Learning Goals

* Implement communication features into a native app
* Implement modern JavaScript APIs into an app
* Use and customize a third-party library

 Estimated Read Time: 1 Hour.

Introduction

Welcome back! In the last Exercise, you learned about various client-side storage systems for your web apps and websites, looking at how and when to use each type of storage to make your apps more performant. Additionally, you learned how to use React Native’s AsyncStorage API to store data on your users’ mobile devices. Finally, you put your new knowledge to the test in the task—storing your chat app’s messages locally so your users can reread their messages while offline.

Your focus for this final Exercise of this Achievement will be the additional communication features you can build into your chat app, such as the ability to share images and location data. To implement such capabilities, you need to learn how to access users’ hardware components via their operating system’s APIs. Furthermore, you’ll examine some ethical considerations for building mobile apps. After all, as a developer, respecting your users’ rights and privacy is of utmost importance.

Without further ado, let’s get started!

Communication, Tracking, and Measurement Features

When building apps for browsers or mobile devices, you can use some special features provided by the platform’s environment, like the ability to access a device’s location, microphone, and camera.

If you go to [Google Maps](https://www.google.com/maps) in your Chrome browser and click the **Padlock** icon to the left of the web address, a browser overlay should pop up with an on-off toggle for **Location** sharing permission (Figure 1). If you toggle this on (and update your device’s system settings), Google Maps will be able to read your location data from the device and display your current location on the map.

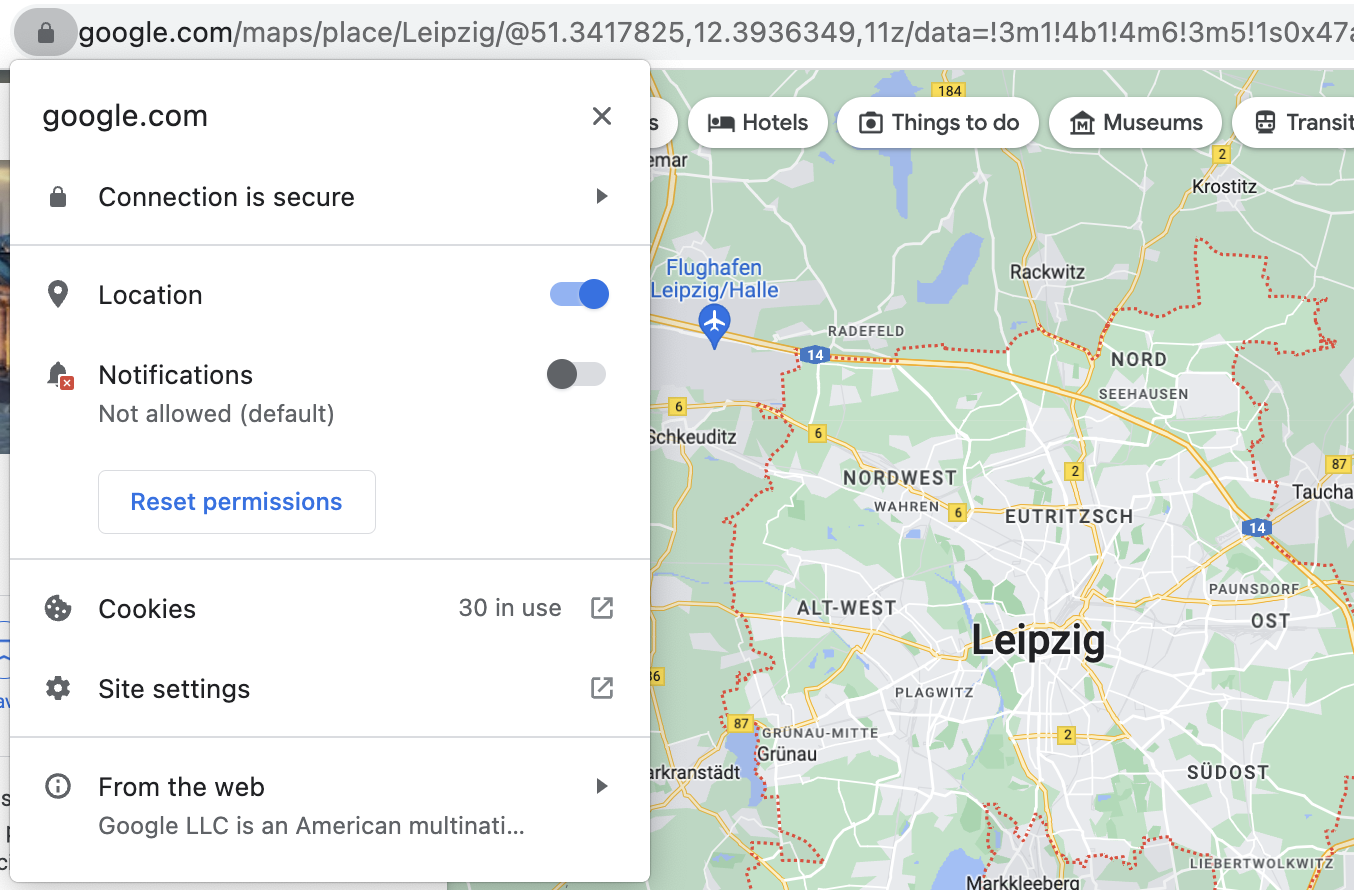


Figure 1

Likewise, if you visit a browser-based video-chat tool like Google Meet, you’ll need to give your browser permission to use your device’s microphone and camera. Otherwise, you can’t use it.

You can make use of similar features when building native mobile apps. Mobile devices generally have a set of specific hardware components that apps can access, such as a camera, a microphone, a barometer (to measure air pressure), motion and orientation sensors, a gyroscope (to measure orientation and angular velocity), and GPS for location services.

Imagine a fitness app that tracks your steps, how fast you run, and so on. Such an app requires access to several of your device’s hardware components to function correctly. To display a map of your running route, for instance, it would need to track your location. Runtastic (Figure 2) is an example of a fitness app that does this.

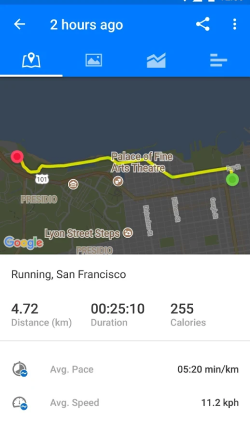


Figure 2

While you can’t always expect your users’ PCs or MacBooks to have components like a microphone or camera, it’s reasonably safe to assume that their mobile devices will have these parts. That’s why developing for mobile devices opens up a bunch of new opportunities.

Let’s say you want to create a game that involves navigating a ball through a labyrinth. On a mobile device, you could activate the motion sensors, allowing the device to double as a game controller. When the user tilts the phone to the left, the ball moves to the left; when the user tilts the phone forward, the ball begins to roll faster. Now, imagine trying to play this type of game on a PC or Mac—not much fun, right?

Over the last couple of years, speech-recognition services have surged in popularity. Siri, Cortana, Alexa, and Google Assistant are installed on various devices, including smartphones, computers, and speakers, and they listen and respond to the needs of millions of people daily. For example, an iPhone owner might ask Siri to write a text message to their friend, while an Amazon Echo customer might ask Alexa when the Leaning Tower of Pisa was built.

For the native chat app that you’re currently building, you can use your phone’s speech-to-text functionality to compose messages—simply tap the microphone icon on your keyboard and start speaking. It doesn’t matter which OS you’re running: both Android and iOS support this functionality.

Because your device natively supports this feature, it can be used to compose text in any app installed on your smartphone that’s capable of voice-to-text recognition—for example, iMessage, Android Message, or WhatsApp. Additionally, Windows and MacOS support speech-to-text, so you could even let your OS write out notes you dictate on your computer.

Browser Functionality  
You may be wondering what your browser is capable of in addition to native mobile and desktop features. There are still a lot of things that you can do natively in a mobile app that are less performant or not well supported in the browser—tracking gestures or a device’s orientation are two examples.

Browser creators, including Google and Mozilla, are working on a range of APIs to bring mobile functionality to the web. For instance, there’s a set of Sensor APIs that allow you to read device sensors like the gyroscope, accelerometer, and so on. The Web Speech API can similarly be used to integrate speech recognition, allowing speech-to-text recognition in browser-based apps.

Speech Services in React Native

As you’ve just learned, there’s no need to integrate a speech-to-text API into your React Native app because users can, instead, use their device’s built-in speech-to-text functionality (which is accessible from the keyboard that appears beneath the text input field).

But what if you wanted to convert speech to text without giving users a text input field? Perhaps you want to create a to-do app that records the tasks dictated by a user, saving them as text in the database. In cases like these, where you don’t want to be reliant on the device’s speech-to-text keyboard shortcut, you can use libraries such as [React Native Speech Recognition](https://github.com/trestrantham/react-native-speech-recognition). Be aware, though, that most of the time, you’ll need to provide the user with a text input field so they can decide for themselves whether to manually type out or dictate text—resulting in a better and more accessible user experience.

As you might recall from [Exercise 5.1: Building Native Apps with JavaScript](https://careerfoundry.com/en/course/full-stack-immersion/exercise/native-applications), native apps are typically built using native languages such as Java (Android) or Swift/C# (iPhone). The toolchain Expo provides a way around this by helping you build native apps using JavaScript. Expo has developed its own set of APIs to bridge the gap between JavaScript and your device’s features. These include GPS, Camera, and ImagePicker. With the [ImagePicker API](https://docs.expo.dev/versions/latest/sdk/imagepicker/" \t "_blank), for instance, you can use JavaScript code to select images from a device’s image gallery.

Although Expo doesn’t have a speech-to-text API just yet, it does have a [Speech API](https://docs.expo.dev/versions/latest/sdk/speech/), which allows you to integrate text-to-speech capabilities into your projects. To let a device’s speech synthesizer read text aloud, you can use Speech.speak(text, options). Here, options is an object in which you can define the language to be spoken, the pitch and rate of the speaking voice, and callback functions that are invoked when the service starts or finishes reading:

Speech.speak(‘Hello there’, {

language: 'en',

pitch: 1,

rate: 0.75,

onStart: myStartCallback,

onDone: myCompleteCallBack

});

Try it out yourself by following along with the [example in the official documentation](https://docs.expo.dev/versions/latest/sdk/speech/).

In summary, if you want to integrate *speech-to-text* recognition into your apps, you should use the **device’s native speech-to-text** feature. On the other hand, when you want a device’s speech service to read messages *aloud* (i.e., text to speech), use Expo’s Speech API.

Speech Services in the Browser: WebSpeech  
If you were developing a chat app for the browser, you could use the [Web Speech API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API) to implement speech-to-text and text-to-speech functionality. The Web Speech API provides a SpeechSynthesis interface, which converts text to speech, and a SpeechRecognition interface, which converts speech to text.

Check out Google’s [Web Speech API Demo](https://www.google.com/intl/en/chrome/demos/speech.html) to give SpeechRecognition a try.

