Native Share Menus - Share to New Expensify

AUTHORS
Georgia, Lizzi @ Infinite Red

SLACK ROOM
#expensify-infinite-red

DEADLINE
N/A

TRACKING ISSUE
[Github link]

Strategic Context

The only way to get a billion users on our platform is to leverage the natural viral dynamics of collaboration. Currently, they do this via WhatsApp, SMS, and other chat tools. We need NewDot to be an effective replacement for those tools in all of their core flows.

High-level overview of the problem

There are plenty of times when you might want to share something from outside sources into New Expensify. But there's no way to quickly share links/photos/information. For example, if you are in the Photos App and decide you'd like to share a photo, you'll have to leave the app → open New Dot → go to the correct chat → click add attachments → find the same image again → and so on. That adds significant friction and makes it less likely that users will default to Expensify as their go-to chat app.

Timeline and urgency

There is no immediate deadline for this project. However, this project is a requirement for Reunification. In Old Expensify, you can share photos to initiate a smart scan. To replicate this functionality of our previous platform, we first need to add support for sharing into the app. Once we have that, we will have a baseline to add support for initiating a smart scan.

Terminology

Native Share Menu - the native UI that pops up when a share is triggered. This includes the list of apps that are available to share into.
Native Error Modal - the native modal that pops up on iOS if there is an error with a share. For example, if the user is not logged into the app they are sharing into.
Share Modal - The custom UI that we are adding for the Sharing flow.
Share-to-Expensify - The user flow while attempting to send something from outside the app to NewDot, such as sending someone a photo from the Photos App.
Share-from-Expensify - The user flow when attempting to share something in-app to an external source, such as sending a QR code to iMessage.

High-level of proposed solution

[Pre-design] / [Design Pre-design]

Let's add New Expensify into the Native Share Menus for iOS and Android to allow sharing attachments and links into New Expensify from outside of the App.

We will allow users to share two types of data: links, and attachment files (photos, PDFs, etc). (slack)

When a share is initiated and NewDot is selected, a custom share modal will pop up. This modal will have two pages. The first page will feature our search component and allow users to search and select the destination of the share (#room or individual/group). The second page will feature a text input and allow the user to type a message to go along with the link or attachment.

While we want to avoid platform-specific code in New Expensify, we decided that we will mimic the default Share flow in both iOS and Android (slack). Those flows can be defined as:

iOS:

1. Native Share Menu is shown, user selects New Expensify
2. The Expensify Share Extension, which you can think of as a mini-app separate from the main app, is shown, containing the custom share modal screens
3. After completion of share, the Share Extension is closed and the user is dropped back into the app they were sharing from.

Android:

1. Native Share Menu is shown, user selects New Expensify
2. The New Expensify app is opened and routed to the share modal screens within the app

3. After completion of share, we navigate to the Expensify app screen showing the chat that was shared to

In iOS, while sending a link or attachment, we will wait for the upload to complete before closing the modal. For attachments that require an upload, we will show a visual indicator to denote the share is sending and then close the modal when the share is complete. For Android, because the share action opens the app, we can use the existing loading UI. (slack).

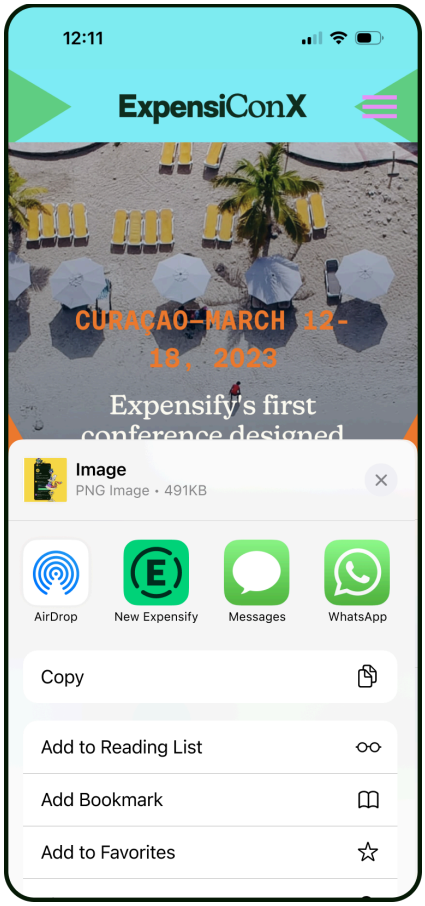
If a user adds a message to an attachment, the resulting chat will appear in New Expensify as the message first and then the attachment on the next line (as a separate message). (slack)

If there is an error that occurs during this process (for example, the user is not logged in), we will use the native error modal for iOS to alert the user. In Android, we will just open the app to the sign-in page. The user will then need to reinitiate the share after signing in.

