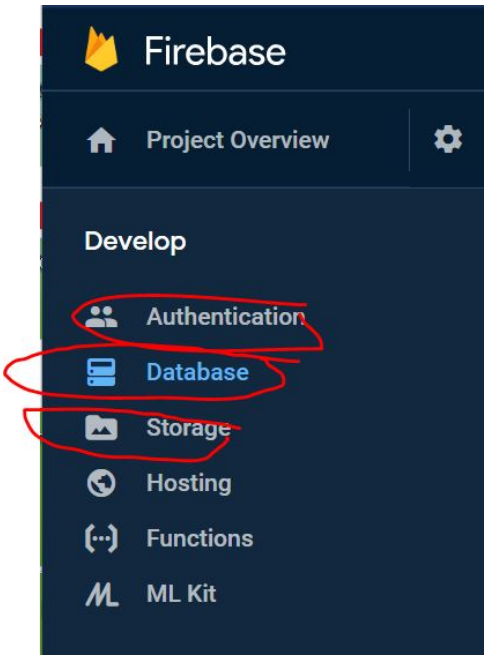
Android programming: Flipped Classroom: Photolist

You will need to create a new project in firebase (at the console) and connect your FC with it. This is done the same way you did it for the authentication FC, culminating in your downloading the `google-services.json` file and putting it in the app directory. You will not need to change the gradle file for this project.

As you have done previously, enable authentication for email/password and create a user called "fake@example.com" with password "123456". You will need to activate the Database (Firestore) and Storage by clicking on them in the web interface, and telling them to create. See the image below, though these entries are now hidden in some submenus.

While we provide a logout button for you to test authentication, for this FC you can just have a single user.

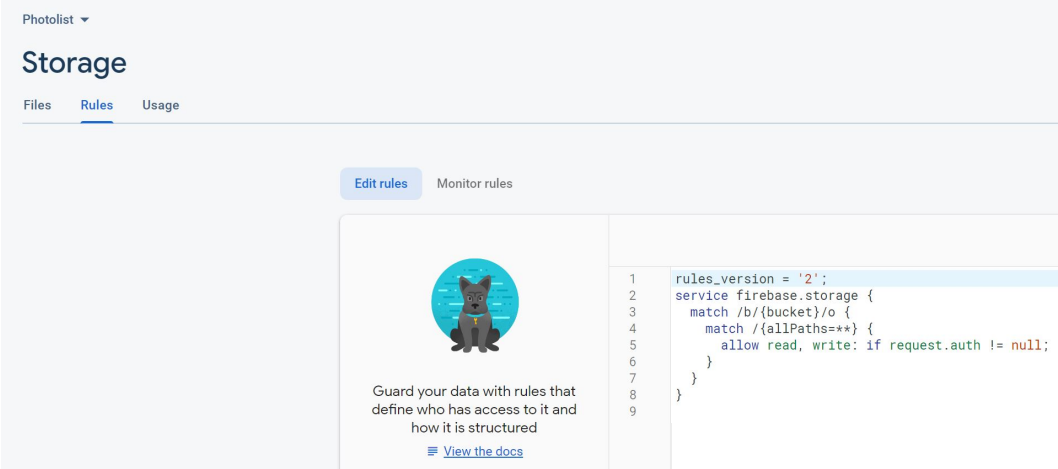


We are using three major cloud subsystems: authentication, a database, and file storage. Authentication is mostly stand alone. But for this lab and in general, we will see that we store related data items in the database and in storage. The database is good for structured data that we want to search. But it is bad for large binary blobs like pictures and video. So in this FC we store pictures in storage and their name and other metadata in a database. This is a common idiom.

Using the firebase console, remember to edit the rules to access `Database` and `Storage`. You want a rule that at heart looks like this.

```
allow read, write:  if request.auth != null;
```

Here is a picture from the Storage rules section, but this rule will work for both firestore and storage.



Most of what you need to do appears in this quick start guide. It also makes reference to some things, like modifying your gradle file, that you don't need to do.
`https://firebase.google.com/docs/firestore/quickstart`

Firestore is a document-based database, so it does not support the full relational model of SQLite and it does not support SQL. But it supports a large subset of SQL's functionality using a different syntax. The root of our database is a collection. The collection contains documents, which are key/value pairs. The value of a pair in a document can be a collection. So we strictly alternate between collections and documents as we traverse down our firestore storage hierarchy. If this is confusing, here is a document which also has a video.
~~`https://firebase.google.com/docs/firestore/data-model`~~

Here is the data model for this FC. We have a collection called `allPhotos` (though it probably should be called `allPhotoMeta`. In it are `PhotoMeta` documents. These documents hold metadata about photos. They do not hold the photo data itself. The name of the document is equal to the value of its `firestoreID` field, which is set by the server. It comes in handy.

In Firebase storage, we will have a Folder called "images" and inside it we store files of type `image/jpg`. You should have a passing familiarity with MIME types (now called Media types `https://en.wikipedia.org/wiki/Media_type`).

Take a look in `TakePictureWrapper` because that code generates a random universally unique identifier (UUID). A UUID contains so many random bits that the effective probability of collision is small enough to ignore. In the `takePicture` method, we store the image's UUID in the view model.

We create photos using UUIDs so that different users can create and store photos at the same time without interferring with each other and without having to communicate back and forth to the server (like asking, hey server, give me a unique name for my photo, then another message saying here are the bytes for that photo).

A photo metadata object has a firebaseID, which is a unique identifier provided by the database. The photo is stored as an object named by UUID, which is unique by construction. Finally, it has a title, which is a string that does not need to be unique.

See the video demo. You sign in and the app shows you a list of photos. If you haven't taken any photos, use the camera button to take a few. But you have to add the photo title before you take the picture.