To see SpeechSynthesis in action, take a look at this [Speech Synthesizer Demo](https://mdn.github.io/dom-examples/web-speech-api/speak-easy-synthesis/). You can find more Web Speech API demos [here](https://github.com/mdn/dom-examples/tree/main/web-speech-api). If you decide to use this API for a future project, note that it isn’t currently supported by all modern browsers, so you’ll need to warn your users that its features may not work in their current browser.

Chat App Communication Features

Now that you’ve explored the communication features landscape for different platforms and devices, it’s time to find out how you can add these communication features to your chat app. More specifically, you’re going to learn how to access the images, camera, microphone, and geolocation of your users’ mobile devices.

To show how these features can be implemented, let’s leave your Achievement 5 example Shopping Lists app project for the time being and, instead, create a new demo project. Creating a new project allows you to experiment freely with the new functionalities you’re going to explore without running into conflicts with existing code.

You’ll also get the chance to adapt what you’ve learned for your main project. Later in the Exercise, you’ll look at how to integrate what you’ve learned into your chat app codebase.

First, create a new Expo project and name it “hello-communication-features” (or something similar), then start it. You’ll write all the code into the App.js file, so go ahead and open it in your code editor.

Media Library and Camera

You may already know from using apps like WhatsApp or Telegram that you can select photos from your phone’s media library to send to your contacts or take new photos to send with the device’s camera. You can make these two features available to your users through Expo’s [ImagePicker API](https://docs.expo.dev/versions/latest/sdk/imagepicker/" \t "_blank), which gives you access to a device’s UI to select images and videos from the library or to take photos with the camera.

You’re going to use the following four methods from the ImagePicker API:

* requestMediaLibraryPermissionsAsync: This displays a prompt to the user asking for permission to access the media library.
* launchImageLibraryAsync: This opens the device’s media library to let the user choose a file.
* requestCameraPermissionsAsync: This displays a prompt to the user asking for permission to access the camera.
* launchCameraAsync: This opens the device’s camera and allows the user to take a photo.

Remember, you can’t just access the camera on your user’s device without asking the user for their permission first! Each Expo API includes functions for requesting permissions for features specific to that API. It’s unethical to access any of the features or content on a user’s device that could involve using sensitive or private data without being granted permission by the user.

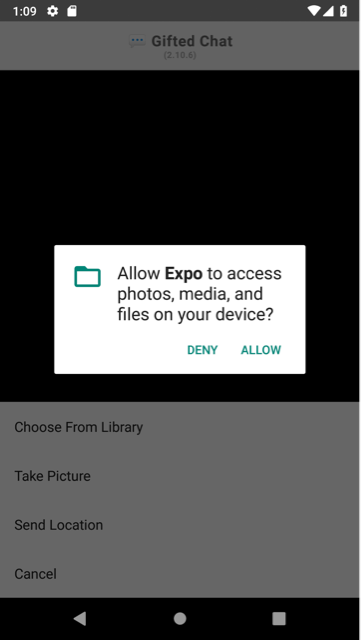


Figure 3

Now, let’s walk through how to allow users of your chat app to select a photo from their library or take a new photo. To use the ImagePicker API, you first need to install the expo-image-picker Expo package. Run the following command in the terminal:

expo install expo-image-picker

Once the installation is complete, import the package into the component file you want to use it in. For this example, the file will be App.js:

import \* as ImagePicker from 'expo-image-picker';

Import Notation  
The asterisk (\*) in the import statement just shown is used to import **everything** exported from the expo-image-picker package. Stating “as ImagePicker” allows you to then reference them as a collective object named ImagePicker.

Next, define a new state, image, using useState. This state will represent the URI source of the image selected from the gallery (or taken from the camera).

const [image, setImage] = useState(null);

You also need to render the following two buttons on your app screen so that users can indicate whether they would like to share an existing image or take a new one:

1. Pick an image from the library.
2. Take a photo.

To create these buttons, go ahead and create a View and add two Button elements to it. As soon as the user presses one of these buttons, you want to call a custom function that either opens up the media library or camera—for this, you use the button’s prop onPress:

const App = () => {

const [image, setImage] = useState(null);

return (

<View style={styles.container}>

<Button

title= “Pick an image from the library”

onPress={pickImage}

/>

<Button

title= “Take a photo”

onPress={() => {}}

/>

</View>

);

}

const styles = StyleSheet.create({

container: {

flex: 1,

justifyContent: 'center'

}

});

export default App;

Next, you need to add some functionality to these buttons. Let’s start with the first one.

**Button 1: “Pick an image from the library”**

Did you notice the pickImage function reference that was used in the onPress event listener of the top button in the code? Let’s begin with this function: pickImage.

pickImage is a callback function. So, create a new async function. You’re going to use two functions, ImagePicker.requestMediaLibraryPermissionsAsync() and ImagePicker.launchImageLibraryAsync(), which both return a promise. Furthermore, you’ll be using async/await to handle them:

const pickImage = async () => {

let permissions = await ImagePicker.requestMediaLibraryPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchImageLibraryAsync();

if (!result.canceled) setImage(result.assets[0]);

else setImage(null)

}

}

At the start of the pickImage function, ImagePicker.requestMediaLibraryPermissionsAsync() is called first, which is asynchronous. How does this work?

1. If the user grants you access to their device’s gallery, this function returns an object that has many properties. In the example, this is referenced by the permissions variable. The property you want to check from the returned object is .granted, which is a boolean.
2. After the user has granted you access to their media library (permissions?.granted is true), call ImagePicker.launchImageLibraryAsync() to let them pick a file.

By default, this method only allows you to select images, however, if you want to let the user choose any different media type (or allow them all) you’ll need to pass an object to ImagePicker.launchImageLibraryAsync().

In this object, you can specify whatever media type the user is allowed to pick. For example, to allow only videos:

let result = await launchImageLibraryAsync({

mediaTypes: ImagePicker.MediaTypeOptions.Videos

});

In the code snippet just shown, MediaTypeOptions is an object accessed from ImagePicker, which you can access from the expo-image-picker module because of the way the import statement is written.

ImagePicker.MediaTypeOptions contains three possible properties: Images, Videos, and All. Videos was used in the example, but for your communication features demo app, you’ll want to only use images. So, for now, delete the options object entirely, or set mediaTypes: ImagePicker.MediaTypeOptions.Images (however, the latter isn’t needed because it’s the default).

Now let’s take a look at the rest of the asynchronous code for the pickImage function. ImagePicker.launchImageLibraryAsync() returns an object (referenced by result in the code example) containing the .assets property. This is an array referencing all of the picked media files (i.e., “assets”). Let’s break the result code down:

* In this example, the array will be limited to only having **one asset**. By default, the user isn’t allowed to pick multiple assets.
* The asset contains a .uri (a string representing the path to the picked media file) along with its .width, .height, and file type.
* The object result also contains the boolean property '.canceled'. This will be true if the user cancels the process and doesn’t pick a file.
* In this example, you’ll just be assigning result.assets[0] to the image state using the setImage() state setter function.

Of course, once the user has chosen an image, you’ll want them to be able to do something with it—and they will! Eventually, you’ll display the image in a chat bubble sent to other chat participants. For now, though, you’re simply going to display the image in your View, so let’s add an Image element.

The only time you want to render this element is when the user has picked an image. The following example’s image has been given a width and height of 200, but feel free to resize it as you like:

{image &&

<Image source={{ uri: image.uri }} style={{ width: 200, height: 200 }} />

}

When inputting this in your App.js file, you can add the Image above or below the buttons. However, make sure that it’s inside the View container.

Run your demo project in the Expo app (on your mobile device) or the emulator/simulator (on your computer) and see what happens. The following functionality should be available in your app so far:

* The first screen should contain two buttons, **Pick an image from the library** (clickable) and **Take a photo** (not currently clickable, as you haven’t yet added this functionality).
* When clicking the **Pick an image from the library** button, a permissions modal appears, asking to “Allow Expo Go to access photos, videos, music, and audio on this device.”
* When selecting “Allow,” the app navigates to the device’s image library.
* From here, an image can be “picked.”
* The picked image is then added to the main app view.

Figure 4 depicts a photo that’s selected from the gallery and then rendered in a view on Android:

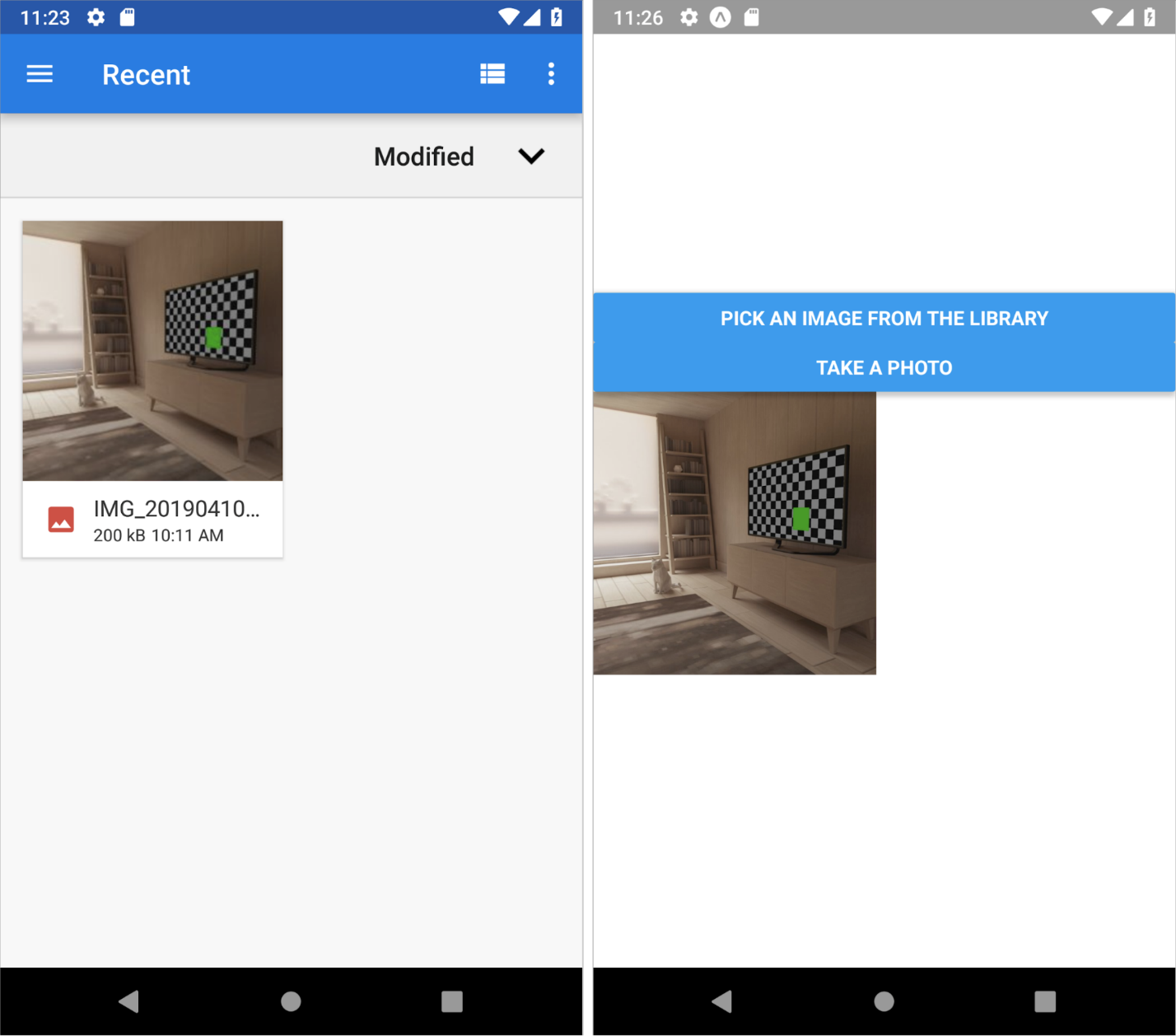


Figure 4

Troubleshooting Tips and Code Check  
This section covers a few common problems or errors (and solutions!) that you might encounter with the Expo Go app.

