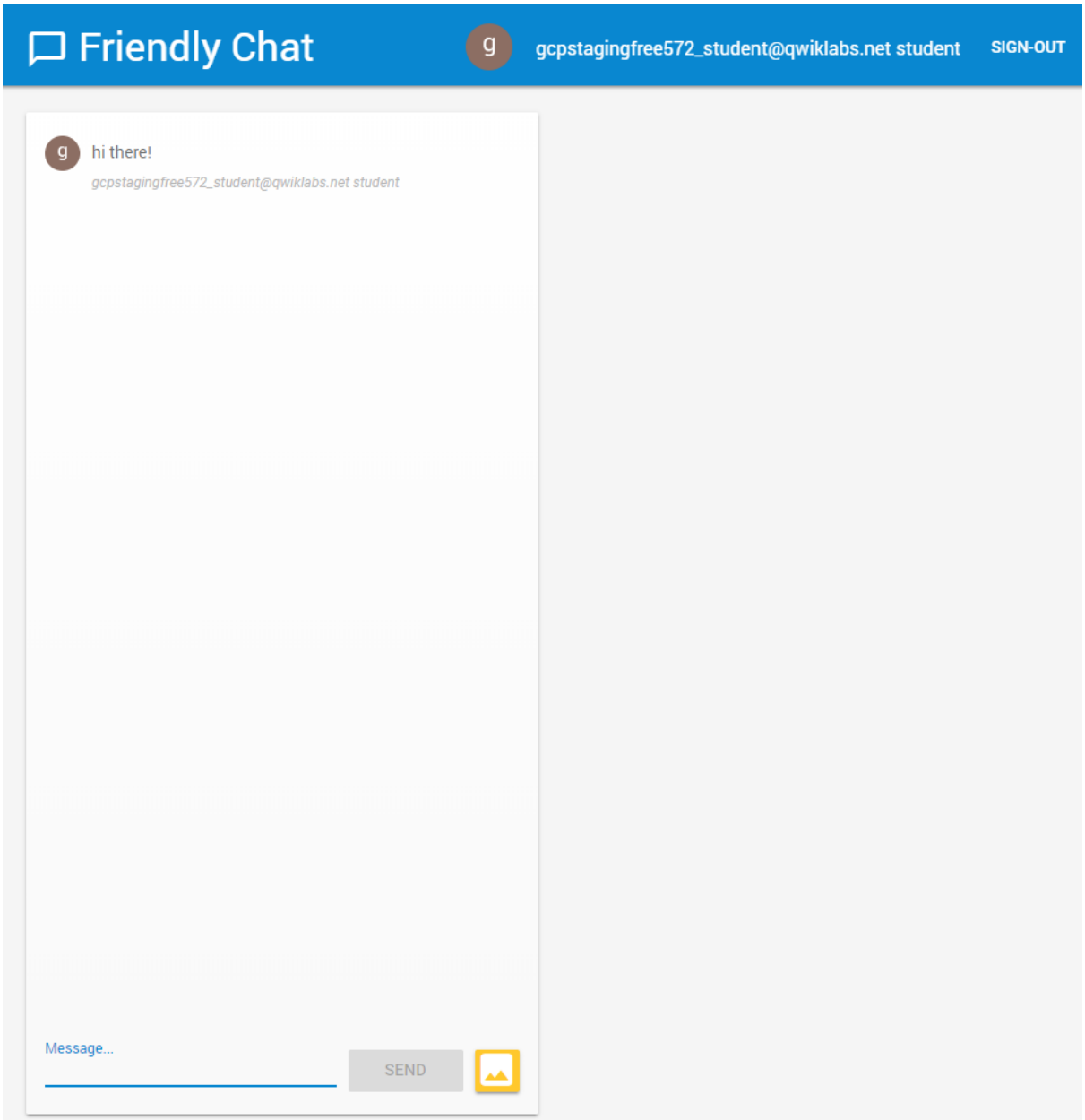
Test message sync

1. Redeploy your app:

```
firebase deploy --except functions --token $(gcloud auth
application-default print-access-token)
```

2. Click on the *Hosting URL* to open the application in your browser.

The messages you typed earlier and saw in the database should be displayed in the Friendly Chat UI. You can also write a new message, it should appear instantly. You can also try manually deleting, modifying, or adding new messages directly in the **Firestore** section of the Firebase Console, the changes should reflect in the UI.