UI Additions & Changes

App Icon in Native Share Menu

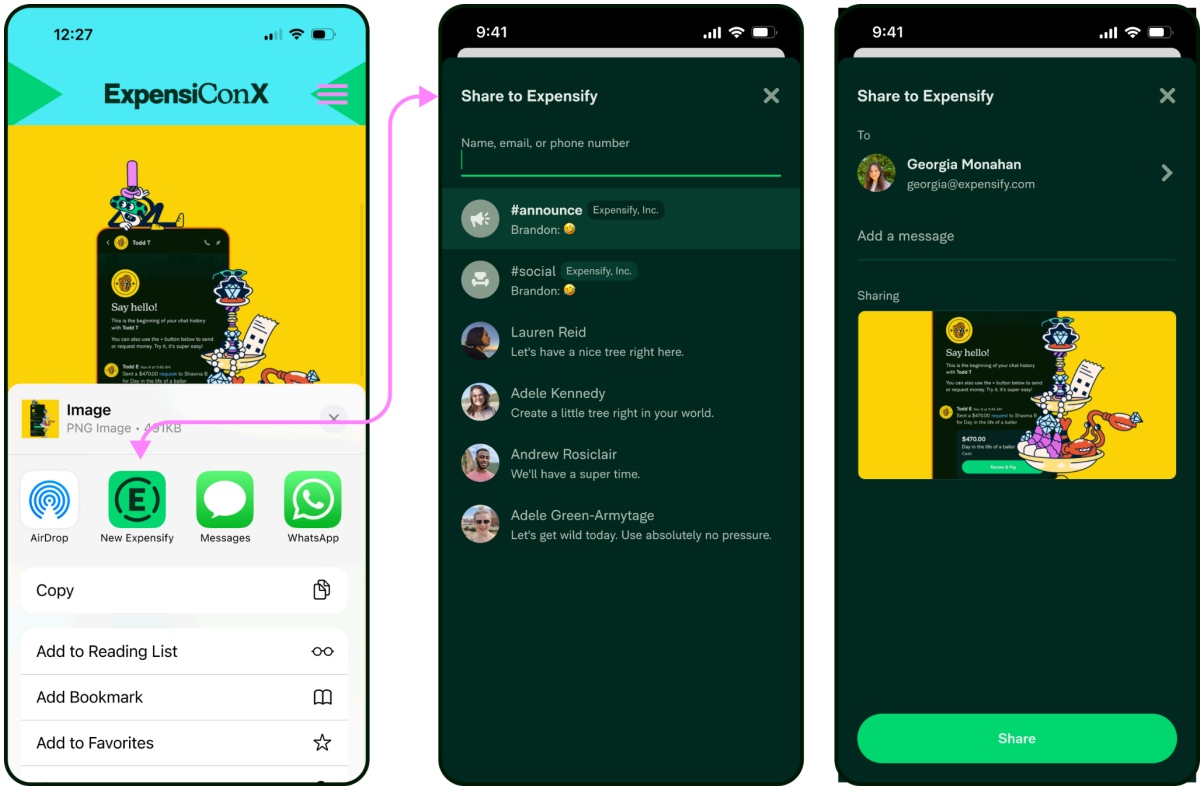
The icon for New Expensify will appear in the Native Share Menu.