**Expo Go Keeps Stopping (Offline Error)**

The emulator might bug out and default to having the WiFi **off** when it’s launched (particularly if you’ve turned the WiFi off during testing in previous Exercises). So, always make sure to turn on the WiFi right away each time you launch the app. Not turning on the WiFi will prevent you from being able to run your apps through Expo Go.

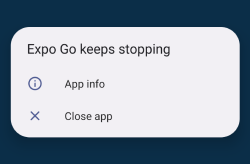


Figure 5

**Adding Images from your PC/Mac to an Emulator (e.g., in Android Studio)**

If you don’t have any images on your emulator, no worries. You can drag and drop an image from your PC/Mac into the emulator. The image will be in the “Downloads” folder.

When you test the app and tap on the **Pick an image from the library** button, you can navigate to the “Downloads” folder by pressing the hamburger icon in the upper-left corner.

**Manually Setting Permissions**  
If you mistakenly refuse to give permissions twice while testing the app on the emulator, you won’t be able to raise the prompt again. However, you can manually allow permissions again to “Expo Go.” To do this, bring up the apps grid menu on the emulator, open “Settings” → “Apps” → "Expo Go” (look for it) → “Permissions.” In “Permissions,” you’ll find “Camera” and “Photos and videos.” Tap to enable them, and allow permissions.

**Code Check**  
If you’re still encountering errors, you can check your code against our example [pickImage demo Expo Snack](https://snack.expo.dev/@cfcurriculum/comms_features_pickimage_demo" \t "_blank)

**Button 2: “Take a photo”**

Awesome! Next, you want to let the user take a picture. For this, you need to access the camera with launchCameraAsync. This function works similarly to launchImageLibraryAsync, in that it’s an asynchronous function that returns an object containing the media file data. Go ahead and create another async function, takePhoto:

const takePhoto = async () => {

let permissions = await ImagePicker.requestCameraPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchCameraAsync();

if (!result.canceled) setImage(result.assets[0]);

else setImage(null)

}

}

Note that you request permissions using the ImagePicker.requestCameraPermissionsAsync() method. Also, remember to use the takePhoto function as the second button’s callback, instead of the empty placeholder arrow function (() => {}).

The UI code for the demo app should now look like this:

<View style={styles.container}>

<Button

title="Pick an image from the library"

onPress={pickImage}

/>

<Button

title="Take a photo"

onPress={takePhoto}

/>

{image &&

<Image source={{ uri: image.uri }} style={{ width: 200, height: 200 }} />

}

</View>

With this code in place, the button should now function as expected. Here’s how it should work:

1. A modal appears, asking for permission to use the camera.
2. The camera opens, and the user can take a picture.
3. The user approves of the image.
4. Image added to the main app screen.

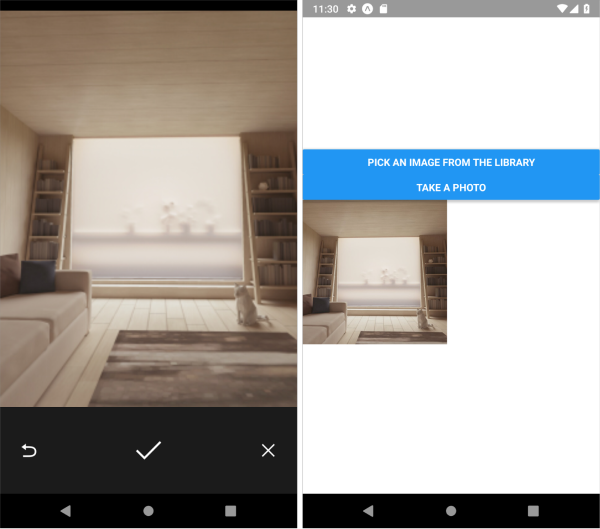


Figure 6. Image is taken and accepted on Android (left); image rendered in app view (right).

Your turn! Run your working version of the demo app in Expo Go to test your code!

Troubleshooting Tips and Code Check  
If you’re having problems with your code, consider the following common issues encountered by other developers to troubleshoot your code.

**Common Camera Bug**  
If you get the following message (Figure 7) on the Android Studio emulator, try closing and relaunching. Doing so should get rid of the error.

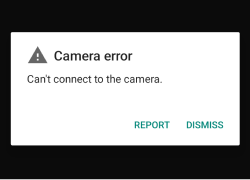


Figure 7

**Code Check**  
If you’re still encountering errors, compare your code against our [takePhoto demo Expo Snack](https://snack.expo.dev/@cfcurriculum/comms_features_pickimage-takephoto_demo" \t "_blank).

Congratulations! You’ve implemented two key communication features into a native app: selecting an image from a media library and taking an image to be sent. For now, you’ve simply rendered these images in a view, but you’ll shortly be learning how to integrate these features into the Gifted Chat project that you’ve been building throughout this Achievement.

Additional Functionality (Optional)  
There are some additional functionality opportunities available. While helpful, they’re optional in this Exercise, so feel free to jump ahead.

**Persisting Photos on the Device’s Storage**  
Photos captured using a device’s camera with the ImagePicker API won’t be saved into the device’s gallery by default. You’ll have to use Expo’s MediaLibrary API to save the photos. To do this, work through the following steps:

1. Install expo-media-library: expo install expo-media-library
2. Import everything as MediaLibrary from the module: import \* as MediaLibrary from 'expo-media-library';
3. Update the takePhoto function’s code to save the photo:

const takePhoto = async () => {

let permissions = await >ImagePicker.requestCameraPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchCameraAsync();

if (!result.canceled) {

let mediaLibraryPermissions = await >MediaLibrary.requestPermissionsAsync();

if (mediaLibraryPermissions?.granted) await >MediaLibrary.saveToLibraryAsync(result.assets[0].uri);

setImage(result.assets[0]);

}else setImage(null)

}

}

The code you’ve added asks permission from the media library (which includes both reading and writing to it). If granted, the code will call MediaLibrary.saveToLibraryAsync() while passing the URI of the photo asset to it.

**Recording Video**  
If you want to let your app’s users record videos, you can use the [Expo Camera API](https://docs.expo.dev/versions/latest/sdk/camera/) (this is also an alternative you can use to let users take pictures). It’s very useful, but you need to be aware of a few things:

* The API can’t access the device’s UI to select existing images and videos. To do so, you need to integrate both the Camera API and the ImagePicker API into your app.
* The API provides functionality for taking photos or recording videos, but it doesn’t provide the **interface** for doing so. You have to implement a capture button by yourself. Additionally, the API doesn’t work on simulators or emulators, making it tricky to test the functionality you’re implementing. The Expo Camera API won’t be a requirement for your current project.

Geolocation

The process of retrieving the user’s location is similar to what you’ve already learned. First, you ask for the user’s permission to read their location, and then you retrieve the required data with an asynchronous method provided by Expo. The API that you would use in this instance is Expo’s [Location API](https://docs.expo.dev/versions/latest/sdk/location/).

Open your “hello-communication-features” test project in your code editor again and start it from your terminal with expo start.

The first step is to import the Location API. You want to display the location data in a map (Google Maps on Android and Apple Maps on iOS), so you also need to import the MapView component.

To import the aforementioned APIs and tools, make sure you install the expo-location and react-native-maps Expo packages (while the Expo team itself didn’t make the latter, it *is* [their recommended package](https://github.com/react-native-community/react-native-maps)). Go ahead and run the following commands in your terminal:

expo install expo-location

expo install react-native-maps

Now, you can import them into your code as follows:

import \* as Location from 'expo-location';

import MapView from 'react-native-maps';

Next, define a state, location, that will hold the location data:

const [location, setLocation] = useState(null);

Add a new button to the app’s view and give it a title (e.g., “Get my location”). When the user presses the button, you want to call your custom function getLocation, which you’ll create in just a moment:

<Button

title="Get my location"

onPress={getLocation}

/>

Next, you need to create the getLocation method. In this method, you’ll request permission to access the device’s location using Location.requestForegroundPermissionsAsync(). As soon as the user grants you permission, you’ll want to read their location data using the Location.getCurrentPositionAsync() location method, which is an async method, so make the getLocation function is also async by adding the async keyword. This method returns an object with the coordinates of the user’s location, which you need to render the correct mapView. You’ll then use setState to update the state location, just like you did when working with images:

const getLocation = async () => {

let permissions = await Location.requestForegroundPermissionsAsync();

if (permissions?.granted) {

const location = await Location.getCurrentPositionAsync({});

setLocation(location);

} else {

Alert.alert("Permissions to read location aren't granted");

}

}

Great! All that’s left to do now is display the user’s current location. As mentioned earlier, you’ll use the MapView component to render a map in your view. This MapView has a prop region, which you’ll update with the device’s current location coordinates. You also need to set latitudeDelta and longitudeDelta, as they [determine the size of the map](https://stackoverflow.com/questions/50882700/react-native-mapview-what-is-latitudedelta-longitudedelta). Finally, give your MapView a width and a height; otherwise, it will set itself to zero, and you’ll see nothing at all!

{location &&

<MapView

style={{width: 300, height: 200}}

region={{

latitude: location.coords.latitude,

longitude: location.coords.longitude,

latitudeDelta: 0.0922,

longitudeDelta: 0.0421,

}}

/>}

At this point, your example app should look something like the images in Figure 8. You’ve created a new button to get the user’s location. When that button is pressed, the user is asked for permission to access the device’s location. If the user selects **Allow** the MapView renders the user’s current location.