Congratulations, you are reading database entries in your app!

Task 13. Send images

Now add a feature that shares images.

While the Cloud Firestore is good to store structured data, files are better stored in Cloud Storage. Cloud Storage for Firebase is a file/blob storage service. Use it to store the images the user shares.

In order to do this, you will need to grant access to your `Firebase Service Management Service Account` and configure security rules for Firebase Storage.

Grant access to service account

- First, give your `Firebase Service Management Service Account` the `Storage Admin` role:

```
gcloud projects add-iam-policy-binding $(gcloud config get-value
project) \

--member="serviceAccount:firebase-service-account@firebase-sa-managemen
t.iam.gserviceaccount.com" \
    --role="roles/storage.objectAdmin"
```

This will give Firebase services full access to your Google Cloud Storage resources.

Next, you will define Cloud Storage rules for Firebase.

Cloud storage security rules

Cloud Storage for Firebase uses a specific rules language to define access rights, security, and data validations.

When setting up the Firebase project at the beginning of this lab, default Cloud Storage security rules were used that restrict access to Cloud Storage. In the Firebase console, in the **Storage** section's **Rules** tab, you can view and modify the default rules, which should look like this:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if false;
    }
  }
}
```

You'll update the rules to allow any authenticated user to read and write from storage.

1. In the Firebase Console, go to the **Storage** section from the left navigation, then click the **Rules** tab.
2. Replace the default rules with the following rules:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /{userId}/{messageId}/{fileName} {
      allow read, write: if request.auth != null;
    }
  }
}
```

3. Click **Publish**.

Save images to Cloud Storage

There's already a button in the UI that triggers a file picker dialog. After selecting an image file, the `saveImageMessage` function is triggered and you get a reference to the selected file. You'll add code to the file that:

- Creates a "placeholder" chat message into the chat feed, so that users see a "Loading" animation while we upload the image.
- Upload the image file to Cloud Storage to the path: `/<uid>/<messageId>/<file_name>`.
- Generate a publicly readable URL for the image file.
- Update the chat message with the newly uploaded image file's URL in lieu of the temporary loading image.

Now actually add the functionality.

1. Go back to the file `src/index.js`.
2. Find the function `saveImageMessage`.
3. Replace the entire function with the following code:

```
// Saves a new message containing an image in Firebase.
// This first saves the image in Firebase storage.
async function saveImageMessage(file) {
  try {
    // 1 - We add a message with a loading icon that will get updated
with the shared image.
    const messageRef = await addDoc(collection(getFirestore(),
'messages'), {
      name: getUserName(),
      imageUrl: LOADING_IMAGE_URL,
      profilePicUrl: getProfilePicUrl(),
      timestamp: serverTimestamp()
    });
    // 2 - Upload the image to Cloud Storage.
    const filePath =
`${getAuth().currentUser.uid}/${messageRef.id}/${file.name}`;
    const newImageRef = ref(getStorage(), filePath);
    const fileSnapshot = await uploadBytesResumable(newImageRef, file);
```

```
    // 3 - Generate a public URL for the file.
    const publicImageUrl = await getDownloadURL(newImageRef);
    // 4 - Update the chat message placeholder with the image's URL.
    await updateDoc(messageRef,{
      imageUrl: publicImageUrl,
      storageUri: fileSnapshot.metadata.fullPath
    });
  } catch (error) {
    console.error('There was an error uploading a file to Cloud
Storage:', error);
  }
}
```

Test sending images

1. Redeploy your app:

```
firebase deploy --except functions --token $(gcloud auth
application-default print-access-token)
```

