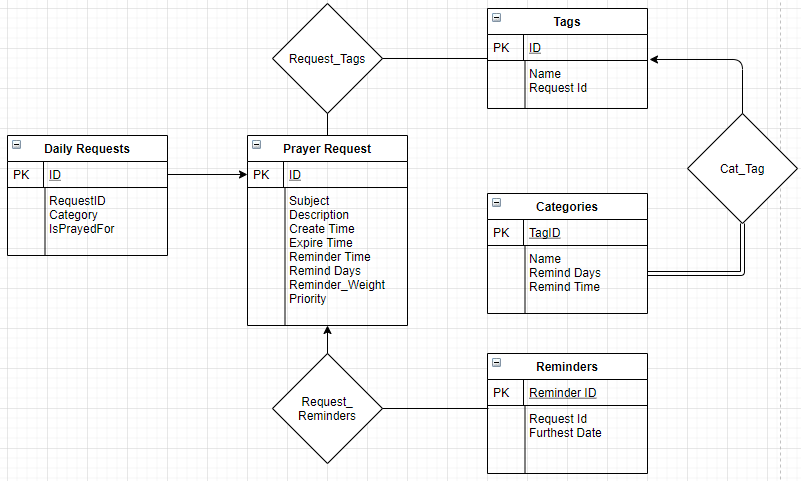
**Development Environment**

We used React Native and Expo because it was cross platform and it was easy to test the app. React native is the framework and Expo is the tool that makes building and testing really easy. We tried some other development environments (like Xamarin) during the testing phase but testing it was a lot more difficult. For React Native, you can download the Expo app to your phone, scan a QR code with it, and then it will download a sandbox version of the app that you can see live updates to if you change the code. Xamarin required you to basically build the app and side load it to your phone every time you wanted to test it. I think emulators might have also been an option but overall, it was just more cumbersome than React Native (which can also be run on emulators).

We kept a text document called “Oreo commands.txt” that specifies what libraries need to be installed. Basically, React Native is not quite like Python or Java where you can just import whatever functionality or libraries you want. You have to rely on what other developers have built and *published* on github. You can download these libraries like sqlite (database library), radio buttons, dropdowns, etc... with npm or yarn (We used npm. It’s important to be consistent. We weren’t consistent at first so sometimes it caused weird errors.) The easiest way to set up the dev environment is to run all the commands in the “Oreo commands.txt” in whatever directory is going to be the top level for the project and then git pull origin master or whatever to get all the actual project files. For example, I created a folder on my desktop (for easy access) called GOPray. I navigate to that folder using the power shell. I run “npm install -g expo-cli”, then “expo install expo-sqlite”, then the rest of the commands. Then, because I have git bash installed on my computer, I can just use the already existing git commands to pull the repo files down. If I did everything correctly, I should be able to run “npm start” and it’ll open a tab in my browser with a QR code I can scan with my phone and the expo app and it’ll run. If we ever had to install new libraries, we would just add them to the “Oreo commands.txt” and the other team members could run the same command and everything should sync up just fine. Just be careful not to ever add the config files (like yarn.lock, package.json, package-lock.json, etc…) to any commits. It should really be in the .gitignore but we didn’t get around to it so if anyone ever committed that, it just became the worst to try and fix it. We could also build a .apk that we could side load using “expo build:android -t apk”.

**Database design**



*Misc*

Above you can see our database schema. Categories is really just an extension of the tags category. It makes some queries easier (for example, we can just say “select \* from categories” on the categories or “Journals” page instead of having to say “select \* from tags where isCategory = 1” or something to that effect) and it makes other’s harder (changing a category name for example must be done in 2 places, in tags *and* in categories, since we’re storing redundant information). Overall, in my database queries, I chose to do more work in the query and less processing afterwards. For example, I could’ve just done “select \* from request, tags where req.reqId = tag.reqId” and then manually looped through and removed the expired ones, but instead, I’d have these nasty long queries that return exactly the data in the form that I need it. This is partly based on the assumption that the database will optimize its queries better than I can optimize my for loops. One thing to look out for in queries is that some OSs treat the select differently. For example, “select request.id from request” returns something like results.rows.\_array which we assign to results. Then we can say results[0].id to get the id. But, android 7 errors when we do this while android 9 and ios have no problem. Android 7 wants us to say results[0].request.id, but that’s obviously annoying, so we modify the query to be “select request.id *as id* from request” so we can treat all queries the same. We also have a miscellaneous flags table that mostly just keeps track of when requests were last scheduled. We were going to have some other flags like “app has been opened before” but ended up not needed to store it.

