1. 图片保存
    A. **使用camera拍照后把图片上传到firestore**
    B. **获得firestore保存图片到url并保存在firebase中**
    C. **render时从firebase下载图片**
2. **推送**
    A. **local/push推送的区别(是否需要后端)**
    B. **使用推送流程**
        a. **获得permission**
        b. **scheduleNotificationAsync**
        c. **设置content和trigger**

13. Storage + Firestore, Notifications

Creating a Photo Gallery App with Storage and Firestore

For our first exercise today, we'll put together a bunch of stuff from the past few weeks: Firestore, Firebase Auth, Cameras, and Firebase Storage.

You should've cloned `week13Gallery` and gotten it ready to go. Right out of the box, it should allow you to:

1. Log in or create a new user
2. Subscribe to `onSnapshot` for the `'users'` collection
3. Correctly display the current user's display name
4. Allow the current user to sign out
5. Allow the current user to take a picture once they are signed in

All of these are things you have seen before, but perhaps put together a bit differently. Let's walk through the code as a quick review.

First, something that isn't a review! A way to suppress warnings that you really don't need to see. In this case we're suppressing the "AsyncStorage" warning that we keep getting because the Firebase SDK is using an out-of-date dependency. Not our problem (hopefully)!

```
import { LogBox } from "react-native";
LogBox.ignoreLogs(["AsyncStorage"]);
```

Review 1: Auth State Listening

in `LoginScreen`, we listen for auth state changes. These will happen when a user logs in or creates an account, including when a user is automatically logged in thanks to "auth state persistence."

```
  useEffect(()=>{
    onAuthStateChanged(auth, (authUser) => {
      if (authUser) {
        dataModel.initOnAuth();
        navigation.navigate('Main', {currentAuthUserId: authUs
er.uid});
      } else {
        navigation.navigate('Login');
      }
    });
  }, []);
```

For the above to work, we need to implement `DataModel.initOnAuth()`, which sets up an `onSnapshot()` listener that monitors the `'users'` collection. We're mainly doing this so that the `MainScreen` won't ask for the current user's `displayName` until we're sure we can give it to them.

```
  initOnAuth() {
    if (this.userSnapshotUnsub) {
      this.userSnapshotUnsub();
    }
    this.userSnapshotUnsub = onSnapshot(collection(db, 'user
s'), qSnap => {
      let updatedUsers = [];
      qSnap.forEach(docSnap => {
        let user = docSnap.data();
        user.key = docSnap.id;
        updatedUsers.push(user);
      });
      this.users = updatedUsers;
      this.notifyUserSnapshotListeners();
    });
  }
```

```
}
```

Since we’re setting some stuff up on login, like the `onSnapshot()` listener, we need to tear it down elegantly on logout. We’ll call a method on `DataModel` to clean up before we actually log out when the user clicks “Sign Out” in the `MainScreen`.

```
      <Button
        title='Sign out'
        onPress={()=> {
          dataModel.disconnectOnSignout();
          signOut(auth)
        }}
      />
```

In `DataModel`, here’s how we clean up.

```
  disconnectOnSignout() {
    if (this.userSnapshotUnsub) {
      this.userSnapshotUnsub();
      this.userSnapshotUnsub = undefined;
    }
  }
```

1. user snapshot listeners

```
  addUserSnapshotListener(callback) {
    const id = Date.now();
    this.userSnapshotListeners.push({
      callback: callback,
      id: id
    });
    callback();
    return id;
  }
  removeUserSnapshotListener(id) {
    const idx =
      this.userSnapshotListeners.findIndex(elem => elem.id ===
id);
    if (idx !== -1) {
      this.userSnapshotListeners.splice(idx, 1);
    }
  }

  notifyUserSnapshotListeners() {
    for (usl of this.userSnapshotListeners) {
      usl.callback();
    }
  }
```

In `MainScreen`:

```
  useEffect(()=>{
    dataModel.addUserSnapshotListener(async () => {
      setUserDisplayName(await dataModel.getCurrentUserDisplay
Name());
    });
  }, []);
```

Review 2: Logging in and Creating Users

Logging in:

```
signInWithEmailAndPassword(auth, email, password);
```

Creating a user—here we need to create the auth user, but also a record for that user in Firestore so we can keep track of metadata like their `displayName`. We'll also need this user `Document` to store photo gallery information later. In `LoginScreen`:

```
    const credential = await createUserWithEmailAndPassword(
      auth, email, password);
    const authUser = credential.user;
    dataModel.createUser(authUser, displayName);
```

In `DataModel`:

```
  createUser(authUser, displayName) {
    setDoc(doc(db, 'users', authUser.uid), {displayName: displ
ayName});
  }
```

Review 3: Taking Pictures

When the user presses the camera button on `MainScreen`...

```
      <TouchableOpacity
        onPress={()=>{navigation.navigate('Camera')}}
      >
        <Icon
          name='photo-camera'
          size={32}
        />
      </TouchableOpacity>
```