Once you have some photos, you should see them displayed with a thumbnail, the title and the size in bytes. The list will be sorted by either title or size and either ascending or descending. Sorting by title ascending is the default. Whatever column controls the sort, a background color of yellow indicates ascending and one of red indicates descending.

You can swipe left on a row to delete it.

Any time you create a new photo, or delete one, or change the sort criteria, the view should update.

We take pictures by starting an activity (see the cameraLauncher in HomeFragment). This is pretty cool since we don't have to write the code to control the camera. But starting the activity is a little awkward and we do have to remember the picture's UUID in the view model. We need to put it in the view model because we are starting an activity to take the picture and that can cause the runtime to kill our view.

To work out the permissions to create the picture file, we need entries in the manifest file and in `file_paths.xml`. The code in TakePictureWrapper deals with permissions so you don't have to.

- ~~**AndroidManifest.xml** We g~~ give you what you need. Please have a look in this file though. It defines a section for a `androidx.core.content.FileProvider` and it uses the file `file_paths.xml` in the `res/xml` directory. You might need something like this if you want photo capability in your project.
- ~~**glide/AppGlideModule.kt** All good. Bu~~t I am fixing the size of the thumbnail image in raw pixels. Yuck! But I tried more reasonable alternative and failed, so this is good enough for now. Maybe someone will suggest an improvement.
- ~~**model/PhotoMeta.kt** All good~~. This defines our model. The database stores these metadata records for us, with the server filling in the time stamp when the record is created. The database provides us with a unique identifier when we want to write the object, so we get that ID and assign it to our record before writing. You'll see.
- **view/HomeFragment.kt** In `onViewCreated` you should hook up some onclick listeners and observe some live data. This is the view.

  When the camera button is pressed, check the EditText box. If the title is empty, complain via Toast or Snackbar, "You must title picture".

  I provide `mainActivity.progressBarOn` and `mainActivity.progressBarOff`. These are not super useful in this FC because the network delays are short. But they are useful in FireNote and you can get some practice using them here. The idea is that progressBarOn sets the progress bar spinning and progressBarOff turns it off. It is a lot like the SwipeRefresh indicator, but just an independent part of the layout, so you can activate it when you want. You should activate/deactivate when changing the sort order. See MainActivity for an example.

- ~~**view/PhotoMetaAdapter.kt** All good. Simple and I hope clear.~~

- ~~**view/TakePictureWrapper.kt** All good.~~ This has most of the logic for taking pictures. You should be able to gloss over most of the details, but pay attention to the uuid, since you will need this when you upload the photo file to firebase storage.

  We call `TakePictureWrapper.fileNameToFile` in `MainViewModel.pictureSuccess`.

- **AuthUser.kt** This should be familiar from the Firebase auth FC. We have modified it to automatically log back in if we get logged out by the server. It also automatically logs in new users. The class listens for changes in authentication status from firebase, and it insulates the rest of the codebase from having to know about Firebase classes.

  You need to write the code to respond in onAuthStateChanged. Lucky for you most of the work is done by a helper function that we provide.

  Also, you need to create and launch the signInIntent in login, just like you did in the firebase auth FC.

- **MainActivity.kt** Much of this code should be familiar to you at this point.

  You do need to complete `onStart`. We can't create the AuthUser object in onCreate, because it will not be valid until the lifecycle that AuthUser observes has completed `onCreate`. So we create the AuthUser object in `onStart` and let it observe MainActivity's lifecycle. (It needs to observe a lifecycle because it might have to launch the sign in activity). AuthUser uses live data to communicate about changes to the current user, so we observe that live data in MainActivity. The MainViewModel also wants to know about the current authenticated user, but view models are not good observers so we have an explicit function to update it.

- **MainViewModel.kt** There are three routines you need to help out.

  - `sortInfoClick`. The user has changed the sort criteria. This calls for action. Recall that resultListener is a function passed in by the caller that takes no arguments and returns no values. You just need to pass it along to `fetchPhotoMeta`, but it is there so you have a way to turn the progress bar off.
  - `removePhotoAt`. The user wants to remove a photo at a given index. What does that require? At least one call to storage and one call to dbHelp.
  - `pictureSuccess`. I wanted to make you write this so that you might make the same mistake that I did and then fix it. Or just read and understand the big comment for this function. After you have used pictureUUID and pictureNameByUser you should set those to a "bad" value like the empty string to avoid accidentally reusing an old value.

- **Storage.kt** Two pretty quick ones that should give you the important insights for how to interact with Firebase storage. Pay attention to the documentation URL above the functions.

  - `uploadImage`. Just get the upload task defined and we will take it from there. Set the MIME type.
  - `deleteImage`. Explains itself.

- **ViewModelDBHelper.kt** Once again, please pay attention to the documentation URL above each function. Read `limitAndGet`. `resultListener` is a caller-provided function that

expects a list of photo metadata. Think about why you would want that when calling the public functions in this file.

We provide the list using a hot one-liner that has: mapNotNull, toObject, and ::class.java. That is programming language power in action.

- `dbFetchPhotoMeta`. Fetch! Call `limitAndGet` at the end.

  One quick note here. If this were a real app, we probably wouldn't refetch from the server every time the user wants to sort their display differently. We should just cache the records and sort locally. But I want y'all to get experience with these firestore/storage calls.
- `createPhotoMeta`. Pretty straightforward given the documentation.
- `removePhotoMeta`. Pretty straightforward given the documentation. But make sure you are deleting the correct entry.

Rules and submission Recall the rules for flipped classrooms: do the work alone or with a partner that is unique for the semester. You can find the assignment information on Canvas.

Only one student needs to submit this code, but both students may do so if they wish. This assignment should be submitted through github (sorry, partners can't share the repository). Include a README at the top directory level that contains both student's EIDs.