*Reminder Table*

Back to the schema. We have the reminder table so that if the user deletes or expires a request, we can look up the request id in this table and delete the reminders that we scheduled for that request. I didn’t handle any of the reminder scheduling though, for more details on that, check Noah’s brain dump. I just created the database and schema.

*Categories Table*

We store the days of the week as a string. For example, “0101010” represents requests that will show up on MWF. “1111111” is every day. “1000001” is only on the weekend, etc… Remindertime is just a date stored as an integer, but we really only care about the time part of the date, we just couldn’t find a simple way to store it as a time and not a date. The user selects what days they want the category to get scheduled for, and a time on those days, and then they get reminders at those days and time for that category. As the schema indicates, a category must be associated with a tag, but not every tag will have a category.

*Request table*

The request table stores some pretty obvious information such as the subject and description. It also stores a priority which is taken into consideration when calculating which requests to schedule each week, and an expire time so that the user won’t amass a list of requests which he isn’t taking seriously. The weight is also used in calculating which requests to schedule. Remind days and remind time are currently unused. Originally, we wanted the user to be able to make a request basically to have very high priority and pop up every week at at specific days and times just like categories, but we ended up not adding this feature. We left the fields in the database in case some later developers decided to add this feature. Create time is properly set, but it’s not really used for anything. We thought maybe if the user wanted to look back at when requests were created or how long they’ve been praying for something, storing the create time would be a nice record but we just didn’t really use it for anything.

*Tags Table*

Tags are how we sort and keep track of each request. It’s not supposed to be possible to create or modify a request in such a way that it isn’t associated with any tags. Some tags are special in that they are categories. When you create a request and associate it with one or multiple tags (one of which should be a category tag, provided we don’t have any bugs that allow otherwise) you will also create that many request tags, but I’ll go over that a lot more when I talk about the individualRequest.js screen.

**Screens design**

A lot of the initial startup and difficulties of this app exist because we didn’t know how to use react native. We didn’t know how to pass data between screens, or to get data out of the database elegantly. I’ve heard rumors about Dr. G teaching react native in the Webapps class but we weren’t taught react native so in this section I’ll talk about how react native works, (functions, arrow functions, callbacks, etc…) and how we chose to use it. Then I’ll go into some details about each of the screens and how we update data in the database and such.

*Navigation setup*

First, navigation. Originally, we just wanted buttons on each screen that would kind of lead to other screens. So you start out at the welcome screen. You hit start and it takes you to the dashboard. Then you can hit the new request and it will take you to the individual request screen, and so on. So, the way we implemented that is by creating a stack navigator in our main app.js and set the welcome screen to be the main screen or start screen or whatever. But then we wanted a drawer that we could drag out and that would do things like let you jump back to the dashboard, or view some extra screens like the “all requests” screen where you could search by tag, etc… But we realized that this “drawer” navigator didn’t play as nicely with the stack navigator as we were hoping. It’s complicated, and has to do with nested navigators. In as few words as possible, some of the app is sequential (welcome→ dashboard→ categories→ requests→ individual req, i.e. we wan’t a stack) whereas some other parts of the app are kinda random (tags isn’t really connected to all requests which isn’t really connected to the about page which isn’t connected to the dashboard and it’s screens, i.e. drawer type navigation). What happens is that the sequential part of the app gets treated as a single screen by the drawer navigator. The confusing part is that now the drawer has a dashboard button that doesn’t go to the dashboard it goes to the dashboard navigator, which may be the dashboard screen, but it may also be whatever screen you were last on, such as the request or categories screen. That’s too bad because we were wanting it to go back to the dashboard screen. So our solution was to have the drawer only accessible from the dashboard screen (and the other drawer screens like all reqs or all tags, but not the categories page or requests page, etc…), and to add a “home” button that just jumps back to the dashboard from most of its sequential screens. I don’t know if that made sense but basically, there’s a reason we have a root navigator that has a drawer navigator that has several screens plus a stack navigator that has several screens. Below is a tree of what our navigators look like. The navigators hold screens, and the screens are where most of the code is. Also, often, it can be helpful to be able to pass data between screens. For example, if you are on the requests page and click on a request, passing something like the request id to the next page so that that screen can query the database and load the rest of the information is a pretty intuitive thing to do. This is done whenever you call navigation.navigate. You can create an object with all the fields you want to pass like the request\_id or category\_id or whatever you want. Then to access it on the next page, you can say route.params.whatever and you can access that data.