... it will launch the `CameraScreen`, which allows the user to take a picture. For now, we just print out the picture information:

```
export function CameraScreen({navigation}) {
  const dataModel = getDataModel();
  const [hasPermission, setHasPermission] = useState(null);

  useEffect(() => {
    async function getPermissions(){
      const { status } = await Camera.requestCameraPermissions
Async();
```

```
      setHasPermission(status === 'granted');
    }
    getPermissions();
  }, []);

  if (hasPermission === null) {
    return <View />;
  }
  if (hasPermission === false) {
    return <Text>No access to camera</Text>;
  }
  let theCamera = undefined;

  return (
    <View style={styles.cameraContainer}>
      <Camera
        style={styles.camera}
        ratio='4:3'
        ref={ref => theCamera = ref}
      />
      <TouchableOpacity
        style={styles.cameraControls}
        onPress={async ()=>{
          let picData = await theCamera.takePictureAsync({qual
ity: 0.2});
          console.log('took a picture:', picData);
          navigation.goBack();
        }}>
        <Text style={styles.snapText}>Snap!</Text>
      </TouchableOpacity>
    </View>
  );
}
```

That's it for the starter code. Now let's start adding some new stuff!

Sorta New Stuff: Adding the Photo to storage

The first thing we'll do isn't really new cuz we did it last week, but we're going to do it a bit differently this week since we want to display a *gallery* of pictures, rather than just one picture like we did before.

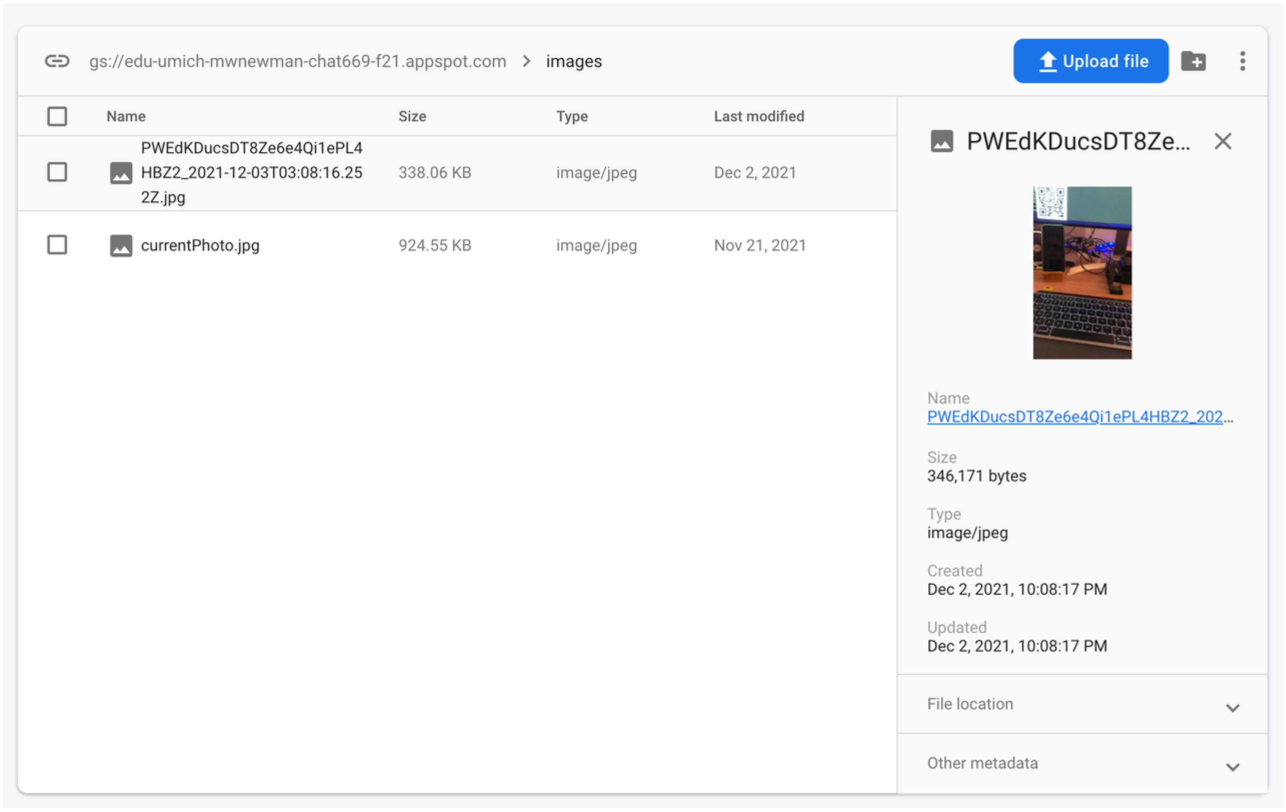
We still need to add the photo to Storage, but this time we'll create a unique filename for each image by combining the user ID and the timestamp. In `DataModel`:

```
  async savePicture(picData) {
    // get the image data from local storage
    const response = await fetch(picData.uri);
    const imageBlob = await response.blob();

    // upload the image data to Storage after creating a uniqu
e filename/ID
    const userID = auth.currentUser.uid;
    const timeStamp = new Date();
    const timeString = timeStamp.toISOString();
    const fileName = userID + '_' + timeString + '.jpg';
    const fileRef = ref(storage, 'images/' + fileName);
    await uploadBytes(fileRef, imageBlob);
  }
```