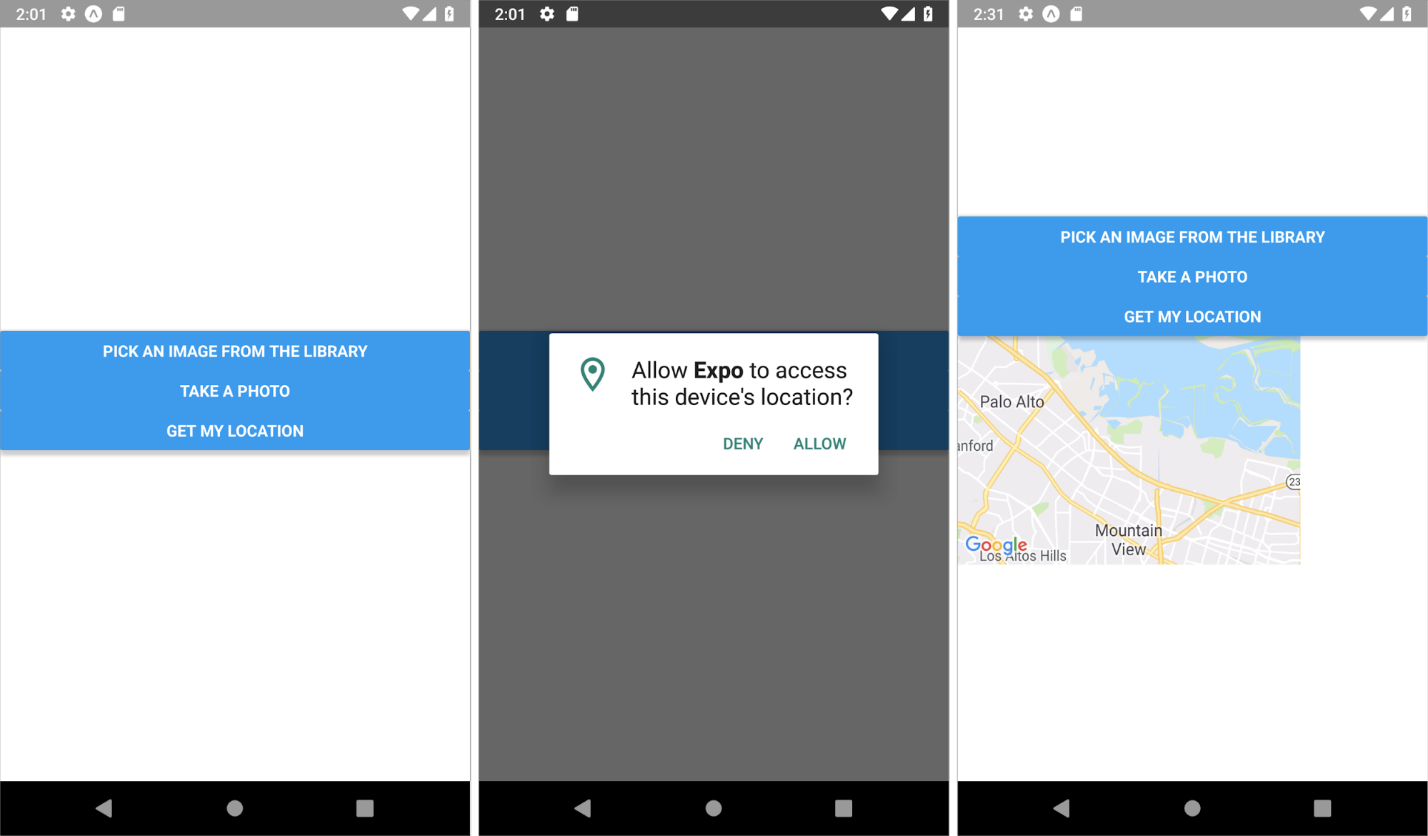


Figure 8

Testing and Code Check  
As with all the communication features so far, there are some things to bear in mind when testing your app’s new code.

**Testing with Android Emulator**  
When testing on Android Emulator, when you give the read location permission for the first time, it won’t work. However, this will just occur the first time. If you click “Get my location” again, the emulator will render the map of your location. You don’t need to worry about this when testing on an actual device (Android or iPhone) or iOS Simulator.

**Code Check**  
Code still not working? Compare what you’ve written in your own App.js file against our [location demo Expo snack](https://snack.expo.dev/@cfcurriculum/comms_features_location_demo). You’ll see that it’s been built on the image demo from earlier, so that code is in there too.

Keep in mind that you can’t run the demo app on ExpoSnack, as it requires native device features. So make sure to run the app on an emulator or actual device.

Congratulations! You’ve now learned how to retrieve images from a device and acquire a device’s location. You’ll apply this knowledge in this Exercise’s task. But first, let’s look at how to implement these features in Gifted Chat.

React Gifted Chat: Integrating Communication Features

In the sample project that you’ve worked on throughout this Exercise, you’ve created three different buttons that let the user 1) pick an image from their media library, 2) take a photo, and 3) view their current location.

However, you can’t integrate these buttons into your chat app as you did in the sample project because there simply isn’t space for them—when working on mobile apps, you have limited screen real estate! As such, you need a solution that gives users access to these features without taking up too much space on their screens.

Fortunately, the creator of the Gifted Chat library is one step ahead! The GiftedChat component has a renderActions prop that lets you create an “action” button to the left of the message composer (input field). You can see this button on the left-hand side of Figure 9 (the “+” sign). A list of options is displayed when a user presses this button (Figure 9, right).

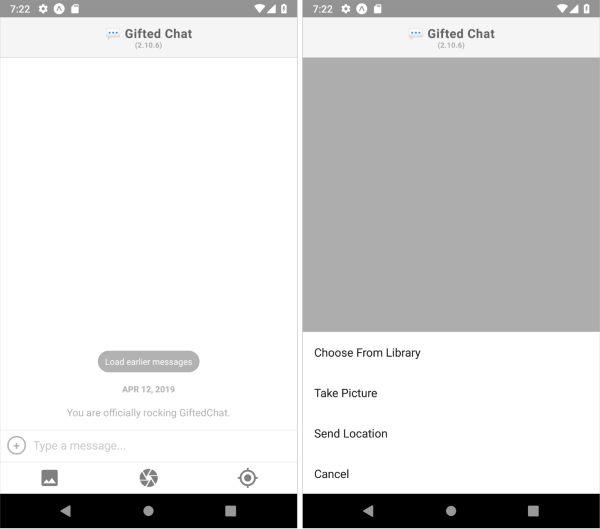


Figure 9

An accessory bar is also rendered along the bottom of the screen, below the text-input field (Figure 9, left). This bar acts as an alternative to the “action” button. The three buttons in this accessory bar essentially serve the same purpose as the “action” button—allowing users to pick an image, take a photo, or share their location. For your chat app, however, you’ll only recreate the “action” button, which will bring up a menu (Figure 9, right). This more minimalist approach leaves more space for messages to be displayed in the chat view!

Learning from Libraries  
If you need to work with a library, it’s always advisable to see if its documentation contains code examples to demonstrate certain features and functionality. These examples can help save you time when evaluating a library or starting your project.

The Gifted Chat library has provided a great example of the “action” button. To see it for yourself, [download the Gifted Chat library from GitHub](https://github.com/FaridSafi/react-native-gifted-chat), open your terminal, and navigate into the “example-expo” folder. From there, start the demo project with expo start. You won’t be working on it yourself (you’ll be working directly on your project!), but you can use it as a learning resource.

To help prepare you for this Exercise’s task, let’s explore how to build and incorporate the “action” button into your chat app.

First, you need to use the renderActions prop. Go ahead and add it to your GiftedChat component. Then, you need to provide the prop with the renderCustomActions function, which you’ll create in the next step:

renderActions={renderCustomActions}

The renderCustomActions function is responsible for creating the circle button. Add it to your Chat.js file, inside the component but above its return (...); code. You’ll create a new component for the custom actions that works with the props that this function receives:

const renderCustomActions = (props) => {

return <CustomActions {...props} />;

};

As you’re going to create a new component for the circle button, you’ll also need to create a new file, CustomActions.js, in the “components" folder. You’ll write your code for the new component in this file. However, to work with this file, you have to import it into Chat.js (at the top, along with your other imports):

import CustomActions from './CustomActions';

Once you’ve created your CustomActions.js file, define the component and export it as a default (export default). Furthermore, render the “action” button as a TouchableOpacity button. In Exercise 5.1, you learned—using React Native—how to create buttons with predefined styles. To customize the new button’s style, you can use [React Native’s Touchables](https://reactnative.dev/docs/handling-touches).

Exercise 5.1’s task briefly touched on TouchableOpacity, so you’re probably already familiar with “touchable” components that capture tapping gestures and trigger an action when a gesture is recognized (basically like a button). TouchableOpacity is just one of many touchable components in React Native (you can explore the others on the React Native page that was just linked).

As these components don’t have any default styling, you’ll need to style them yourself. In this case, a touchable is the best choice because you want to create a circular button with a “+” at its center, and you can’t style regular React Native buttons like that.

So far, your code in CustomActions.js should look like this:

import { TouchableOpacity } from "react-native";

const CustomActions = () => {

return (

<TouchableOpacity style={styles.container}

</TouchableOpacity>

);

}

export default CustomActions;

Note!  
In the ongoing example, [TouchableOpacity](https://reactnative.dev/docs/touchableopacity" \t "_blank)’s been used as a wrapper that reduces the button’s opacity when it’s pressed down. This allows the user to see the screen’s background. However, feel free to use any of the other [touchables](https://reactnative.dev/docs/handling-touches" \t "_blank) in this list.

When the user clicks the button, you’ll want to call the onActionPress function (you’ll create this soon), which will display a menu containing options such as taking a photo, selecting a photo, or sharing a location. As some initial prep work, it’s a good idea to create an empty function inside the CustomActions component declaration:

const CustomActions = () => {

const onActionPress = () => {}

return (

<TouchableOpacity style={styles.container} onPress={onActionPress}>

</TouchableOpacity>

);

}

Inside TouchableOpacity, add a View that you’ll style to look like a circle. Within this View, add the necessary text to display the “+” symbol:

return (

<TouchableOpacity style={styles.container} onPress={onActionPress}>

<View style={[styles.wrapper, wrapperStyle]}>

<Text style={[styles.iconText, iconTextStyle]}>+</Text>

</View>

</TouchableOpacity>

);

You might be wondering what wrapperStyle and iconTextStyle are. Let’s take a step back to Chat.js, where you’ve just added renderActions={renderCustomActions} to <GiftedChat />. Gifted Chat will make these props available to whatever renderXXXX function you pass to, expecting that you want to add some defaults to the components you aim to customize. This is why you’re passing these props to the custom component you’re trying to create and return in the renderCustomActions() function of Chat.js:

const renderCustomActions = (props) => {

return <CustomActions {...props} />;

};

This process is exactly what happened when you added “InputToolbar” in renderInputToolbar() in [Exercise 5.4: Storing Data on the Client Side](https://careerfoundry.com/en/course/full-stack-immersion/exercise/storing-data-client-side#testing-a-native-app-offline), and when you customized the message bubble in Exercise 5.1 with renderBubble(). What these functions have in common is that you passed them to <GiftedChat /> to customize some rendering behavior on Gifted Chat:

<GiftedChat

messages={messages}

renderBubble={renderBubble}

renderInputToolbar={renderInputToolbar}

onSend={messages => onSend(messages)}

renderActions={renderCustomActions}

user={{

\_id: userID,

name

}}

/>

You don’t have to use all of the default props provided by Gifted Chat, however, there’s nothing wrong with making use of some of them, which is what you did with wrapperStyle and iconTextStyle props in CustomActions.js.