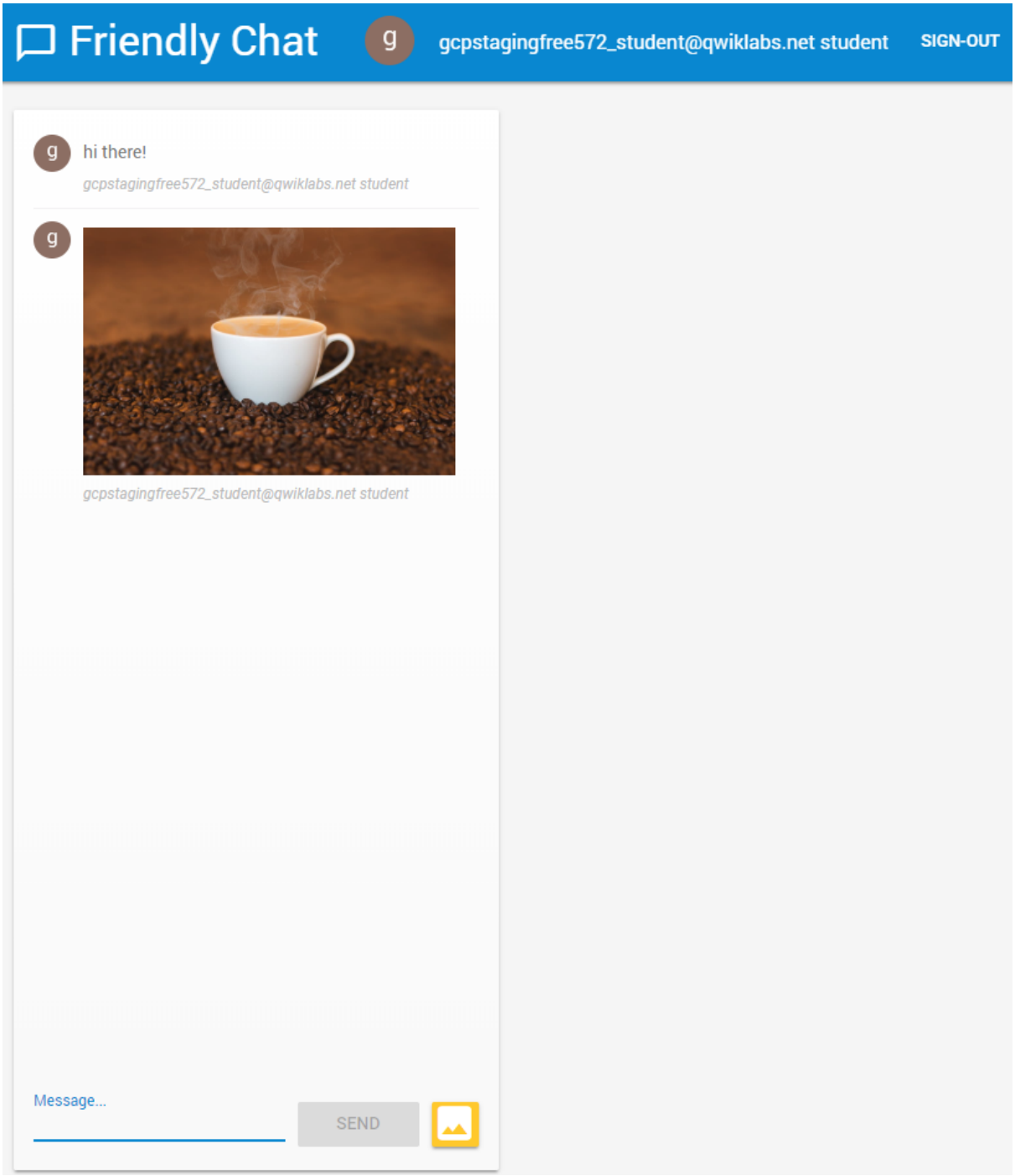
Copied!

content_copy

2. Click on the *Hosting URL* to open the application in your browser.

3. Click the image upload button  and select an image file from your computer using the file picker. If you're looking for an image, feel free to use this nice pic of a coffee cup.

A new message should be visible in the app UI with your selected image:



Note: It sometimes takes a short amount of time for the security rules and IAM permissions to take effect. Ensure you have hard refreshed your app.
Note: If you're still unable to see your image, wait a minute, close your tab, re-open it, and try uploading another image.
If you try adding an image while not signed in, you will see a Toast telling you that you must sign in to add images.

Task 14. Show notifications

Now add support for browser notifications so users receive a notification when a new message has been posted in the chat. Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably deliver messages and notifications at no cost.