Now call `dataModel.savePicture()` from `CameraScreen`

```
      <TouchableOpacity
        style={styles.cameraControls}
        onPress={async ()=>{
          let picData = await theCamera.takePictureAsync({qual
ity: 0.2});
~~          console.log('took a picture:', picData);~~
          dataModel.savePicture(picData);
          navigation.goBack();
        }}>
        <Text style={styles.snapText}>Snap!</Text>
      </TouchableOpacity>
```

To test this out, take a picture and look in your Storage bucket to see the results. It should be in the `/images` folder within your Storage bucket.



Name	Size	Type	Last modified
PWEdKDucsDT8Ze6e4Qi1ePL4HBZ2_2021-12-03T03:08:16.252Z.jpg	338.06 KB	image/jpeg	Dec 2, 2021
currentPhoto.jpg	924.55 KB	image/jpeg	Nov 21, 2021

Name
PWEdKDucsDT8Ze6e4Qi1ePL4HBZ2_202...

Size
346,171 bytes

Type
image/jpeg

Created
Dec 2, 2021, 10:08:17 PM

Updated
Dec 2, 2021, 10:08:17 PM

New Stuff 1: Saving the Picture information in Firestore

We're going to use Firestore to associate the pictures that each user takes with the user. We'll do this by creating a `pictures` collection within the user `Document` and stashing the information we need to retrieve the picture from Storage whenever we need it. The most important piece of information will be the `downloadURL`, which is a special “public but hard to guess” URL that can be generated for every file in Storage.

To get the `downloadURL` for our newly stored picture,

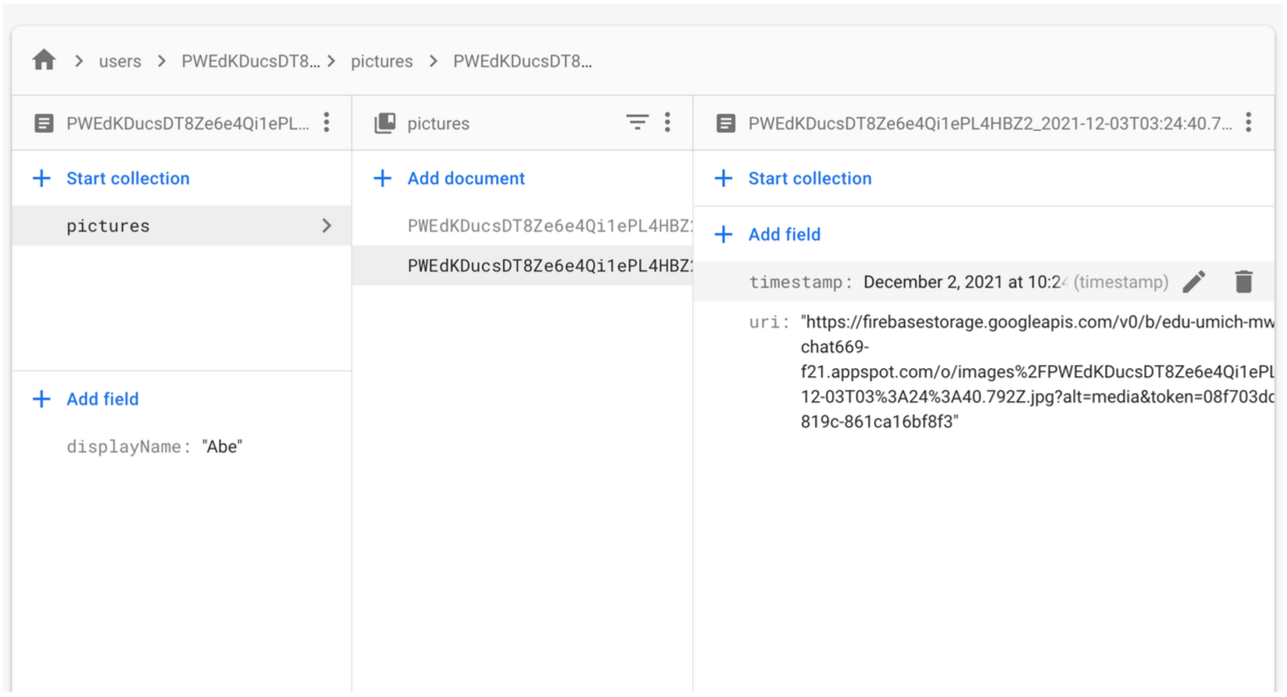
```
  async savePicture(picData) {
    // get the image data from local storage
    const response = await fetch(picData.uri);
    const imageBlob = await response.blob();

    // upload the image data to Storage after creating a uniqu
e filename/ID
    const userID = auth.currentUser.uid;
    const timeStamp = new Date();
    const timeString = timeStamp.toISOString();
    const fileName = userID + '_' + timeString + '.jpg';
    const fileRef = ref(storage, 'images/' + fileName);
    await uploadBytes(fileRef, imageBlob);

    // get the downloadURL for the fileRef we just saved
    const downloadURL = await getDownloadURL(fileRef);
    console.log(this.downloadURL);
  }
```

Test this out, you should see a long, ugly URL print to the console. You should see each new picture in Storage as well.