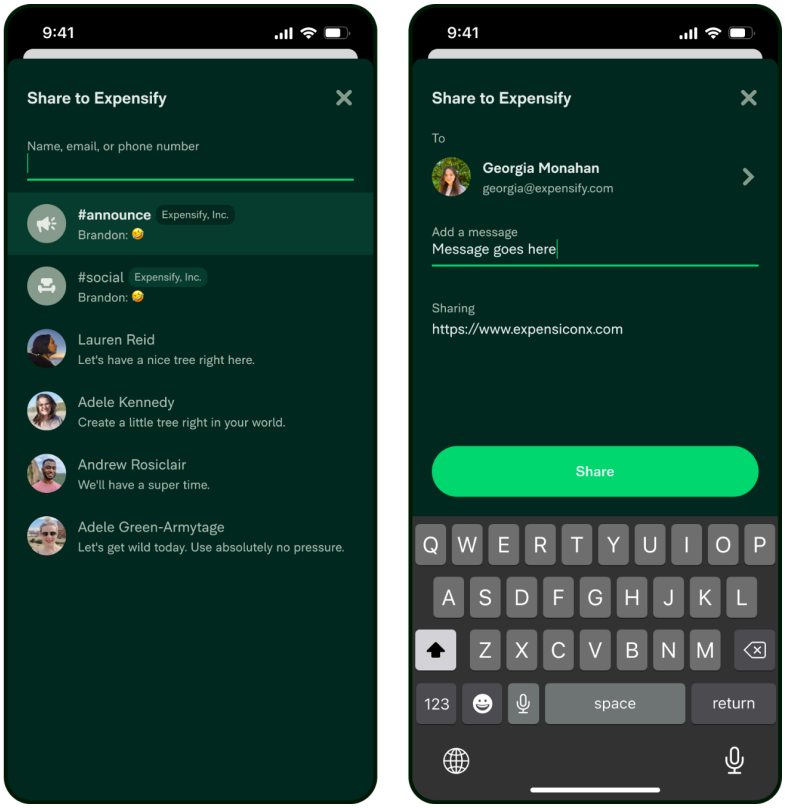
Custom Share Modal

We will create a custom Share modal. It will consist of two pages, one to select a destination and one to add a (optional) message. It will also feature an “X” button to allow the user to close the share modal.

For sharing an attachment (on iOS):

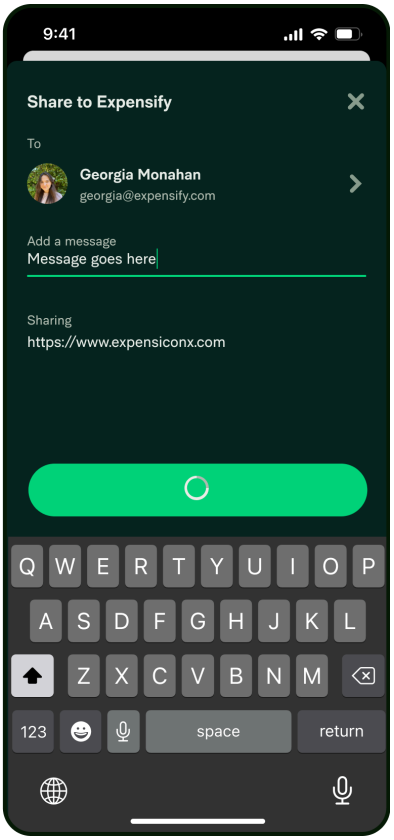


For sharing a link:



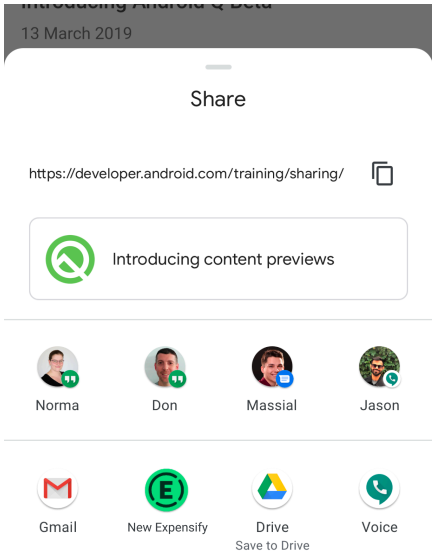
iOS Loading

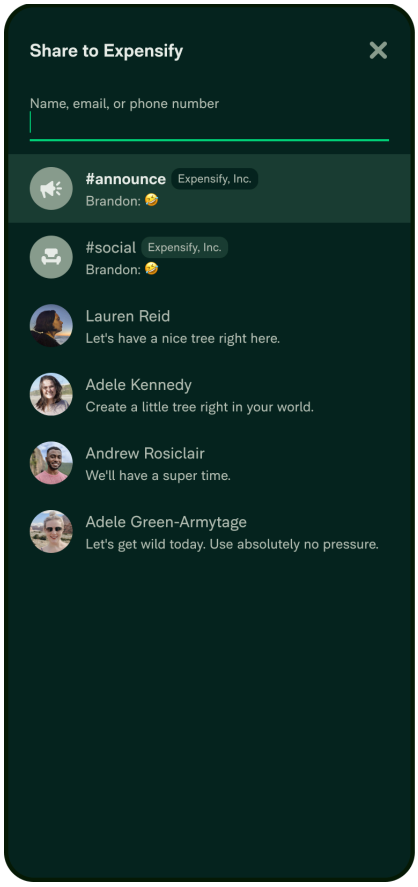
Because the loader will only show for a split second, we will leave all of the UI and show the default spinning loader inside the send button while the photo/attachment uploads.