Whitelist the GCM Sender ID

1. In the web app manifest, specify the `gcm_sender_id`, a hard-coded value,and indicate that FCM is authorized to send messages to this app.

Friendly Chat already has a `manifest.json` configuration file.

2. In the Cloud Shell code editor, navigate to `friendlychat-web` > `web-start`> `public` and open the `manifest.json` file. Update the browser sender ID exactly as shown (do not change the value):

```
{
  "name": "Friendly Chat",
  "short_name": "Friendly Chat",
  "start_url": "/index.html",
  "display": "standalone",
  "orientation": "portrait",
  "gcm_sender_id": "103953800507"
}
```

Add the FCM service worker

The web app needs a Service Worker that receives and displays web notifications.

1. From the `web-start` directory, in the `src` directory, open `firebase-messaging-sw.js`.
2. Add the following content to that file:

```
// Import and configure the Firebase SDK
import { initializeApp } from 'firebase/app';
import { getMessaging } from 'firebase/messaging/sw';
import { getFirebaseConfig } from './firebase-config';
const firebaseApp = initializeApp(getFirebaseConfig());
getMessaging(firebaseApp);
console.info('Firebase messaging service worker is set up');
```

The service worker needs to load and initialize the Firebase Cloud Messaging SDK, which will take care of displaying notifications.

Get FCM device tokens

When notifications are enabled on a device or browser, you'll be given a **device token**. This device token is used to send a notification to a particular device or browser.

When the user signs in, the `saveMessagingDeviceToken` function is called. That's where you'll get the FCM device token and save it to the Cloud Firestore.

1. Go back to the file `src/index.js`.
2. Find the function `saveMessagingDeviceToken`.
3. Replace the entire function with the following code:

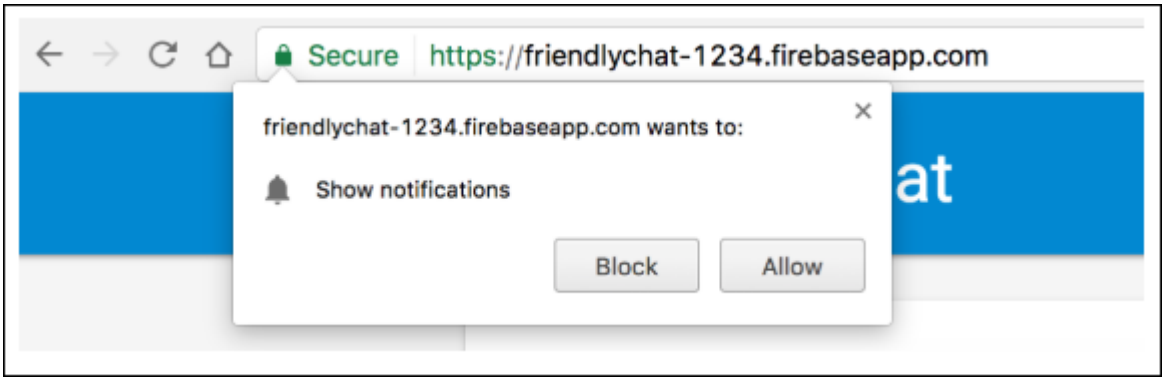
```
// Saves the messaging device token to Cloud Firestore.
async function saveMessagingDeviceToken() {
  try {
    const currentToken = await getToken(getMessaging());
    if (currentToken) {
      console.log('Got FCM device token:', currentToken);
      // Saving the Device Token to Cloud Firestore.
      const tokenRef = doc(getFirestore(), 'fcmTokens', currentToken);
      await setDoc(tokenRef, { uid: getAuth().currentUser.uid });
      // This will fire when a message is received while the app is in
the foreground.
      // When the app is in the background, firebase-messaging-sw.js
will receive the message instead.
      onMessage(getMessaging(), (message) => {
        console.log(
          'New foreground notification from Firebase Messaging!',
          message.notification
        );
      });
    } else {
      // Need to request permissions to show notifications.
      requestNotificationsPermissions();
    }
  } catch(error) {
    console.error('Unable to get messaging token.', error);
  };
}
```

4. **Save** the file.

For your app to be able to retrieve the device token, the user needs to grant your app permission to show notifications. You'll set that up next.