Now that we have the `downloadURL`, stash it in Firestore, along with some other useful info.

```
  async savePicture(picData) {
    // get the image data from local storage
    const response = await fetch(picData.uri);
    const imageBlob = await response.blob();

    // upload the image data to Storage after creating a uniqu
e filename/ID
    const userID = auth.currentUser.uid;
    const timeStamp = new Date();
    const timeString = timeStamp.toISOString();
    const fileName = userID + '_' + timeString + '.jpg';
    const fileRef = ref(storage, 'images/' + fileName);
    await uploadBytes(fileRef, imageBlob);

    // get the downloadURL for the fileRef we just saved
    const downloadURL = await getDownloadURL(fileRef);
    console.log(this.downloadURL);

    // put the downloadURL, along with other into, into Firest
ore
    const userDoc = doc(db, 'users', userID); 上传firestore的图片url
    const picturesColl = collection(userDoc, 'pictures');
    const pictureDoc = doc(picturesColl, fileName);
    const pictureData = {
      timestamp: timeStamp,
      uri: downloadURL
    };
    setDoc(pictureDoc, pictureData);
}
```

Verify this works by looking in Firestore after taking another picture:



users > PWEdKDucsDT8... > pictures > PWEdKDucsDT8...

PWEdKDucsDT8Ze6e4Qi1ePL...

Start collection

pictures

Add field

displayName: "Abe"

pictures

Add document

PWEdKDucsDT8Ze6e4Qi1ePL4HBZ

PWEdKDucsDT8Ze6e4Qi1ePL4HBZ

PWEdKDucsDT8Ze6e4Qi1ePL4HBZ2_2021-12-03T03:24:40.7...

Start collection

Add field

timestamp: December 2, 2021 at 10:2 (timestamp)

uri: "https://firebasestorage.googleapis.com/v0/b/edu-umich-mw chat669-f21.appspot.com/o/images%2FPWEdKDucsDT8Ze6e4Qi1ePL 12-03T03%3A24%3A40.792Z.jpg?alt=media&token=08f703dc 819c-861ca16bf8f3"

Now all we need to do is display the images!

Now You Try #1
Now we have a familiar situation. We have some data in a Firestore collection that we want to display in our app. We want to make sure that our app displays the data when it first starts up (or, in our case, when the user logs in—i.e., when the `MainScreen` mounts for the first time), and we want to make sure that it displays the latest greatest data whenever that data changes (e.g., when the user takes a new picture). Let’s brainstorm what steps we need to carry out in order to make this happen.
This slido poll is set up to “accept multiple answers,” so you can post things as soon as you think of them. We’ll keep this one live and discuss ideas as they come up.

Here are the ones I thought of while preparing the lecture:

- MainScreen needs to subscribe to updates from DataModel
- DataModel needs to subscribe to updates from Firestore
- MainScreen needs to display images in a FlatList

MainScreen subscribes to updates

```
  const [pictures, setPictures] = useState([]);

  useEffect(()=>{
    dataModel.addUserSnapshotListener(async () => {
      setUserDisplayName(await dataModel.getCurrentUserDisplay
Name());
    });
    dataModel.addCurrentUserGalleryListener(pictures => {
      setPictures(pictures);
    });
  }, []);
```

...so next we need to implement `DataModel.addCurrentUserGalleryListener()`...

DataModel subscribes to onSnapshot (while also handling the listener)

```
  addCurrentUserGalleryListener(callback) {
    if (this.gallerySnapshotUnsub) {
      this.gallerySnapshotUnsub();
    }

    const q = query(
      collection(db, 'users', auth.currentUser.uid, 'picture
s'),
      orderBy('timestamp', 'desc'));

    this.gallerySnapshotUnsub = onSnapshot(q, qSnap => {
      let pics = [];
      qSnap.forEach(docSnap => {
        let picData = docSnap.data();
        picData.key = docSnap.id;
        pics.push(picData);
      });
      callback(pics);
    });
  }
```

Note that since we only expect one listener, we can take a shortcut and not maintain a list of listeners, support unsubscription, etc. (though this latter one will prove to be a bit shortsighted as we may see). Instead, we just call the`MainScreen`'s `callback()` within the `onSnapshot()` callback.