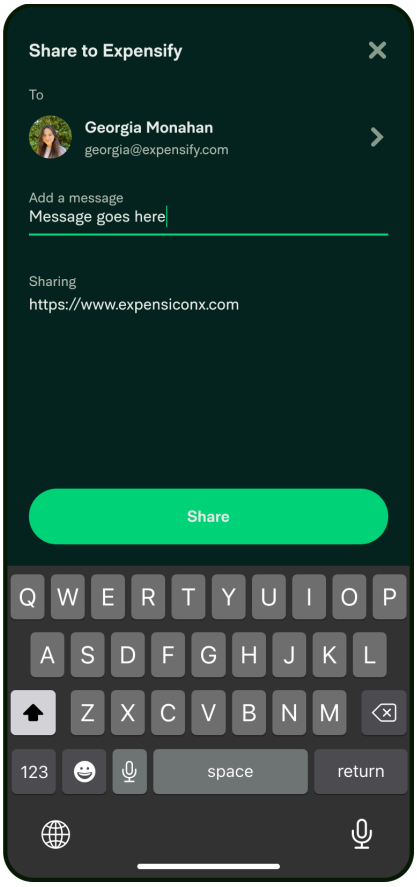


Android Distinction

It is important to note that for Android, the Share Modal is opened within the app. Therefore, we will use the same design, but it will fill the entire screen. If a user clicks on the ‘X’, they will remain in New Expensify.







Expensify.com / new.expensify.com

NewDot specific. Detailed section and implementation will be worked on by Infinite Red.

Data storage

No additional Data is stored.

Economic considerations

Are there any specific costs associated with your solution? If so, are these costs fixed (e.g. the cost of building a lounge), or are they variable based on usage (e.g. no. of API calls)? N/A

Does Expensify have the ability to spread payments out over time? And is there a cost associated with doing so? N/A

Are there aspects of the engagement that enable Expensify to only pay once a certain task is completed (e.g. onboarded a company and a billing card is on file)? N/A

How long will Expensify be committed to these costs? What risks might be associated with this length of commitment? N/A

On the flip side of all of this, what does Expensify stand to gain via your solution (e.g. interchange, active user revenue, etc.)? N/A

Accounting Implications

Does this require a new vendor? No
Vendor Name:
Vendor Contact Email:
Service Period:
Estimate spend:
Invoice Frequency: *Daily*
Expense Category and Department:
Require 1099: *No*
Is this related to a physical asset (lease, equipment, etc.)? *No*

Does this change relate to revenue, discounts, expenses, commissions, bonuses, active users or any other inward, outward or internal movement of funds? Do we need to start counting these revenue generating users as *Paid Members?* How is the accounting team addressing this change? Unknown

Does this change require moving money through Stripe, ECard, our ACH system or any bank account? Unknown

*Note to doc authors: Please make sure to **get at least one review from someone on the accounting team before moving to the detailed portion**. See related SO here. Thanks!*

Reviewed By

2023-04-24 CJ

If applicable, link the disclosure document here.

Legal and Compliance considerations

Does this require an update to ToS and/or privacy policy?	No
Does this justify any additional controls in our regular SOC audit?	No
Does this impact PCI?	No
Are there any possible trademark considerations?	No
Could anything about your design be patented?	No
Will you be working with a new vendor and signing a contract?	No

Risk Assessment

We need to evaluate every project and every proposed vendor for potential risk.

For every project that requires a code/hardware change:

- Create a GH with the title `projectName Risk Assessment`. Apply the labels `Task`, `Ring1`, `Compliance`, and an appropriate KSv2 option.
- Include a link to your design doc and any related research or notes you've already compiled.
- Infra will then follow this process to assess information change risk.

https://github.com/Expensify/Expensify/issues/277613

*Note to doc authors: Please make sure to **get at least one review for each section before moving to the detailed portion**.*

Out of scope considerations

- In the first version of this project, we will only support clicking on the App Icon to initiate a share (as opposed to being able to click on specific recent chats).
- We will add sharing to recent DMs or rooms in the future, but it was decided to keep it simple and share to App Icon only for the first version of this feature. (slack)
- Once receipt scanning is added to New Expensify, we will update the attachment flow to support share-to-scan-receipt. (slack)

Alternate solutions

- We discussed standardizing closing the share modal and remaining in the app you shared from (iOS) or opening New Expensify after a share (Android). It was decided that because this occurs outside the app, we will follow the default for each OS.