Go ahead and extract the mentioned props in the CustomActions component:

const CustomActions = ({ wrapperStyle, iconTextStyle }) => {

The next step is to create a StyleSheet containing all style objects used in CustomActions.js’s UI. At the bottom of your CustomActions.js file, add the following:

const styles = StyleSheet.create({

container: {

width: 26,

height: 26,

marginLeft: 10,

marginBottom: 10,

},

wrapper: {

borderRadius: 13,

borderColor: '#b2b2b2',

borderWidth: 2,

flex: 1,

},

iconText: {

color: '#b2b2b2',

fontWeight: 'bold',

fontSize: 10,

backgroundColor: 'transparent',

textAlign: 'center',

},

});

Note  
Keep your import statements up to date. So far, you should have TouchableOpacity, Text, View, and StyleSheet imported from react-native into CustomActions.js.

Let’s talk about what onActionPress is supposed to do. This function displays an action menu that contains four options. This menu is called an “ActionSheet,” which is a component created using the [@expo/react-native-action-sheet](https://github.com/expo/react-native-action-sheet) module. GiftedChat already uses this module internally and has it prepared for you to use. However, you need to obtain it, initialize it with options/actions, and finally show it.

So, to summarize, when the user presses the action button CustomActions in the input field, onActionPress is called, which shows an ActionSheet that displays a set of defined actions. When the user selects one of these actions, a method for performing that action (e.g., picking an image from the gallery) is called.

ActionSheet  
While React Native provides ActionSheets for iOS, there’s no native ActionSheet UI element for Android. Using the Expo library @expo/react-native-action-sheet, you can ensure that similar-looking UI elements are displayed to both Android and iOS users (despite not being a perfect match).

Now, in the onActionPress function, define an array of strings to display in the ActionSheet. You’ll also display a cancel button, so you need to determine its position in the ActionSheet so that ActionSheet can close the view if the user presses “Cancel.”

const onActionPress = () => {

const options = ['Choose From Library', 'Take Picture', 'Send Location', 'Cancel'];

const cancelButtonIndex = options.length - 1;

}

Next, you need to fetch Gifted Chat's ActionSheet so that you can add these options to it. To fetch it, you need to use useActionSheet(), which is found in "@expo/react-native-action-sheet". This function's job is to fetch whatever ActionSheet is included inside the wrapper component. In your case, Gifted Chat **is** the wrapper component. Inside it, there’s an ActionSheet component that’s rendered.

Go ahead and add the following import to CustomActions.js:

import { useActionSheet } from '@expo/react-native-action-sheet';

Note  
You don’t need to install @expo/react-native-action-sheet because it will already be installed when you install the react-native-gifted-chat module.

Now, call useActionSheet() at the top of your CustomActions component. The function should return a reference to Gifted Chat’s ActionSheet:

const CustomActions = ({ wrapperStyle, iconTextStyle }) => {

const actionSheet = useActionSheet();

This reference (object) contains the showActionSheetWithOptions() function, which will initialize and show the ActionSheet. As you want to perform whichever action the user chooses from the ActionSheet, let’s write some console.log()s just to check that everything works as it should when one of the actions is clicked. If the user chooses “Take Picture” from the ActionSheet, your app will log “user wants to take a photo” in your developer tools console:

const onActionPress = () => {

const options = ['Choose From Library', 'Take Picture', 'Send Location', 'Cancel'];

const cancelButtonIndex = options.length - 1;

actionSheet.showActionSheetWithOptions(

{

options,

cancelButtonIndex,

},

async (buttonIndex) => {

switch (buttonIndex) {

case 0:

console.log('user wants to pick an image');

return;

case 1:

console.log('user wants to take a photo');

return;

case 2:

console.log('user wants to get their location');

default:

}

},

);

};

To let the user pick a picture, take a picture, and get their current location (when selecting each respective option) from ActionSheet, you have to call the functions pickImage(), takePhoto(), and getLocation(), rather than printing text to the console. So, it’s time to add these functions to CustomActions.js and call them in onActionPress():

const onActionPress = () => {

const options = ['Choose From Library', 'Take Picture', 'Send Location', 'Cancel'];

const cancelButtonIndex = options.length - 1;

actionSheet.showActionSheetWithOptions(

{

options,

cancelButtonIndex,

},

async (buttonIndex) => {

switch (buttonIndex) {

case 0:

pickImage();

return;

case 1:

takePhoto();

return;

case 2:

getLocation();

default:

}

},

);

};

Don’t forget about installing and importing the expo-location and expo-image-picker modules into CustomActions.js.

Great job! Now that you’ve implemented a button allowing users to *select* what they’d like to send (image or location, in this example), let’s look at how you would send a user’s images and location in Gifted Chat. Let’s start with how to send the location.

Code Check  
Take a look at our [ActionSheet demo Expo snack](https://snack.expo.dev/@cfcurriculum/actionsheet_demo" \t "_blank) to troubleshoot any problems in your code or to see if there’s anything you can do to optimize your code. Mainly, refer to the CustomActions.js file. This Expo Snack is just meant as a way to share code, not to run it. Currently, pressing any option will just log a message onto the console.

Sending Location and Rendering a Map

To render a map in a chat bubble, you need to create your own custom view—for this, GiftedChat has a prop called renderCustomView, which you can use to render the map.

As you might recall from [Exercise 5.2: Chat UIs & Accessibility](https://careerfoundry.com/en/course/full-stack-immersion/exercise/chat-ui-accessibility#working-with-gifted-chat), you can add custom key-value pairs to the Gifted Chat message object.

Gifted Chat has a predefined set of keys. However, you can also develop—and add—your own keys to use in the object as needed. However, the library ignores keys that it doesn’t know. Gifted Chat won’t use them unless you customize GiftedChat’s behavior, which you’ll do in this section.

Adding location data to the message object should look something like this:

{

\_id: 1,

createdAt: new Date(),

user: {

\_id: 2,

name: 'React Native',

avatar: 'https://placeimg.com/140/140/any',

},

location: {

latitude: 48.864601,

longitude: 2.398704,

},

}

Now, you need to create a function that lets you render a MapView if the message object contains location data. First, in Chat.js, add the prop renderCustomView to your GiftedChat component and let it call a function. You can name this function renderCustomView:

<GiftedChat

messages={messages}

renderBubble={renderBubble}

renderInputToolbar={renderInputToolbar}

onSend={messages => onSend(messages)}

renderActions={renderCustomActions}

renderCustomView={renderCustomView}

user={{

\_id: userID,

name

}}

/>

The renderCustomView function is where you’ll check if the currentMessage contains location data. If the answer is yes, it will return a MapView—just as you learned earlier!

const renderCustomView = (props) => {

const { currentMessage} = props;

if (currentMessage.location) {

return (

<MapView

style={{width: 150,

height: 100,

borderRadius: 13,

margin: 3}}

region={{

latitude: currentMessage.location.latitude,

longitude: currentMessage.location.longitude,

latitudeDelta: 0.0922,

longitudeDelta: 0.0421,

}}

/>

);

}

return null;

}

Make sure to import MapView in Chat.js:

import MapView from 'react-native-maps';

Tip! Rendering Multiple CustomViews  
If you need to render multiple custom views (bubbles), you have to add a switch case or if-else statement in the function passed to Gifted Chat’s renderCustomView prop.

For example:

const renderCustomViews = (props) => {

const { currentMessage} = props;

if (currentMessage.location) // render a map

if (currentMessage.3dModel) // render a small 3d model viewport

// ..etc

}

You’ve now learned how to render the map in a message bubble. It’s time to explore how to carry out the actual process of sending the location. You’ll need access to the onSend() method of Gifted Chat in CustomActions.js. Fortunately, Gifted Chat passes the onSend method as a prop to the function assigned to the renderCustomActions prop.

Now, let’s bring up what the renderCustomActions prop has assigned to it, which is the function renderCustomActions (which you declared in the last section in Chat.js):

const renderCustomActions = (props) => {

return <CustomActions {...props} />;

};

As you can see, you passed the props to the CustomActions component. Remember, this props object contains Gifted Chat’s onSend() method. As a result, CustomActions will have access to onSend(), which makes life easier!

All that’s left in order to send the location is to extract onSend from the props of CustomActions in its file, then call onSend() while passing an object to it that contains the location details needed to render the map view.

Open CustomActions.js and extract onSend from the component’s props object argument:

const CustomActions = ({ wrapperStyle, iconTextStyle, onSend }) => {

Next, call onSend in getLocation() as follows:

const getLocation = async () => {

let permissions = await Location.requestForegroundPermissionsAsync();

if (permissions?.granted) {

const location = await Location.getCurrentPositionAsync({});

if (location) {

onSend({

location: {

longitude: location.coords.longitude,

latitude: location.coords.latitude,

},

});

} else Alert.alert("Error occurred while fetching location");

} else Alert.alert("Permissions haven't been granted.");

}

To summarize so far:

* You’re sending a message that only contains the location property (though there will be other properties added by default, such as createdAt, \_id, and user);
* The object assigned to the location property sent in onSend contains all the data necessary for renderCustomView to render the MapView in a message bubble.

Once implemented in your chat app, test your app and try to send your location. The map view should look something like the following image (Figure 10):

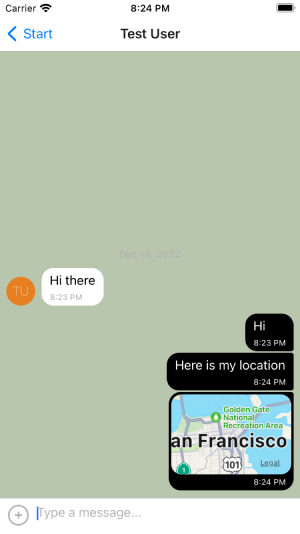


Figure 10

Code Check  
You can find sample code for sending a location and rendering a map in our [location sharing demo Expo Snack](https://snack.expo.dev/@cfcurriculum/location_sharing_demo). You can use it to compare your own code and to try out the app’s functionality. Remember to replace the Firebase app config with your own in App.js to make it work. To run it, install dependencies by running npm install and then expo start.

Sending Images

To send an image in Gifted Chat, include its path in the message object, like so:

{

\_id: 1,

text: 'My message',

createdAt: new Date(Date.UTC(2016, 5, 11, 17, 20, 0)),

user: {

\_id: 2,

name: 'React Native',

avatar: 'https://facebook.github.io/react-native/img/header\_logo.png',

},

image: 'https://facebook.github.io/react-native/img/header\_logo.png',

}

Gifted Chat will render the image in the chat bubble!