This will result in the `MainScreen`'s callback being invoked whenever the curren't user's `pictures` collection changes (so, right away when `onSnapshot()` is set up, and anytime the collection changes after that), which in turn will update the `MainScreen`'s `pictures` state variable.

```
    dataModel.addCurrentUserGalleryListener(pictures => {
      setPictures(pictures); 设置每张图片的回调函数
    });
```

All that's left is for the `MainScreen` to display the images:

MainScreen displays the "images"

For our first pass and sanity check, let's just have the `MainScreen` display the `downloadURL`s (aka `uri`s). These should show up on the screen, and taking a new picture should generate a new item in the list.

```
  return (
    <View style={styles.container}>
      <Text> Hi, {userDisplayName}! </Text>
      <Button
        title='Sign out'
        onPress={()=> {
          dataModel.disconnectOnSignout();
          signOut(auth)
        }}
      />
      <TouchableOpacity
        onPress={()=>{navigation.navigate('Camera')}}
      >
        <Icon
          name='photo-camera'
          size={32}
        />
      </TouchableOpacity>
      <View style={styles.gallery}>
        <FlatList
          data={pictures}
          renderItem={({item}) => {
            return(
              <View>
                <Text>{item.uri}</Text>
              </View>
            );
          }}
        />
      </View>
    </View>
  );
```

You’ll need a new style…

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  gallery: {
    flex: 0.7,
    width: '100%',
    alignItems: 'center'
  },
...
```

Now You Try #2

Remember how to display images from last week? Sure you do! Work on making the images actually show up on the `MainScreen`. You might want to refer to last week's lecture notes for a refresher (pro tip: open this in a new tab so you don't lose your place here).

Paste your revised `renderItem()` code into slido.

It's pretty simple!

```
      <View style={styles.gallery}>
        <FlatList
          data={pictures}
          renderItem={({item}) => {
            console.log(item);
            return(
              <View>
                <Image
                  source={item}
                  style={styles.galleryImage}/>
              </View>
            );
          }}
        />
      </View>
```

```
  gallery: {
    flex: 0.7,
    width: '100%',
    alignItems: 'center'
  },
  galleryImage: {
    height: 200,
    width: 200,
    resizeMode: 'contain',
    margin: 20
  },
```

Now sign out and sign in as a different user. Take a few pictures as the new user!

Final example code: https://github.com/SI669-internal/week13Gallery/tree/addGallery

Announcements

1. Checkpoints
   a. Will try to get feedback as soon as possible—probably over the weekend.
   b. If you have specific concerns/questions that you need addressed to make progress, ping me on Slack.
2. Final Presentations
   a. Expectation: In Person and On Time (arrive by 8:30)
   b. If you can't meet that expectation, let me know (and have a good reason!)
   c. Sign up for your slot. (People who come to class get first crack!)
   d. You must demo "on a phone" via your laptop
      i. iOS & Mac: use Quicktime Player

   ii. Other platforms: use Zoom (create your own meeting) and share screen from phone
   iii. Simulators/Emulators are OK too
   iv. **TEST THIS THOROUGHLY USING MWIRELESS BEFORE YOUR DEMO!!!!!!!**
3. SI 699-3 User-Centered Agile Development is *way* undersubscribed (2020: **24**, 2021: **20**, 2022: **7**). Why?
   a. I'd love any feedback, personal concerns, rumors, etc. — it can be anonymous! (Use the SI 669 Anonymous Feedback Form)

Break time!

📱📱📱 Surprise! One more App of the Week! 📱📱📱

🌎🌍🌏 Shuwan Feng: Location, Location, Location! 🌎🌍🌏

Notifications

Notifications are a critical aspect of many mobile apps. If your app needs to remind people about something, bring their attention to something, or send them a message of any kind, notifications are how you will do this. Notifications are quite powerful, because they allow your app to *initiate* interaction, rather than waiting around for the user to decide to do something.

Notifications are a rich area when it comes to design: How can you best design them to engage the user? When should they appear and how often? How do you balance between your app's desire for user engagement and the user's desire to allocate their attention where they like? How do you design a notification strategy that doesn't lead to disengagement, which can happen when notifications feel like a burden or when they turn into "noise" because they are too frequent and not valuable?

Unfortunately, we won't be able to get into those considerations here, but we can take a quick look at how to *use* one type of notification in your React Native app.

Broadly speaking there are two types of notifications that can be used in mobile apps:

- **Push Notifications** are initiated by a backend service of some sort, and can "push" information to the user's device at any time. To make this work, the user's device needs to be "registered" with the backend so that the backend can send notifications at any time. Push notifications are somewhat complicated and work slightly differently on iOS and Android, but 3rd party services, including one provided by Firebase, can help to make the whole process less painful.
- **Local Notifications** are initiated by an app running on the user's device, and can be delivered right away (which isn't very useful) or scheduled for a specific time in the future. The future time can be relative to "now" (e.g., one hour from "now") or a specific date and time (e.g., 8am on December 25, 2021). Various other ways of specifying the notification delivery time are also possible.

Both types of notifications have a few things in common:

- Notifications are handled by the OS, not your app. In both cases, you configure the notification and hand off control to the OS. This is essential, because it allows notifications to be shown when your app isn't running—which is pretty much always when you want a notification to be shown. With the OS handling notifications, not only does your app not need to be currently in use, it doesn't even need to be loaded into memory. The user's phone doesn't even need to be awake. In many cases, it doesn't even need to be ON—as the notification will be delivered once the user turns the phone back on, whenever that may be.

- Notifications appear the same way to the user, whether they're Push or Local. Generally speaking they will show the app that generated them, a title, and a message body. They can also show a thumbnail image and, in some cases, buttons that allow the user to do something. By default, when the user taps on an app's notification, the app will be shown to the user. If the app is already running, the user will see whatever screen they were on when they last used it. If the app isn't running, it will be loaded from scratch and the default screen will be shown. Both of these behaviors can be overriden by defining NotificationHandlers, but we're not going to get into this.

Push vs Local?

How do you know which type of notification to use? Here are a few considerations:

Push

- Pro: Can inform the user of events that occur *off of the phone*. Examples might include messages sent by other users, social media posts, real-world process updates (e.g., package tracking, DoorDash delivery status), contextual triggers (e.g., weather alerts).
- Con: Requires a backend. The logic for delivering Push Notifications *cannot* reside within your mobile app. This means you need to develop a backend in addition to your mobile app. There are some services that allow you to create backend logic without building a full-fledged server—Firebase Cloud Functions are one example—but there's no getting around the fact that needing a backend will add significant complexity to your app.

Local

- Pro: Can provide information to the user at specified times, provided that (1) the timing can be determined at when the notification is scheduled and (2) the information can be determined at schedule-time as well. Local notifications are perfectly fine for reminders (e.g., remember to take your medicine at 8am, remember to turn in your assignment by 11:59pm). They're not good for much else.

- Pro: No backend required.
- Con: Not useful for anything where the timing and/or information cannot be determined by the app running on a user's phone. Or anything where scheduling might need to happen when the app isn't running.

So without further ado, here is your most basic scheduling app! Replace `App.js` with this:

```
import React, { useEffect, useState } from 'react';
import { Button, StyleSheet, Text, View } from 'react-native';
import * as Notifications from 'expo-notifications';

export default function App() {
  const [ hasPermission, setHasPermission ] = useState(false);

  useEffect(() => {                                设置notification权限
    async function getPermissions(){
      const { status } = await Notifications.requestPermission
sAsync();
      setHasPermission(status === 'granted');
    }
    getPermissions();
  }, []);

  if (!hasPermission) {
    return (
      <View style={styles.container}>
        <Text>Notification permissions not granted.</Text>
      </View>
    );
  }

  return (
    <View style={styles.container}>
      <Button
        title='Schedule Notification (15s)'
        onPress={async ()=>{
          await Notifications.scheduleNotificationAsync({
            content: {
              title: "week13Notif",
              body: "Here is your notification!",
            },
            trigger: {
              seconds: 15
            }
          })
        }}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Let's break that down:

First, we need to get the user's permission to send notifications. This is very similar to what we did with the `Camera`.

```
  const [ hasPermission, setHasPermission ] = useState(false);

  useEffect(() => {
    async function getPermissions(){
      const { status } = await Notifications.requestPermission
sAsync();
      setHasPermission(status === 'granted');
    }
    getPermissions();
  }, []);

  if (!hasPermission) {
    return (
      <View style={styles.container}>
        <Text>Notification permissions not granted.</Text>
      </View>
    );
  }
```

Then, when the user taps the `Button`, we schedule a notification for 15 seconds in the future:

```
  return (
    <View style={styles.container}>
      <Button
        title='Schedule Notification (15s)'
        onPress={async ()=>{
          await Notifications.scheduleNotificationAsync({
            content: {
              title: "week13Notif",
              body: "Here is your notification!",
            },
            trigger: {
              seconds: 15
            }
          })
        }}
      />
    </View>
  );
```

Note that to test this out, you'll need to tap the button and then quickly exit the app (going to the phone's home screen is adequate—you don't need to kill the app). By default, notifications for an app won't display if the app is in the foreground. This can be overriden, but we're not going to get into that. Try it out!

The core of this whole business is `Notifications.scheduleNotificationAsync()`. This takes one argument, which is a `NotificationRequestInput` object. This object is pretty straightforward—it has just two properties: `content` and `trigger`. `content` specifies what the notification *says* (e.g., the title and message body), and `trigger` specifies the timing. Both of these have a number of properties—most of which you'll have to explore on your own if you need to use them. The ones shown in the example above are pretty self-explanatory and can cover most of what you will want to do. The `trigger` property accepts a wide variety of properties, including several variations of recurring triggers. Here are a few examples:

Schedule a notification for 8am, Dec. 25, 2021:

```
            trigger: {
              year: 2021,
              month: 12,
              day: 25,
            hour: 8,
          }
```

Schedule a notification for every day at 11:15am:

```
          trigger: {
            hour: 11,
            minute: 15,
            repeats: true
          }
```

Schedule a notification for every Thursday at 6pm:

```
          trigger: {
            weekday: 5 // 1 = Sunday; 7 = Saturday
            hour: 6,
            repeats: true
          }
```

And so forth...