High-level overview reviewed by

Authors:
If you've made it this far in your design doc, now is the time to pause and ensure you've added your project to the CAP Sheet.

Please make sure to **get at least two reviews from each G&R tier before moving on** to the detailed portion. Please follow this SO to guarantee reviews by applying the DesignDocReview label to your tracking issue.

Reviewers:
After you have thoroughly reviewed this doc, add your name and date in the section that corresponds to your Growth and Recognition tier.

Expensifiers + Graduates
2023-04-17 - Rory - Looks good to me. I feel good about the scope-related choices here 👍
2023-04-18 - Jules- looking great, personally I'd like to see us implement contact-specific share as part of this project so we can review the full details
2023-04-21 - Caitlin
2023-05-01 - Ariel
2023-05-02 - JLi - Awesome!

Project Managers
2023-04-13 - Stites - Super excited for this!
2023-04-21 - Greeny

Product Managers
2023-04-17 Steph E

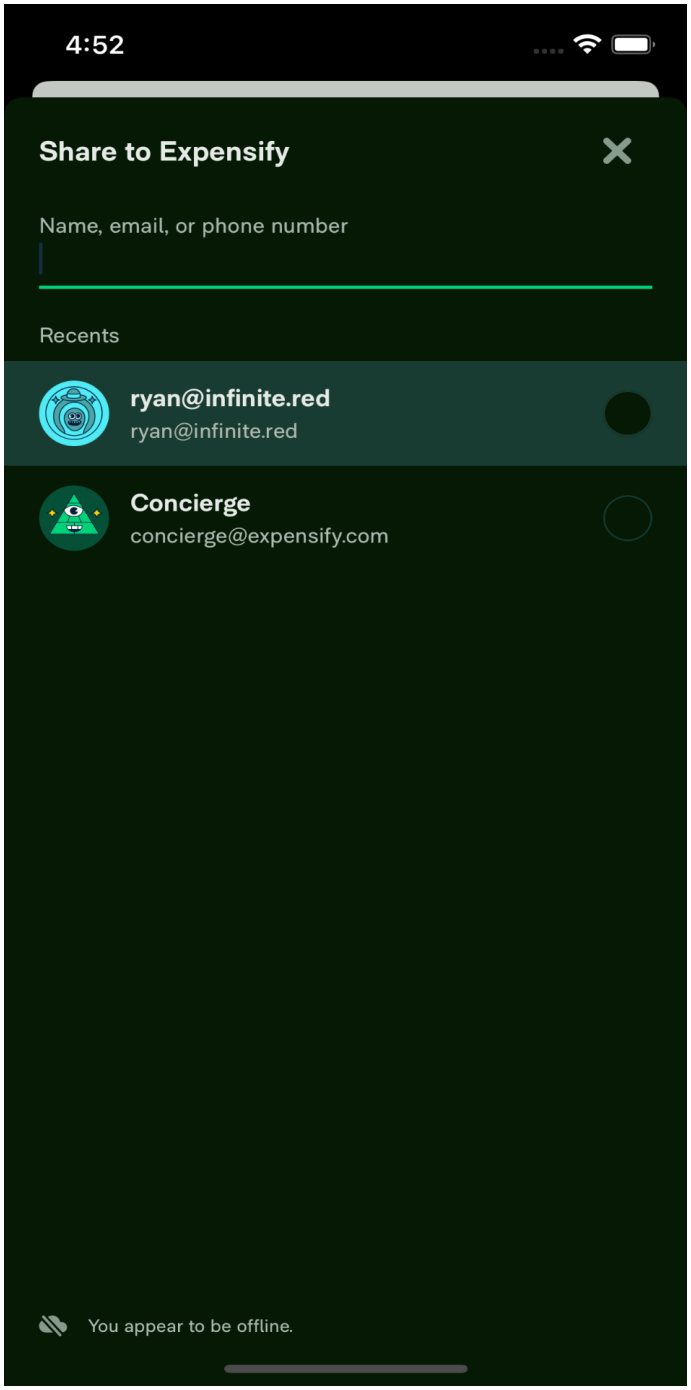
`2023-04-19` *- Conor P*
`2023-09-13` *- Robert C.*
`2023-10-24` *- Rocio*