Next, think about where you want to store your users’ images. You need to ensure that users have constant access to any images they’ve ever sent. As such, it makes sense to remotely store the sent images. For that, you’ll be diving back into Google Firebase.

Storing Images in Google Firebase

You need somewhere to store images that get sent via your chat app. Chat participants won’t be able to see these images without a storage place for them. The solution is likely already familiar to you: storing the images on a remote server (the cloud). Storing images on a remote server allows all mobile devices to connect to the server to fetch the sent images.

To store and send images in your native chat app, you’ll be using [Cloud Storage for Firebase](https://firebase.google.com/products/storage/). Specifically designed to store and serve photos and videos, it’s a great choice for this project as you’re already using Google Firestore to store your chat messages, which means all your chat and image files will be kept in one place!

If you sign up for the Firebase Blaze plan, you can use up to [5GB storage per month for free](https://firebase.google.com/pricing/), which should be plenty for your chat app project. If you find yourself approaching the limit, you can always delete some images to free up some space.

Note  
To sign up for the Blaze plan, you will need to input your own personal credit/bank card details. But please note that you will not be charged as long as you do not exceed the 5GB limit. 5GB should definitely be more than enough for your current use case.

To set up Firebase Storage, head to the Google Firebase website homepage. You should be taken to the console (make sure you’re logged in), from which all your different projects will be shown on a dashboard. Select your chat app from the projects displayed in the dashboard. To create a new storage space, click **Build** and then **Storage** from the left-hand menu. Your page should look something like this:

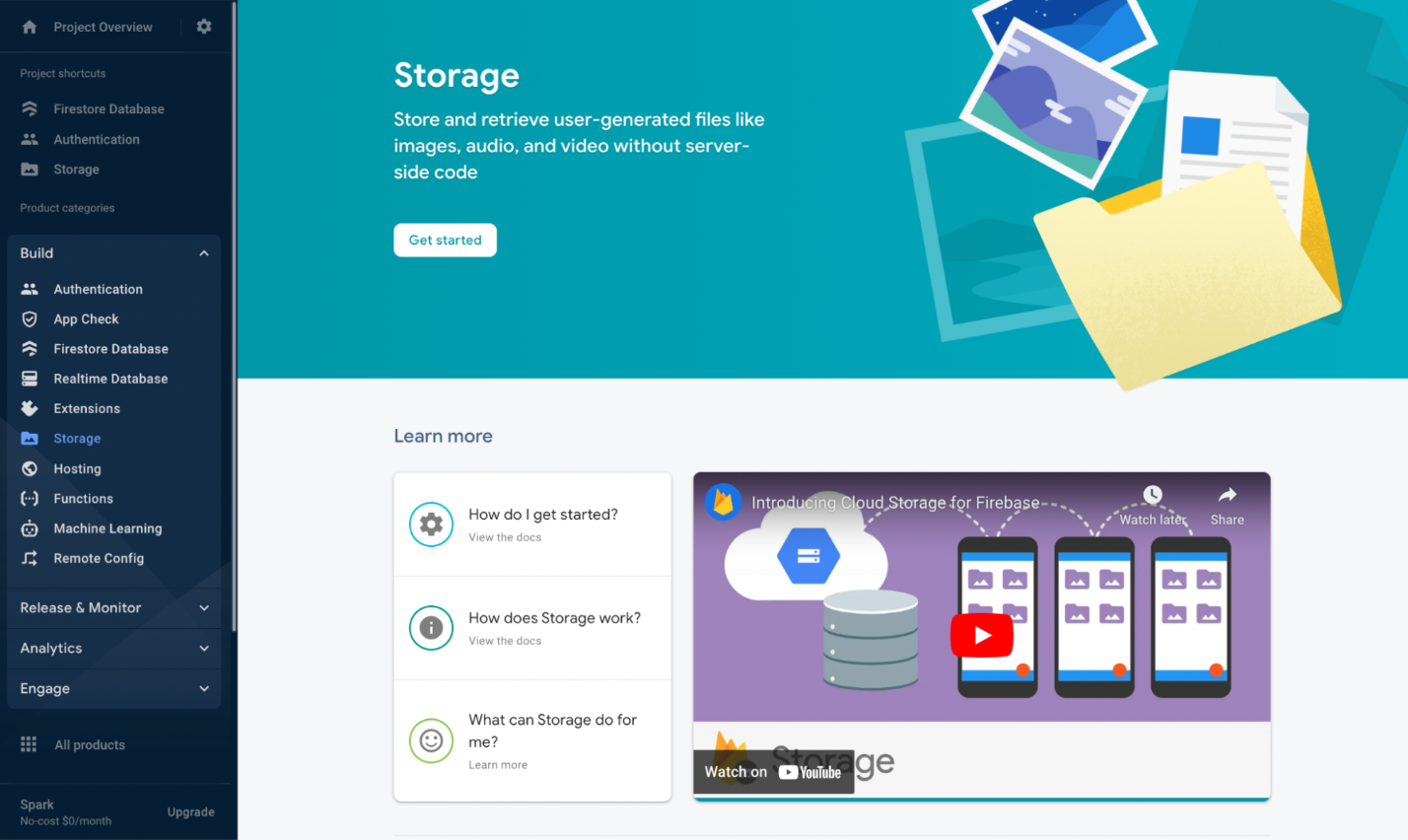


Figure 11

Go ahead and click the **Get started** button. A popup will open that asks you to set your cloud storage. Keep everything on default and press **Next**, then **Done**. The cloud storage dashboard (Figure 12) should open.

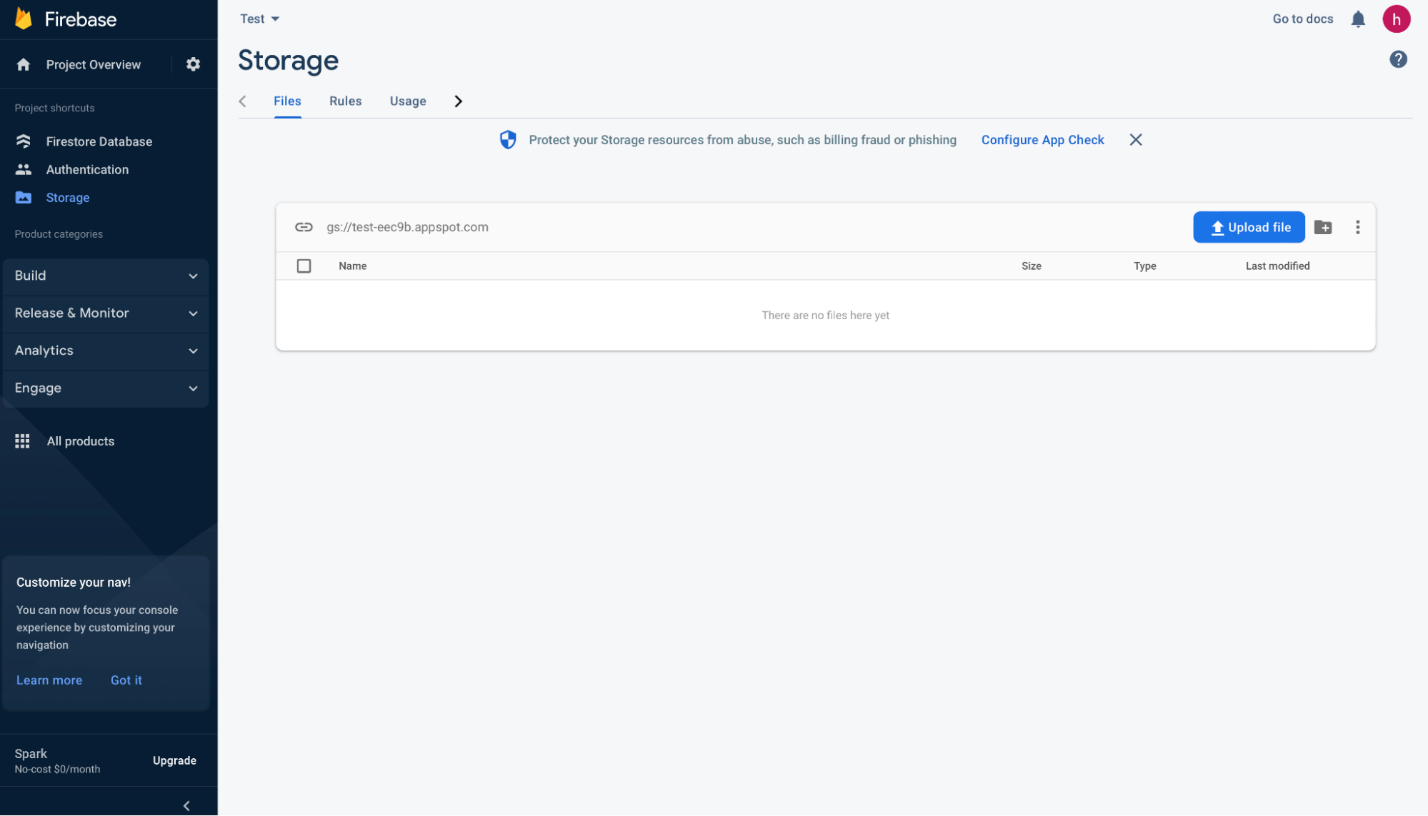


Figure 12

Now you need to allow uploading and downloading files to and from the storage, from whichever device connects to your Firebase Storage. To do so, go to the **Rules** tab, change “allow read, write: if false;” to “allow read, write: if true”, and click **Publish**. This is similar to what you did for your Firestore Database.