Now You Try #3

Schedule three notifications:

- One that says “Have a great day!” (title: “Good Morning”) every day at 8:30am
- One that says “You made it!” (title “TGIF”) every Friday at 5pm
- One that says “And many more!” (title: “Happy Birthday”) every year on your birthday. You pick the time of day.

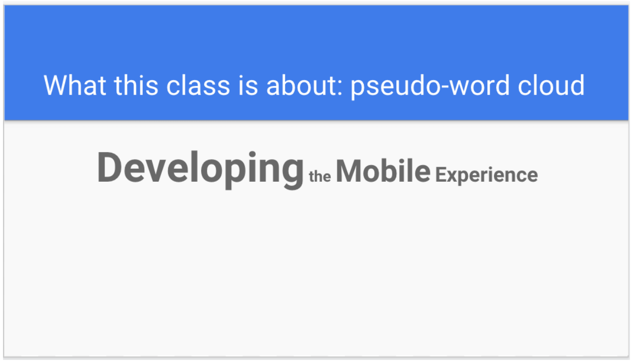
Put your answers into slido.

```
// have a great day
{
  content: {
    title: 'Good Morning',
    body: 'Have a great day!'
  },
  trigger: {
    hour: 8,
    minute: 30,
    repeats: true
  }
}

// TGIF
{
  content: {
    title: 'TGIF',
    body: 'You made it!'
  },
  trigger: {
    weekday: 6,
    hour: 17,
    repeats: true
  }
}

// Happy Birthday
{
  content: {
    title: 'Happy Birthday',
    body: 'And many more!'
  },
  trigger: {
    month: 10,
    day: 6,
    hour: 18,
    repeats: true
  }
}
```

Here's the example from today: https://github.com/SI669-internal/week13Notif

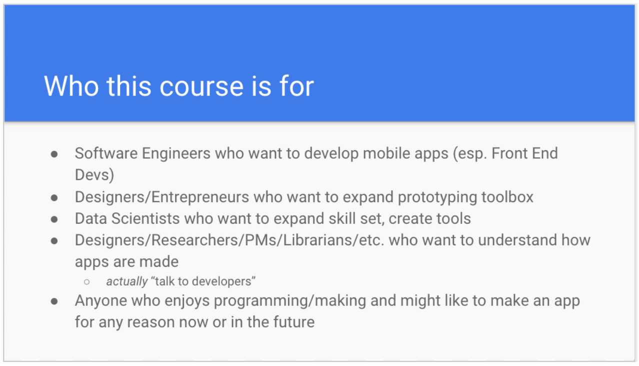
Wrap Up

Where we started...



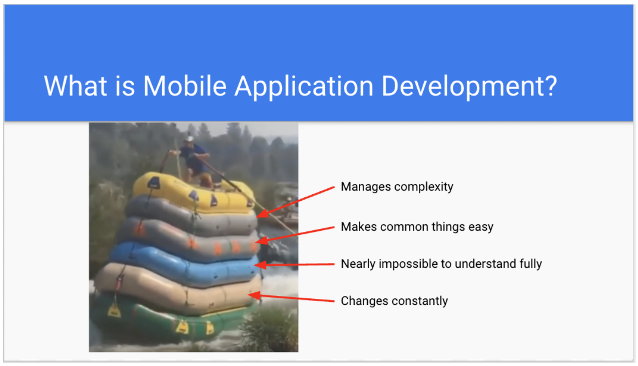
What this class is about: pseudo-word cloud

Developing the **Mobile** **Experience**



Who this course is for

- Software Engineers who want to develop mobile apps (esp. Front End Devs)
- Designers/Entrepreneurs who want to expand prototyping toolbox
- Data Scientists who want to expand skill set, create tools
- Designers/Researchers/PMs/Librarians/etc. who want to understand how apps are made
  - *actually* "talk to developers"
- Anyone who enjoys programming/making and might like to make an app for any reason now or in the future



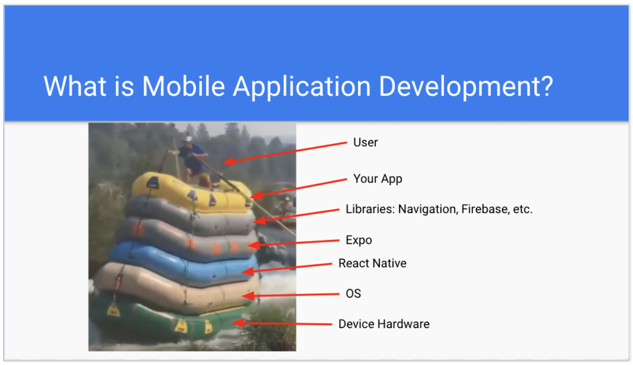
What is Mobile Application Development?

Manages complexity

Makes common things easy

Nearly impossible to understand fully

Changes constantly



What is Mobile Application Development?