Root{

Drawer {

Dashboard (navigator, it’s a stack type) {

Dashboard (screen)

Welcome

Categories

Requests

Daily Prayer

Scan QR Code

Individual Request

}

All Requests

All Tags

Scan QR code (Same screen as above, different instance)

About

}

}

*Understanding code location*

App.js is what is going to run automatically, so in some way or another, all the rest of the code needs to be connected to app.js. This is done though imports (no duh). We create other screens like a welcome screen which we can import something like “import \* as welcome from ../pages/welcome.js”. This is imported into one of the navigators and then we can jump to it through the navigator tools, which I’ll discuss later. We can also have code that isn’t really connected to any kind of display code. In the case of the various screens, they will all have html-esq code that will display buttons and text and such to the screen, but for things like the database calls or the scheduler, these are just disembodied functions. We can still import them just like before. Javascript lets you import everything as an object or as individual functions. I can say import \* as functions from funcs.js and then say functions.func1(), or I can say import { func1 } from funcs.js and then just say func1(), it’s the same thing. We tried to organize stuff in a fairly logical way. Screens go under “pages”, any database calls go under “database”. We tried to further organize these calls by whether they were inserting, deleting, modifying, creating, querying, etc… various fields in the database. Importing files from different directories works just like you expect it to. “..” means go up a directory, “.” is the current directory, and you can go down directories by specifying “/directoryName” or whatever.

*Understanding React Native Functions and Paradigm*

Again, when we first started, we had no idea how react native worked, so we were pretty confused when we would try to do something as simple as write a “Hello World” app and we would find 3 different implementations. As it turns out, react native has 2 main ways of doing things. An old way that they are/were trying to phase out, and a new way that is supposed to be better. The old way involves classes. Like other object oriented languages such as java, everything is a class. Namely, each component is a class, and has to do things like render the screen, update the screen, interact with the screen, etc...there are these things called lifecycle hooks which help with the constructor, rendering, checking if components mounted or unmounted or updated and so on. But, since this was the old way of doing things that supposedly wasn’t as good, we obviously chose not to do it this way. I just mention it so that if you see lots of things like classes, lifecycle hooks, components, render methods, etc...and are wondering why these things are nowhere in our codebase, it’s because we used the other method. Here’s a code snippet from a technical presentation we had to do showing a couple of the differences between the two.

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

The other method is to use functions instead of classes, and as you can see above, the first block is a function that returns a component, specifically a header component which greets the name of the user which was passed in through props. Here’s where things get even more complicated though. Even within the function paradigm (as opposed to the class paradigm) there are several ways to declare things.

function foo (a, b) { (a, b) => {

return a + b; return a + b;

} }

let foo = (a, b) => {

return a + b;

}

These 3 snippets are almost equivalent. They do the exact same thing. There are just some slight differences in how the scope (specifically global scope, I think? Or it could be the “parent” scope) works. I never came across an instance where the scope nuances ended up burning me though. It seems like a very niche thing. For the most part, we just used named arrow functions except when we were writing a callback function (discussed below) in which case we just used a straight arrow function. If you want to try and understand the scope, just google “arrow functions and scope” or something like that and you should be able to find several different explanations of it pretty easily.