Request permissions to show notifications

To show notifications the user must give permission by calling the `firebase.messaging().requestPermission()` method, which will display a browser dialog asking for this permission in supported browsers:



1. Go back to the file `src/index.js`.
2. Find the function `requestNotificationsPermissions`.
3. Replace the entire function with the following code:

```
// Requests permissions to show notifications.
async function requestNotificationsPermissions() {
  console.log('Requesting notifications permission...');
  const permission = await Notification.requestPermission();
  if (permission === 'granted') {
    console.log('Notification permission granted.');
    // Notification permission granted.
    await saveMessagingDeviceToken();
  } else {
    console.log('Unable to get permission to notify.');
  }
}
```

4. **Save** the file.

Get your device token

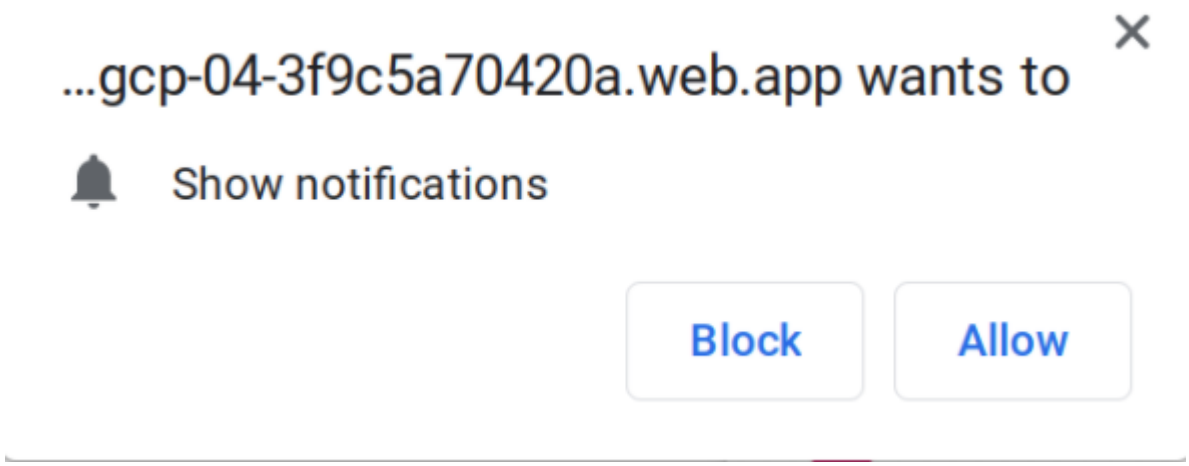
Note: If you are using Chrome, open this application in a browser window that is **NOT** Incognito or guest mode. You won't be able to see the notifications permission dialog if you are.

1. Redeploy your app:

```
firebase deploy --except functions --token $(gcloud auth
application-default print-access-token)
```

2. Click on the *Hosting URL* to open the application in your browser.

After signing in, you should see the Notifications permission dialog being displayed:



3. Click **Allow**.
4. Open the JavaScript console of your browser (If you are using Chrome, go to **More tools** > **Developer tools** > **Javascript Console**.) You should see a message that reads:

```
Got FCM device token: cWL6w:APA91bHP...4jDPL_A-wPP06GJp1OuekTaTZI5K2Tu
```