User

Your App

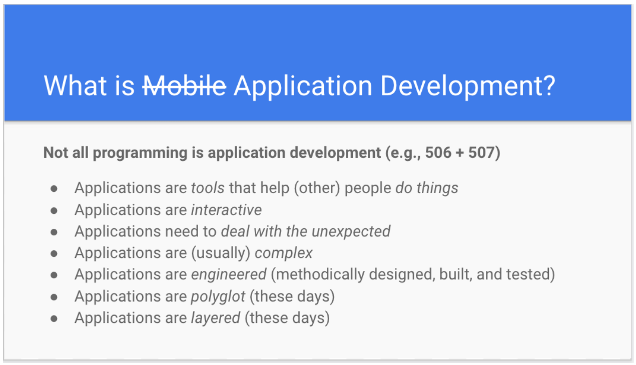
Libraries: Navigation, Firebase, etc.

Expo

React Native

OS

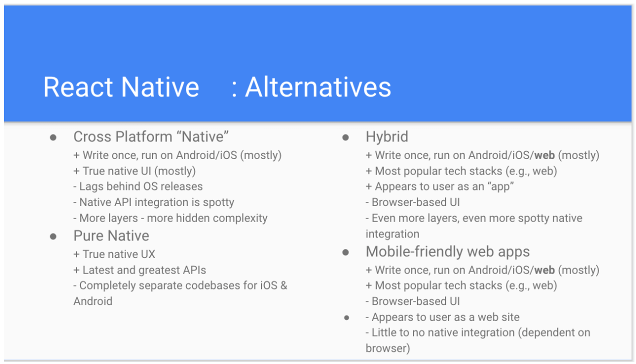
Device Hardware



What is ~~Mobile~~ Application Development?

Not all programming is application development (e.g., 506 + 507)

- Applications are *tools* that help (other) people *do things*
- Applications are *interactive*
- Applications need to *deal with the unexpected*
- Applications are (usually) *complex*
- Applications are *engineered* (methodically designed, built, and tested)
- Applications are *polyglot* (these days)
- Applications are *layered* (these days)



React Native : Alternatives

- Cross Platform "Native"
  + Write once, run on Android/iOS (mostly)
  + True native UI (mostly)
  - Lags behind OS releases
  - Native API integration is spotty
  - More layers - more hidden complexity
- Pure Native
  + True native UX
  + Latest and greatest APIs
  - Completely separate codebases for iOS & Android
- Hybrid
  + Write once, run on Android/iOS/**web** (mostly)
  + Most popular tech stacks (e.g., web)
  + Appears to user as an "app"
  - Browser-based UI
  - Even more layers, even more spotty native integration
- Mobile-friendly web apps
  + Write once, run on Android/iOS/**web** (mostly)
  + Most popular tech stacks (e.g., web)
  - Browser-based UI
- - Appears to user as a web site
  - Little to no native integration (dependent on browser)

Working with Developers

- Some ideas are *much* harder to implement than others
- But there's often an easy way—can push back when a dev says "no"

Prototypes and MVPs

- Recognize the difference between throwing something together and engineering a robust system

Application Development

We established a base, but don't stop here!

JS basics & asynchrony

- objects and properties, including dereferencing
- callbacks, callbacks, callbacks
- all variants of function syntax
- module import and export
- how to name variables and functions
- asynchrony: callbacks, Promises, and async/await
- listener patterns

Event-driven programming

- Nothing happens in React Native (or JavaScript, really) without an event
- `onPress()`
- `onValueChange()`
- `onSnapshot()`
- `useEffect()`
- `setState()`

JSX

- instantiating built-in, 3rd party, and custom components
- integrating JSX and JS expressions
- expressions vs statements
- conditional rendering (with conditional operator and otherwise)

FlatLists

- data, renderItem, and item

Layouts and Styles

- transfer of knowledge to/from HTML/CSS
- not my strong suit!

Navigation

- passing and reading params—a lot like passing data using Forms or QueryStrings

CRUD

- Treat it as a checklist: C, R, U, and D
- Keep track of what's happening in memory (e.g., state variables defined using `useState()`, variables maintained in a `DataModel`) and what's happening in persistent storage (e.g., Firebase Collections and Documents, Storage)

UI components

- We looked at several, but there are many more!
- read the docs!

Firebase

- Hopefully you understand the basic concepts, but there's a lot more
- read the docs!!

Camera, Image, Notifications, etc.

- read the docs!!!

What's next? Here's my list

- Ejecting from expo and building “true native” apps for beta testing and app store release
- Beta test and App store deployment (I have a bit of experience with this—feel free to ask!)
- Test Automation and Debugging
- Continuous integration, build automation, version management

- Redux state management
- Push notifications
- Integrating with backends using REST
- Firebase Cloud Functions
- Different types of authentication and stronger security

- Visualization libraries
- Device rotation (Portrait vs Landscape)
- Location & Map UI components
- Background services (e.g., continuous location tracking, geofencing)
- Other sensors (accelerometer, microphone, gyro) — background and foreground
- Health & fitness APIs
- Wearable integration
- Bluetooth (esp BLE) for IOT applications
- Augmented Reality
- …

It's been fun!