*Understanding Variables (accessing and setting) and Hooks*

Obviously, you’ll need to be able to set variables, but they’re pretty tricky to figure out. I won’t bother talking about the class method for storing variables since we didn’t use it and I’m rusty on how they work since I was mostly dealing with them last semester while trying to figure out what the heck was going on. But basically, in the screen you are trying to render, the proper thing to do (from what I understand) is to create a variable/setter combination using something called “useState”. This kind of terminology is probably left over from the class method of doing things because you would set the state variable of the class whenever you wanted to save data, but useState is a “hook” that is used in the function method of creating and storing components. Here’s an example. “let [x, setX] = useState(0)”. ‘x’ is the variable that can be referenced anywhere in this screen. If you want to set x to like 7 or something, you don’t just say x = 7, you would say setX(7). These variables can be anything, strings, ints, floats, arrays, etc…. You can just set the “type” per say in the constructor of the use state. If you pass empty brackets, it knows it’s an array. If you pass empty quotes, it’s a string. (Though js is dynamically typed so does it really matter?) I literally just realized that instead of all this, you can say let x = 0, and then just set x, but from what I understand, that isn’t the “proper” thing to do. I don’t really know why setting the state is necessary when it seems so much easier to just say let x = 0 and treat it like a normal variable instead of having useState return the variable and a setter function, but whatever.

While I can’t speak for the necessity of the useState functions, I can guarantee that useEffect is necessary. It sort of acts as your constructor for the screen. If you want to display a bunch of requests that you are gonna pull from the database, you pull them from the database inside the useEffect function and set them to some local variable that you can use later. Some screens like individualRequest have very complex useEffects because a lot of work needs to be done to set up all the variables and dropdowns and such. Other screens like allTags have very simple useEffects because all we have to do is load the tags. Easy. Lastly, some screens like requests have a useFocusEffect instead of a useEffect. This essentially does the same thing but it runs every time the screen is focused instead of whenever the screen is rendered the first time. This is important because we were having problems where the user would maybe update a request subject or something and when they hit back, it wouldn’t render correctly on the request screen since the screen “remembered” what the old value was and didn’t know to re-render. Frankly, as far as useFocusEffects go, I really don’t understand all that’s happening. There’s several lines of boilerplate code that I just copied offline somewhere. The return, the unsubscribe listener, the callback, idk, it’s all complicated. But what I do understand is that all the code that was in the use effect goes inside the listener’s callback function (I’ll talk about callbacks in the next section). Every time the listener notices that this screen is focused, it’ll run the code within the callback. Lastly, a huge issue we had last semester was with the second parameter to the useEffect. It was empty brackets. That’s not important right? If we just pass nothing, that’s the same as passing “nothing”. Well, it turns out that that if you pass [ ] ,<-- this kind of nothing, then the useEffect knows that you don’t want to run the code inside the useEffect 60 times a second, but if you pass nothing instead, it assumes that your initial render and rerender function are the same and it’ll run the code in the useEffect *often*, to say the least. Which doesn’t break anything, I might add, it just means the phone battery is going to suffer. So, if you want to rerender on like a focus or something like that, use the onFocusEffect. If you just need to render it once since the user can only get there once (i.e. they can’t hit the back button to get to that screen) then just use the useEffect and pass [ ] as the second parameter.

Side note, I just remembered why useStates are important. Whenever the state is updated, the render function (which is kinda abstracted away from us) is called again, so if you have like a counter displayed, or literally anything that needs to be rendered, it’ll have to be stored in a variable that will call the render function. So yeah, we did the right thing. :)