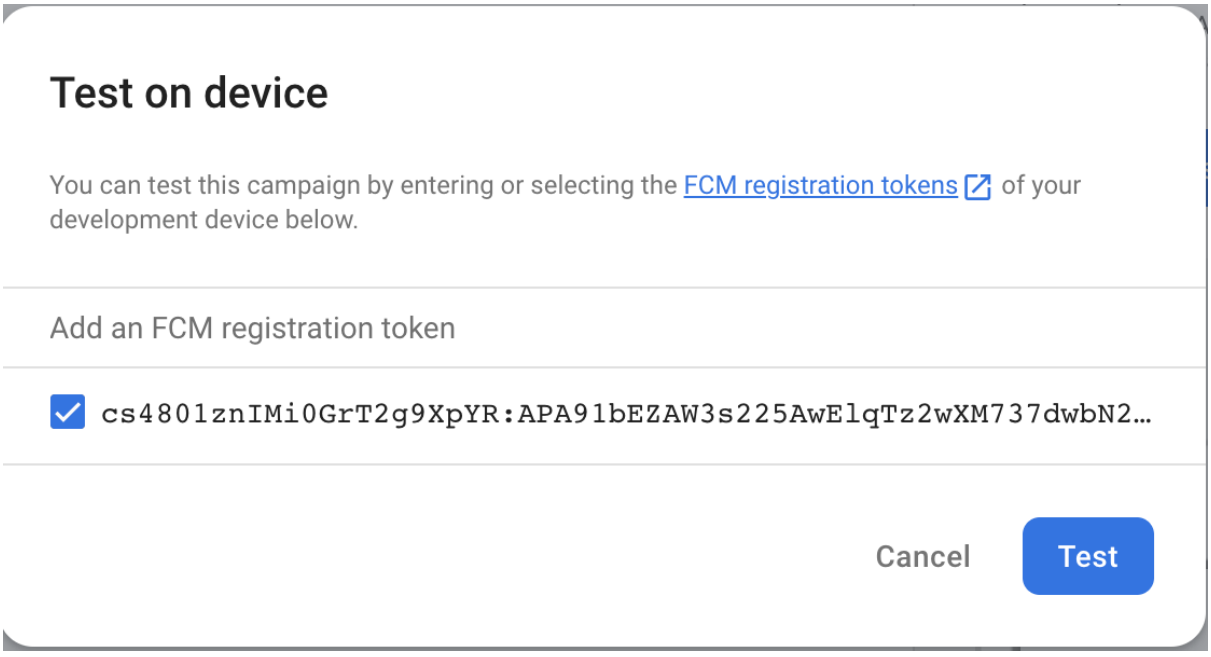
5. Copy and save your device token, you will need it for the next step.

Send a notification to your device

Now that you have your device token, you can send a notification.

1. Open the Cloud Messaging tab of the Firebase console.
2. Select your project ID.
3. Click **Send your first message**.

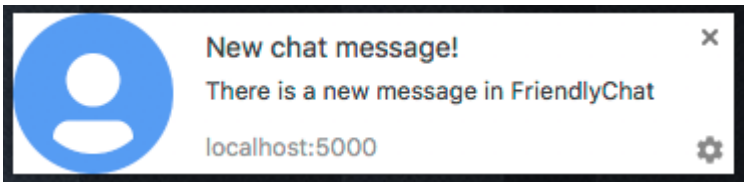
4. Enter **notification test** for the **Notification Title**.
5. Enter **Hello** for the **Notification Text**.
6. On the right side of the screen, click **send test message**.
7. Paste the device token you copied from the JavaScript console of your browser, then click the plus ("+") sign.



8. Click **Test**.

If your app is in the foreground, you'll see the notification in the JavaScript console.

If your app is in the background, a notification should appear in your browser, as in this example:



You may not see a notification if running in incognito mode or something similar. You can verify the message was received in the **Javascript console** of your **Friendly Chat** app browser tab.

Task 15. Cloud Firestore security rules (optional)

The Cloud Firestore uses specific rules language to define access rights, security, and data validations.

When setting up your Firebase project at the beginning of this lab, you chose to use "Test mode" default security rules that do not restrict access to the database.

In the Firebase console, in the **Firestore Database** section's **Rules** tab, you can view and modify these rules.

1. Go there now to see the default rules:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write;
    }
  }
}
```

2. When you update the rules to restrict access, use the following:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Messages:
    //   - Anyone can read.
    //   - Authenticated users can add and edit messages.
    //   - Validation: Check name is same as auth token and text length
below 300 char or that imageUrl is a URL.
    //   - Deletes are not allowed.
    match /messages/{messageId} {
      allow read;
      allow create, update: if request.auth != null
                    && request.resource.data.name ==
request.auth.token.name
                    && (request.resource.data.text is string
                      && request.resource.data.text.size() <= 300
                      || request.resource.data.imageUrl is string
                      &&
request.resource.data.imageUrl.matches('https?://.*'));
      allow delete: if false;
    }
    // FCM Tokens:
    //   - Anyone can write their token.
    //   - Reading list of tokens is not allowed.
    match /fcmTokens/{token} {
      allow read: if false;
      allow write;
    }
  }
}
```

There are two ways to edit the database security rules; in the Firebase console or from a local file deployed using the Firebase CLI.