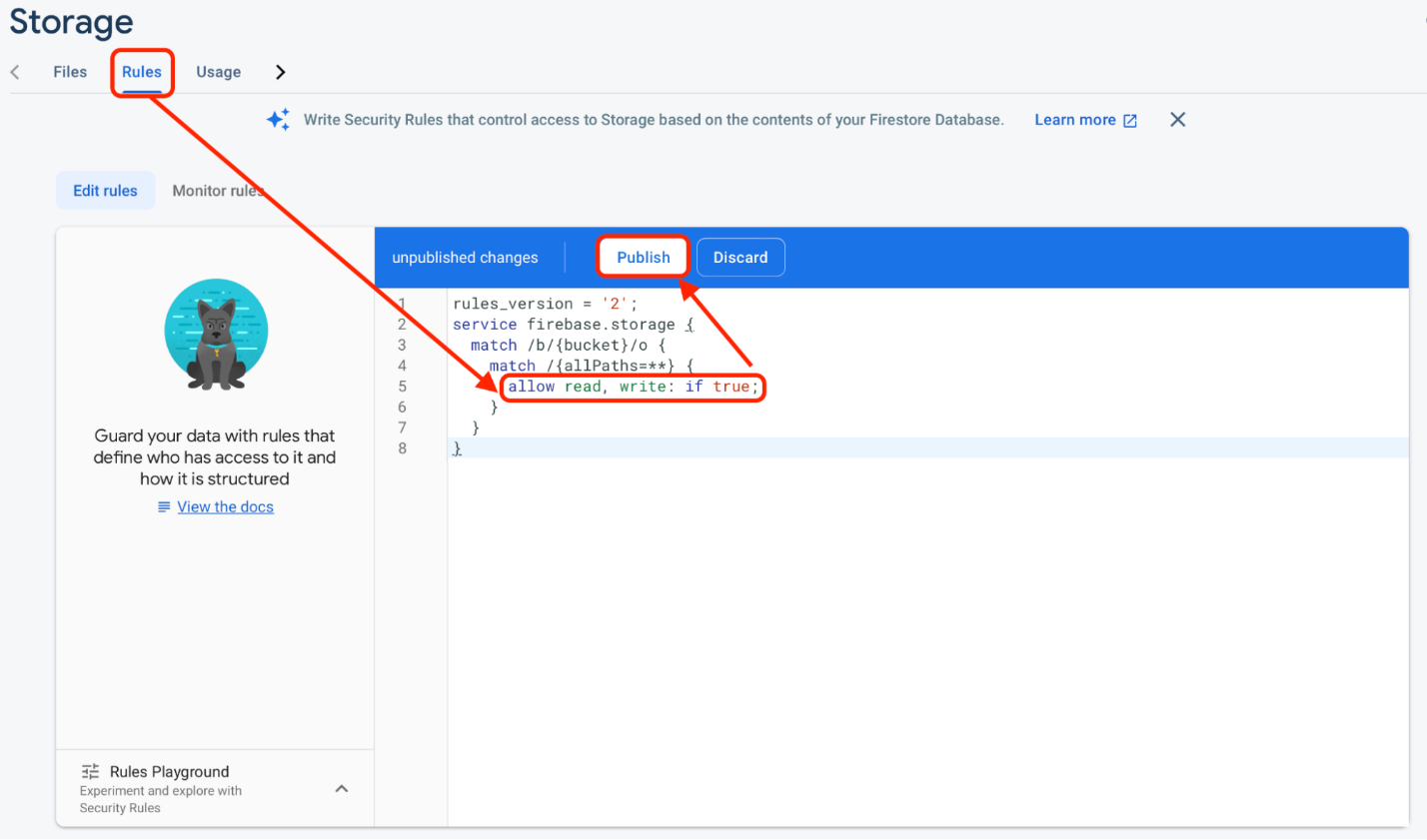


Figure 13

Back in the **File** tab, upload a file of your choice to see how the page looks with some actual content. You don’t need to configure anything else because everything you need to work with Firebase Cloud Storage is already built into the Firestore library that you set up in Exercise 5.3: Real-Time Apps & Data Storage.

Before you upload an image to Firebase Storage programmatically (in code), you’ll need to convert the file into either a [Blob](https://developer.mozilla.org/en-US/docs/Web/API/Blob), [Uint8Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array), or [Base64 string](https://developer.mozilla.org/en-US/docs/Glossary/Base64).

In the running example, you’ll convert the image file to a **Blob** (a “binary large object”). A blog is a collection of binary data that gets stored in a database. Images and audio objects are typically saved in databases as blobs. However, Firebase Storage also accepts other data formats. To learn more about which ones these are, check out this page from the [Firebase documentation](https://firebase.google.com/docs/storage/web/upload-files).

Suppose a user picks an image from their device’s library, and you now have the path to that file. To fetch content from the given URI, you use the fetch method (note that the fetch method expects a URL, but URLs are simply a subset of URI). You then need to convert this content into a blob. For this, you use Expo’s blob() function. The conversion to a blog needs to happen after you’ve obtained the URI of the picked image (or the taken photo, in that scenario). So, in the pickImage() function, you need to place it after you have access to result.assets[0].uri:

const pickImage = async () => {

let permissions = await ImagePicker.requestMediaLibraryPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchImageLibraryAsync();

if (!result.canceled) {

const imageURI = result.assets[0].uri;

const response = await fetch(imageURI);

const blob = await response.blob();

}

else Alert.alert("Permissions haven't been granted.");

}

}

To upload the blob into Firebase Storage, you must first prepare a storage location reference on the Firebase Storage Cloud. However, to create such a reference, you need to initialize a handler to your Firebase Storage. This is similar to what you did with the Firestore Database handler, db, with const db = getFirestore(app); in App.js. However, the difference is that you call getStorage() instead of getFirestore. getStorage is imported from firebase/storage. So, go ahead and open App.js and import it:

import { getStorage } from "firebase/storage";

Next, initialize the storage handler inside the App component. Place the line of code after const db = getFirestore(app); so that they’re grouped together.

const storage = getStorage(app);

Now you can pass storage to the Chat component as a prop, similar to how you passed db. Go ahead and update the “Chat” Stack.Screen:

<Stack.Screen

name="Chat"

>

{props => <Chat

isConnected={connectionStatus.isConnected}

db={db}

storage={storage}

{...props}

/>}

</Stack.Screen>

Now, in Chat.js, you can pass the same storage to CustomActions so that you can use it in the location where picking images and taking photos takes place.

Update the renderCustomActions function definition to the following:

const renderCustomActions = (props) => {

return <CustomActions storage={storage} {...props} />;

};

Finally, you have to access the Firebase storage handler. Go ahead and extract the prop in CustomActions.js:

const CustomActions = ({ wrapperStyle, iconTextStyle, onSend, storage }) => {

As mentioned earlier, to upload a file, you have to prepare a new reference for it on the Storage Cloud. Think of this process as if you’re creating an address for the location the file will be placed once it’s uploaded. To create one, write the following:

const newUploadRef = ref(storage, 'image123');

Let’s break this up:

* ref() is a function imported from firebase/storage. Make sure to import it in CustomActions.js: - Import { ref } from 'firebase/storage';;
* Regarding storage passed to ref(), it’s the Firebase Storage handler you passed from Chat.js.
* Regarding the 'image123' string passed as the second argument to ref(), it’s the reference string that you can use as an identifier to retrieve/download the file once it’s uploaded.

You can upload an image file blob using the Firebase Storage method uploadBytes(), as follows:

const pickImage = async () => {

let permissions = await ImagePicker.requestMediaLibraryPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchImageLibraryAsync();

if (!result.canceled) {

const imageURI = result.assets[0].uri;

const response = await fetch(imageURI);

const blob = await response.blob();

const newUploadRef = ref(storage, 'image123');

uploadBytes(newUploadRef, blob).then(async (snapshot) => {

console.log('File has been uploaded successfully');

})

}

else Alert.alert("Permissions haven't been granted.");

}

}

Here are the points of interest from this code snippet:

* uploadBytes() is a method imported from the firebase/storage module (do so in CustomActions.js, if you haven’t already). This is the function that you’ll use to upload the file.
* The first argument, newUploadRef, passed to uploadBytes() is the reference that the file will be uploaded to.
* The second argument, blob, is the blob of the image file you want to upload.

Test your app and try to pick an image from the library. You should see that it’s successfully uploaded to Firebase Cloud Storage, titled “image123,” as defined in the code:

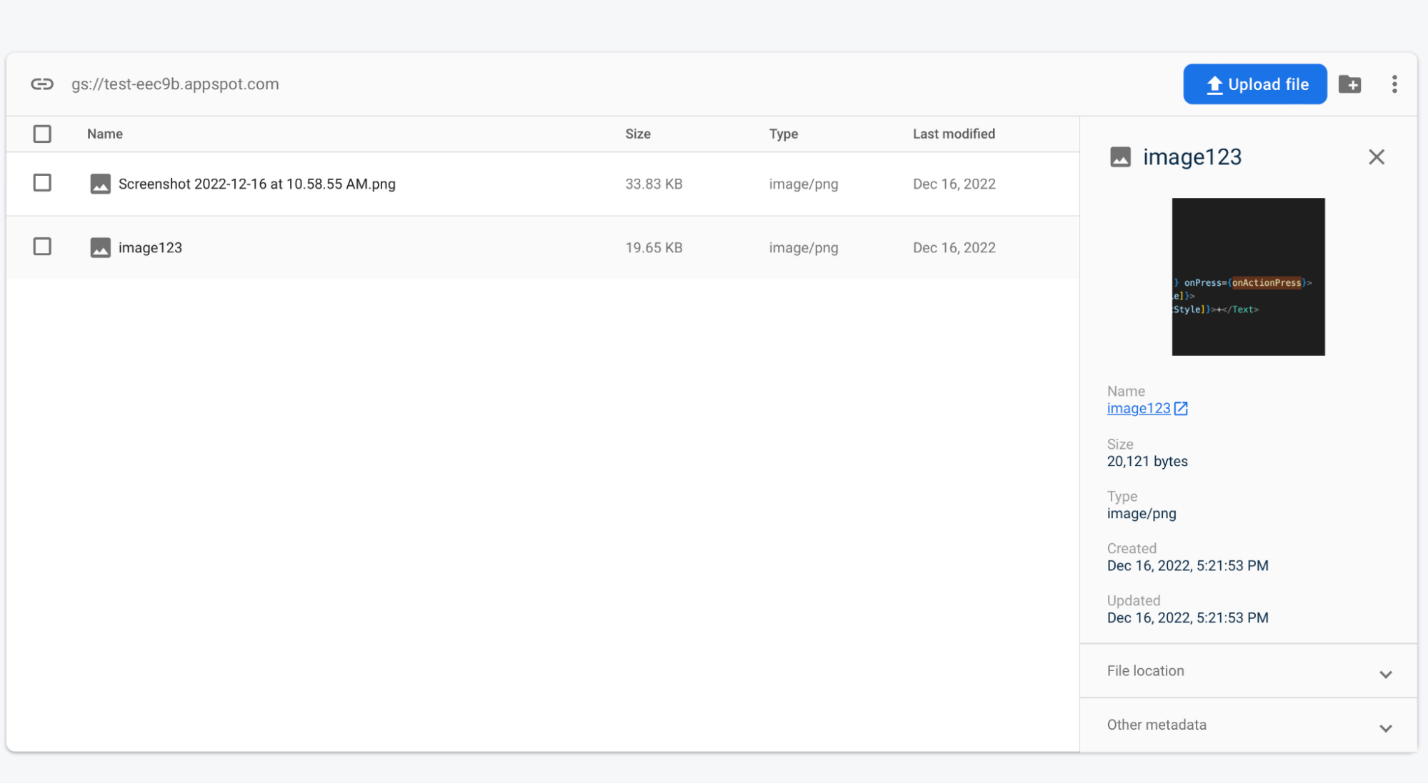


Figure 14

If you try to pick another file to upload, it will be uploaded. However, this new file will replace the image uploaded previously (in this case, “|image123”). This is because the new image will also have the same reference string—image123. To upload multiple images, you need to use a unique reference string each time a new file is uploaded. For this, you’ll need to add another function in the CustomActions component:

const generateReference = (uri) => {

const timeStamp = (new Date()).getTime();

const imageName = uri.split("/")[uri.split("/").length - 1];

return `${userID}-${timeStamp}-${imageName}`;

}

The function has one argument that represents the picked image’s URI. The function combines multiple strings to produce a string that can be used as a unique reference for the image to be uploaded. The strings combined are as follows:

The logged-in user ID (you can pass it from the Chat component to CustomActions): Make sure to extract it in CustomActions.js: const CustomActions = ({ wrapperStyle, iconTextStyle, storage, onSend, userID }) => {. Note that this is assuming that you’ve named it userID.  
The current timestamp: This is converted to milliseconds (will be concatenated as a string).  
The original image file name.