*Understanding Callbacks and BataDases*

The last big thing you need to understand is callback functions. React Native uses Javascript, which is asynchronous, which has proven to be the most unhelpful thing in all the history of unhelpful things that have been invented. Please never build an unnecessarily asynchronous language. Anyway, the single handedly most difficult problem we solved during the first semester of senior design (in my opinion) was figuring out how to elegantly get data out of the database. The difficulty is that the database call is asynchronous, so we can’t just say “let requests = database.getRequests()”. I can’t remember what my original workaround was, but I know I had a plan B for if I couldn’t figure it out that was really gross and super not the right way to do things. But then I ended up discovering callback functions. Callbacks essentially say, “whenever you’re done running your function, call me”. So you can have a callback function that just sets a variable. In our case, we can say, “database.getRequests(setReqsCallback)”, and then in the database function, we can pass the results to the callback function. Something like “setReqsCallback(results)”. It works really well, it should be pretty easy to follow in the code. I did all the hard work of figuring it out. If you ever see a function that has an arrow in the parameters, it’s probably a callback function.

**Screen Enumerations...**

A lot of the stuff I’ve talked about up until this point has been general stuff that applies all throughout the app. For example, function based components are just the way we do things. Accessing data that was passed through the screen above is done by saying route.params.whatever. These are things that we spent most of the first semester figuring out but that hopefully, you can get up to speed just by reading this document. In the next section, I am going to talk about a lot more of the specifics. For example, how does the individual request page handle tags? How does the request page know what requests to load? How does the all requests page filter the requests? Hopefully between the section before and the sections to follow, you will have an idea of how react native works (sections before), how we used it (sections before and after), and how our app works (sections after).

*Welcome screen*

This screen has a couple unique things. One is that we have to request permission to send notifications to the user. Since this is the first screen users see, it made sense to put it here. This is also one of 2 screens where we ignore warnings. The other one is individual requests. All it does is suppress the log message. As far as we can tell, the warning on this screen is coming up because of one of the libraries we used. Apparently they access a deprecated global variable or something like that, but since the app was still working fine, it was easier to ignore the warning than try to hunt down where the problem was and fix it. Aside from that, the useEffect just creates the database if it hasn’t been created before. Then the user can click on the “let’s start praying” button or whatever and it’ll jump to the next screen. As a side note, we do have a hidden debug screen that can be accessed by clicking on “a prayer reminder app to strengthen your quiet time” 10 times. From that screen the user can do things like populate the database with dummy prayer requests for testing purposes, clear the database, re-create it, delete all the scheduled notifications, and there’s one other button that honestly, we just used for debugging queries or something where we needed a bit more control and triggering it with a button press was the easiest option.

*Dashboard*

This screen is also not too special. It just has the 4 main buttons that jump to other screens, and the pullout menu that also allows you to jump to other screens. The use effect calls a function named “checkBooks”. Most screens call the same function and all it does is check how long it’s been since requests were scheduled. Then it reschedules them if it wasn’t within the last day. It also archives requests that have expired. The reason we call this function on as many screens as possible is because we don’t want to assume that the user will close the app at the end of each day. If by some miracle, they manage to not close the app for 2 weeks while still using it consistently, they will still get notifications even though we only schedule 2 weeks ahead because every time they change screens, we check/reschedule notifications as needed.