Pick one of the following ways to update the rules.

1) Update security rules in the Firebase console

1. Go to **Firestore Database** in the left navigation and click on the **Rules** tab.

2. Replace the default rules with the rules above.
3. Click **Publish**.

Note: The `request.auth` rule variable is a special variable containing information about the user if authenticated. The `request.resources` rule variable points to the new data being written. More information can be found in the Understand Firebase Realtime Database Rules documentation.

2) Update security rules from a local file

Note: If you have updated database security rules using Firebase console then skip this step and move forward directly to the **Cloud Storage security rules** step.

1. In the `web-start` directory, create a file called `firestore.rules`.
2. Add the rules shown above.
3. From the `web-start` directory, open `firebase.json`.
4. Add the `firestore.rules`attribute, as shown below. (The `hosting` attribute should already be in the file.)

```
{
  // Add this!
  "firestore": {
    "rules": "firestore.rules"
  },
  "hosting": {
    "public": "./public"
  }
}
```

5. Deploy the security rules using the Firebase CLI by running the following command:

```
firebase deploy --only firestore --token $(gcloud auth
application-default print-access-token)
```

Output:

```
=== Deploying to 'qwiklabs-gcp-29dca141bb7bec58'...
i  deploying firestore
i  firestore: checking firestore.rules for compilation errors...
✓  firestore: rules file firestore.rules compiled successfully
i  firestore: uploading rules firestore.rules...
✓  firestore: released rules firestore.rules to cloud.firestore
✓  Deploy complete!
Project Console:
https://console.firebase.google.com/project/friendlychat-1234/overview
```

Task 16. Deploy your app using Firebase hosting

Firebase comes with a hosting service to serve your web app. You deploy your files to Firebase Hosting using the Firebase CLI.

1. Before deploying you need to specify which files will be deployed in your `firebase.json` file.

This has already been done for you because it was required to serve the file for development through this lab. These settings are specified under the `hosting` attribute:

```
{
  // If you went through the "Cloud Firestore Security Rules" step.
  "firestore": {
    "rules": "firestore.rules"
  },
```

```
  // If you went through the "Storage Security Rules" step.
  "storage": {
    "rules": "storage.rules"
  },
  "hosting": {
    "public": "./public"
  }
}
```

This tells the CLI that you want to deploy all files in the `./public` directory (`"public": "./"`).

2. Confirm that you are still in you app's local `web-start` directory, then deploy your files to Firebase project by running:

```
firebase deploy --except functions --token $(gcloud auth
application-default print-access-token)
```

Output:

```
=== Deploying to 'friendlychat-1234'...
i  deploying database, storage, hosting
i  database: checking rules syntax...
✓  database: rules syntax for database friendlychat-1234 is valid
i  storage: checking storage.rules for compilation errors...
✓  storage: rules file storage.rules compiled successfully
i  storage: uploading rules storage.rules...
i  hosting: beginning deploy...
i  found 8 files in ./public
✓  hosting: file upload complete
i  database: releasing rules...
✓  database: rules for database qwiklab-gcp-xxxxx released successfully
✓  storage: released rules storage.rules to
firebase.storage/qwiklab-gcp-xxxxx.appspot.com
i  hosting: finalizing version...
✓  hosting: version finalized
i  hosting releasing new version
✓  hosting: release complete
✓  Deploy complete!
Project Console:
https://console.firebase.google.com/project/qwiklab-gcp-xxxxx/overview
```

```
Hosting URL: https://qwiklab-gcp-xxxxx.web.app
```

3. Visit your web app hosted on Firebase Hosting on `https://<project-id>.web.app`.
4. Go to the Hosting section in the Firebase console to view useful hosting information and tools, including the history of your deploys, the functionality to roll back to previous versions of your app, and the workflow to set up a custom domain.

Congratulations!

You used Firebase to easily build a real-time chat application. You covered:

- Authorizing Firebase
- Cloud Firestore
- Firebase SDK for Cloud Storage
- Firebase Cloud Messaging
- Firebase Hosting