So, update pickImage() to use generateReference(), then use what’s returned in place of "image123" in const newUploadRef = ref(storage, "image123");:

const pickImage = async () => {

let permissions = await ImagePicker.requestMediaLibraryPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchImageLibraryAsync();

if (!result.canceled) {

const imageURI = result.assets[0].uri;

const uniqueRefString = generateReference(imageURI);

const response = await fetch(imageURI);

const blob = await response.blob();

const newUploadRef = ref(storage, uniqueRefString);

uploadBytes(newUploadRef, blob).then(async (snapshot) => {

console.log('File has been uploaded successfully');

})

}

else Alert.alert("Permissions haven't been granted.");

}

}

The next thing to do is to actually send the image as a message so that it gets rendered in a message bubble inside GiftedChat. You’ll need to implement two things in pickImage():

1. Get the remote URL of the image you’ve just uploaded.
2. Call the onSend() prop in CustomActions, inside the pickImage() function.

The first step is done by calling the async Firebase storage method, getDownloadURL(), and passing snapshot.ref to it. Add the following in the callback function of then() attached to uploadBytes():

uploadBytes(newUploadRef, blob).then(async (snapshot) => {

console.log('file has been uploaded');

const imageURL = await getDownloadURL(snapshot.ref)

})

Note  
At this point, you should have three methods imported from firebase/storage in CustomActions.js: import { ref, uploadBytes, getDownloadURL } from "firebase/storage";

As for using Gifted Chat’s onSend() method, you already have it in the props of the CustomActions component due to the prep work you did earlier. So, go ahead and call onSend() right after obtaining the URL of the uploaded image:

uploadBytes(newUploadRef, blob).then(async (snapshot) => {

console.log('file has been uploaded');

const imageURL = await getDownloadURL(snapshot.ref)

onSend({ image: imageURL })

});

Now, when you pick an image from the library, not only will it be uploaded to the Firebase Cloud Storage, but it will also be sent as a message, so it should appear in the Chat screen:



Figure 15

Before wrapping up the Exercise, it’s time for some refactoring! You can move the code for uploading and sending the image as a message to a standalone function so that you can call it inside takePhoto() as well:

const uploadAndSendImage = async (imageURI) => {

const uniqueRefString = generateReference(imageURI);

const newUploadRef = ref(storage, uniqueRefString);

const response = await fetch(imageURI);

const blob = await response.blob();

uploadBytes(newUploadRef, blob).then(async (snapshot) => {

const imageURL = await getDownloadURL(snapshot.ref)

onSend({ image: imageURL })

});

}

const pickImage = async () => {

let permissions = await ImagePicker.requestMediaLibraryPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchImageLibraryAsync();

if (!result.canceled) await uploadAndSendImage(result.assets[0].uri);

else Alert.alert("Permissions haven't been granted.");

}

}

const takePhoto = async () => {

let permissions = await ImagePicker.requestCameraPermissionsAsync();

if (permissions?.granted) {

let result = await ImagePicker.launchCameraAsync();

if (!result.canceled) await uploadAndSendImage(result.assets[0].uri);

else Alert.alert("Permissions haven't been granted.");

}

}

Uploading Images ExpoSnack  
If you’re still having trouble converting your image files into blobs, take a look at the two functions in this [communications features complete demo Expo Snack](https://snack.expo.dev/@cfcurriculum/communication_features_complete_demo).

To test the code, you’ll have to download the code, then navigate to the downloaded folder within the terminal to run npm install to install dependencies listed in package.json. Also, make sure to use your Firebase configurations in App.js. You’ll be ready to go and test the app by running expo start. Or, if you just want to compare your code again, you can just use the Expo Snack.

Communication Features: Ethical Considerations

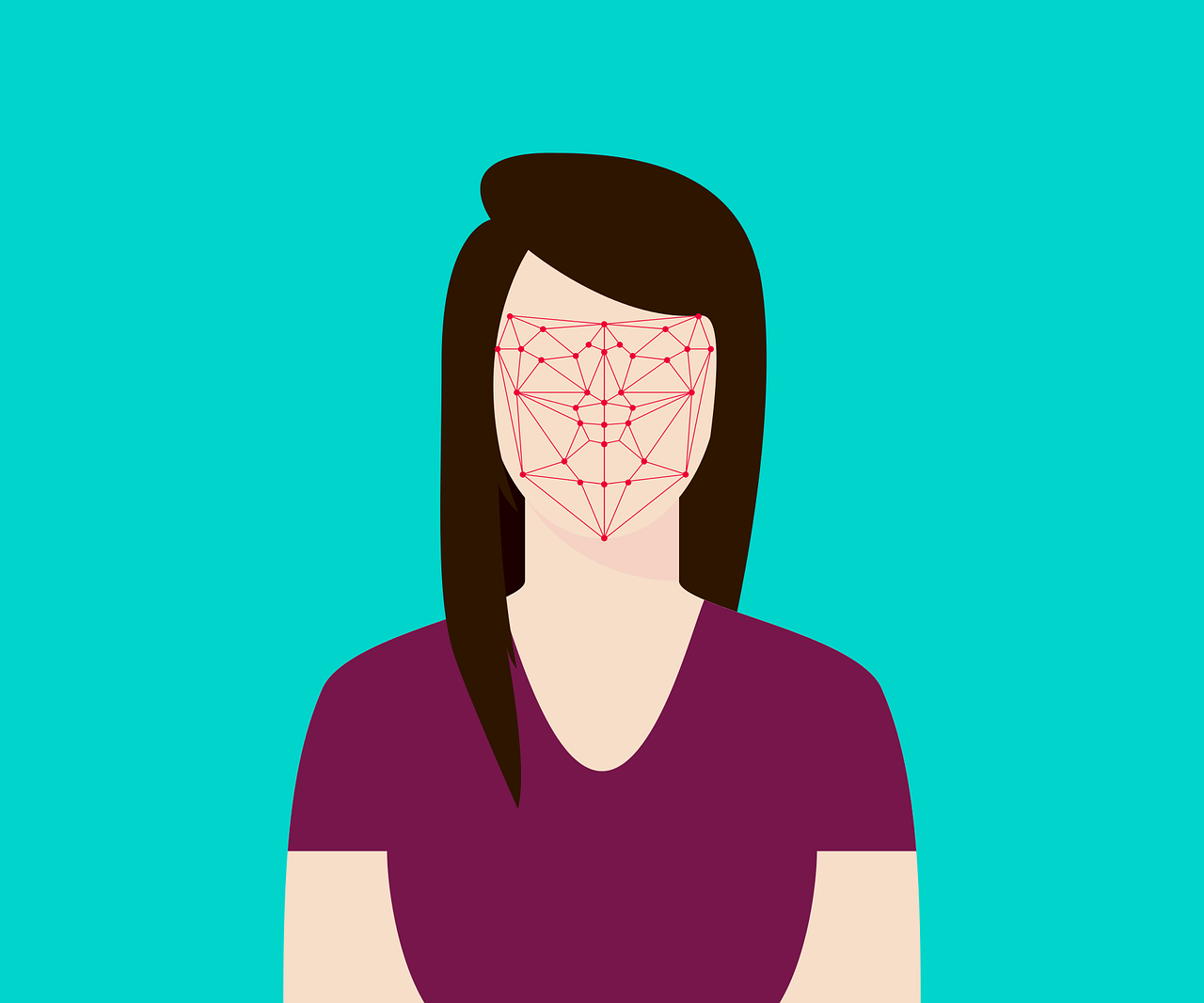


Figure 16

Whether you’re developing a website, web app, or native mobile app, you must always think about how you’re handling users’ data and respect their privacy—especially when working with a device’s camera and microphone. Many users distrust products that use communication features, and it’s part of a designer or developer’s job to create apps that make the user feel secure.

When users give your app access to their camera (or audio), you need to ensure that you don’t breach their trust. For example, don’t save the image or video (or audio) files if it’s not necessary for your app to do so.

At the beginning of this Exercise, you learned about the WebSpeech API as a way of implementing speech-to-text functionality in a web app and viewed [Google’s Web Speech API demo](https://www.google.com/intl/en/chrome/demos/speech.html) to see how it works. Although there’s no need for this demo to save any of the audio it records, it’s not entirely clear whether or not this is happening. Telling users what happens to their recordings would be just one way to achieve greater transparency.

Furthermore, it’s important to make it clear to the user when the microphone (or camera) is enabled or disabled. If this is ambiguous, as with the Web Speech demo, it can make users feel uncomfortable. You can remove this ambiguity by indicating to the user when the camera or microphone is active and stating what your app is doing when it’s active or open.

Many users are more trusting of features on their mobile devices. How many people do you know who tape over their laptop camera but not their smartphone’s front and back camera? Users are much more accustomed to mobile apps using the phone’s camera and microphone. Think about all the apps that depend on the user’s camera (Instagram, Snapchat, TikTok, and the like).

You’ll often find that people are more accepting of communication features on mobile devices; however, this doesn’t mean that you shouldn’t take an ethical approach to storing their data. It’s important to always communicate what the app has access to, ask for permission to use certain features, and clearly indicate when certain features are enabled or disabled.

When working with components like the camera in combination with intelligent algorithms or AI, it’s also important to make sure that the app you’re building is usable for everyone. You’ve probably heard about some of the [problems around facial recognition software](https://www.theverge.com/2019/1/25/18197137/amazon-rekognition-facial-recognition-bias-race-gender), which often can’t detect people with darker skin or mistakes women for men. While programmers aren’t intentionally writing biased algorithms and researchers are working hard to solve these issues, the fact remains that if someone can’t use an app or an important feature of an app because the technology behind it is biased, it’s not inclusive.

In an age where digital technology features so prominently, developers must think about what they need to do from an ethical and moral standpoint to build excellent products for the future.

Summary

Well done—that was a lot to take in! You’ve learned how to access the camera, microphone, and geolocation of your users’ devices from within a native mobile app using Expo. What’s more, you’re now able to integrate additional communication features besides text into your native chat app, and you know how to store media files in Google Cloud Storage. Finally, you examined some of the accessibility, ethical, and UX design considerations surrounding mobile applications, along with why they’re so important.

In the task, you’re going to recap what you’ve learned in this Exercise by integrating some new communication features into your Gifted Chat app.