*Categories*

This is the first screen that has any complexity to it. The useEffect which can often be quite involved is still pretty simple. It just populates the different categories or journals with the category names from the database and makes the time picker visible for ios, though I will get to the time picker in a minute. What the user will see is a list of Journals which we call categories in our code. They can press one of the categories and it will jump to that category and display the requests in that category. We pass the name of the category down so we can keep track of where we came from, which is useful once we get all the way down to the individual request screen. If instead of pressing on a category, the user were to long press or tap the 3 dots on the side, the user would see an edit popup. The edit popup allows the user to change the category name and times/days that requests from that category will be scheduled. This is where we go back to the time picker visibility from before. Android does time pickers as a popup (at least with the library we used) but for ios, it’s just sitting there, it isn’t a popup. For a similar reason, we have a “display time” on android. When the user picks a time, it’s actually stored as a date. We just parse the time out of the date, convert it to a string, and display that string on a touchable opacity. Then when the user touches that button/touchable opacity, it triggers the time picker popup. All that to say, you’ll see little android/ios specific code snippets but the handlers are the same for both platforms. The code gets a little hard to follow. I wish we had been more consistent with the styles. Inline styling convolutes meaning. Try not to do it. A decent bit of the complex stuff is in the modals down around line 340. The modal is just a popup. If the user hits the save button, edit button, delete button, or changes stuff like the time or what days of the week are selected, that’s where we have to do a bunch of work. When they save, we have to do database calls to make sure we save the new name of the category *and* the tag. If they tap on a day of the week, we have to change that day in the use effect so that it triggers a re-render and displays correctly, and we have to save that new string of days to the database whenever the user hits save. If, (heaven forbid) they decide they want to delete a category, we decided that the best thing to do is to archive the requests in this category that don’t show up in another category. That turns out to be quite a nasty query, but it’s not too bad of a function call, so you’ll see that too. Though we do have popup “alerts” to warn the user that they might not have meant to click that button. We also have to refresh the page so that the category actually renders as gone now or else the user would be confused. We refresh the page by having passing a callback to the delete/archive query. The callback just reruns the queries from the use effect. 2 more notes. We pass cat\_id down and use it to get the requests in the appropriate category on the requests screen. Secondly, we might have some redundant variables. For example, selectedCategoryName is equivalent to selectedCategory.name, so it doesn’t make a lot of sense to have both, but we’re really close to a code freeze and I’m working on documentation and I’m afraid of breaking something that I won’t have time to fix, so….yeah, sorry about that.

*Requests*

We use a “useFocusEffect” instead of a useEffect for this screen to fix a small bug having to do with going back from individual request after changing a request subject or something. With just a normal useEffect, the request subject wouldn’t update when going back to the requests page but now it does. In the useFocuseEffect, we just grab all the requests in the category whose id is equal to the cat\_id we passed through the parameters and display them in a flatlist (flat lists are great, but something that Yayira should talk about, so hopefully you can find out about them in her brain dump). We have a popup on this screen too which allows the user to archive or delete a request if he so pleases. I’m just realizing it’s kinda hard to find a balance between explaining what the app does which you can just test on your own pretty easily and telling you the details of how we made it do that. For example, on this page, the main difficulty has to do with making sure the page refreshes after we delete/archive the request. Otherwise, the user will be confused about why it’s still displaying (even though it’s actually been deleted). But then I tend to over explain and tell you about every single thing that the app does. Eh. I’ll try to only tell you what you need to know to get up to speed. So, the way we originally (and in this case) refreshed the page is to just wait a few (200) milliseconds and then run the database queries again. We figured that 200 ms was enough time for the delete/archive query to run. On later pages (such as categories) we actually use callbacks, which is the safer, more proper way to do things, but we’re out of time to change stuff.

*IndividualRequests*

On this page, we have 2 more warnings that we are ignoring, again, having to do with deprecated libraries. Supposedly, classes will no longer be supported in 2022, so I don’t know if that means our app will break or what, but it was still working and only threw a warning so we worked with it. The first warning looks like it has to do with class components, the second warning, I don’t even know. If I remember correctly, that had something to do with the dropdown we were using? But I really don’t know. It still works. I’m going to try to only explain how we populate stuff, there’s no need to go in depth about what everything looks like and how it works, you can already see that.

So we start out by getting the request out of the database based on the request id that was passed down from the page above (on the navigation.navigate… line). We have to set variables like the subject, description, priority, and expiration date. We handle the expiration date similarly to the reminder times in categories, except this time we actually care about the date, not just the time. But as far as android and ios being different, having to parse out the android date and making a popup pop up when clicking on the button, all that’s the exact same thing. So we populate all those variables. (We populate them differently depending on if it’s a new request or not. New requests, either from a qr code or the new request button will come in with some parameters of their own, but also we set some ourselves, such as the expiration date. We just set it to be 2 weeks (or 2592000000 mili? seconds) from the current date). The priority is stored as an integer between 1 and 3 inclusive. (I originally went from 0 to 2 but if I recall correctly, Noah’s reminder code was multiplying the priority or adding it to something else to calculate what requests to schedule and going from 0 to 2 broke stuff.) Oof, here’s where I really hope I knew what I was doing and why. Because the next thing I do is check again if it’s a new request. If it is, I just get all the tags out of the database so the user can pick which ones he wants to associate with the new request. If it isn’t a new request, I’ll get all the tags associated with the request, and within the callback of that function, I’ll get all the tags that exist. I don’t know why I had to do things in that order. It seems like I could just say get all the tags. Then, if it’s not a new request, get all the request tags, but...I remember switching to doing it the way that I did for a reason. Probably if I tried to switch it back again I’d remember and be able to tell you.

For keeping track of which tags are selected and which tags have changed, I have 2 arrays. One to store the current value, and one to store if this is a request that has been changed. Let’s say this is a new request, so no tags have been selected. So the user taps a tag, let’s say “family” for the sake of example. Now I set the value of the family tag to true (I probably set it to tags[x] = !tags[x], but same difference) so that it renders as clicked and I set the value of changed to true as well, so I know to add a request tag that links family with this request whenever I hit save. This is all handled in handleTagPress or something like that. I just mention it so that when you see a bunch of arrays being built in the callback functions for getting the tags and request tags, you know what’s happening. Lastly, we get the categories so we can populate the dropdown.

There’s also quite a bit of work happening after the useEffect. For example, the tag press handler also handles making sure that a category has been selected, since categories are how requests are organized. Or, once the user has selected a category, it won’t let the user unselect that tag. The other big function is saveChanges. I think it’s pretty intuitive. It mostly involves setting all the fields of the request and either updating or inserting it (depending on if it’s a new request or not). Then updating all the tags as well. I added an inline function to update the tags. Then I pass it as a callback function into the update queries. Ah, one other interesting thing. This whole page is set up into two parts. One is edit mode, the other is...idk, not edit mode. Most of the display react native code is very similar, it’s just that we don’t have onPress handlers for the “not in edit mode” whereas we do have handlers for edit mode.

*Prayer Time*

I don’t think there’s anything special about this screen. We just grab the requests out of the special table in the database. We do have to do a little custom filtering because of complexities with how databases work. The rest is pretty intuitive. The “isPrayedFor” boolean gets updated on press. Tapping a request just runs navigation.navigate on press. Etc…

*QR Code SCanner*

This page is pretty short and simple. It just uses a qr code scanner library that we found to scan qr codes. Then the handler checks to see if it’s a valid go pray qr code (we just check to see if the first 5 characters are cupray...sad.) Then we assume that the rest is correctly formatted, read it in as a json object, and pass the appropriate data (subject and description?) to the individualRequest screen (or the multi scanner screen).

*MultiScan*-*All Tags*

Yeah, there is really nothing special about either of these two screens. They work exactly like you would expect them to. The only thing I’ll mention is that you might have difficulty getting to the multi scan screen if you don’t have one of those kinds of qr codes, so here’s one to save you the trouble.



*All Requests*

This is the last significant screen. I didn’t talk about stack nav or temp dash or maybe a couple others but I got the main important ones. The main thing I need to talk about on this page is the search bar. It was kinda fun to get this to work. Basically, we have a query (2 actually) that looks for requests that have categories *like* whatever text is in the text input. Then, we run that query every time the user presses a key. The query we run is dependent on the state of the checkbox. If the box is pressed, the version of the query that allows archived requests is run. If the box isn’t pressed, the version of the query that filters out archived requests is run. We also do a little manual filtering on this page with a hash map thing or something like that which I copied offline and named remove dupes.

One other kinda goofy thing I had to do/chose to do is keeping a map of all the requests. This is because I usually pass the category down whenever we go to the individual request screen, but the filtering query does not associate requests with categories, only with tags, so we had this bug where clicking on the request would go to it, but it would show the tag as the category, which is bad. So, we have all the requests in a map indexed by request id so that we can just grab the request that we need, grab it’s category (if it’s in multiple categories, it’ll probably grab the category that shows up first in alphabetical order).

I didn’t talk about the new tags button