Generalists
2023-04-25, 2023-09-04 - puneet - seems straightforward!
2023-04-28 - anu
2023-05-03 - @dbarrett - Looks great, but let's combine the message and link when sharing a URL.
`2023-05-04` - `Jason Mills` *- Really excited for this. Use this flow all the time.*
2023-09-07 - Yuwen - This is nice

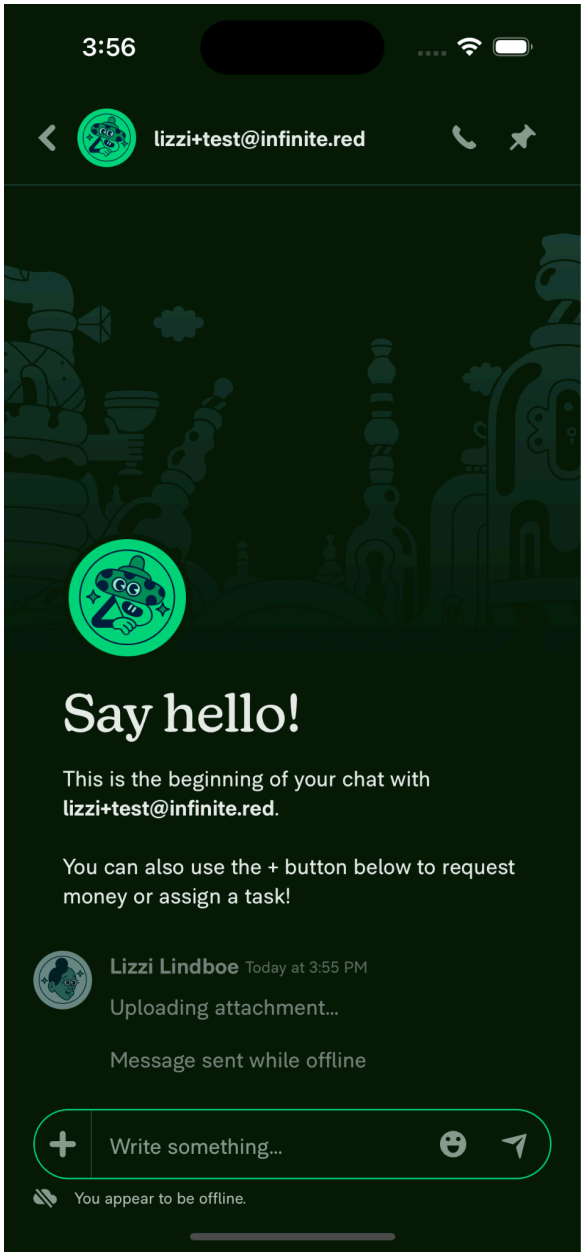
Offline support

Users will still be able to share offline. If commands are not sent before the share is complete and the user leaves the app, the share will be sent the next time the user opens the app while online.

We will indicate the user is offline by using the offline indicator at the bottom of each screen, as is already the case for the new group page, shown below rendered inside the iOS share extension.



When looking at the conversation that they shared to, users who have remained offline since sharing will see the same UI as when sending a message or attachment while offline otherwise:



On iOS, since the share extension is closed immediately after clicking “Share”, messages shared while offline will not be sent until the user opens the main app again while online.

Detailed background

Content types

Both iOS and Android have systems for configuring what types of data can be shared to the app. This controls whether New Expensify will show in the system share menu.

iOS requires configuring an `NSExtensionActivationRule`. There are presets for certain types, and iOS also allows custom predicates to decide if a file or other data is supported.

Android is configured using a list of mime types, where wildcards are also allowed. For example, `image/jpeg`, `image/*`, and `*/*` are all valid items in the configuration list.

Android Send Intents

Android uses "intents" to message requests between different apps and their components. To receive shared data from another app, we configure our app to respond to the `SEND` intent with an "intent filter".

iOS Share Extensions

In order to be able to share into an iOS app, that app has to implement a specific type of app extension, called a share extension, which is a completely separate executable from the main app. (This is in contrast to Android, where data can be shared directly to the app).

Share extensions were designed to be quick to spin up, send data off, i.e., to a web API, and then exit. They're not designed to send data directly to the local app (although this is possible).

Share extensions should be quick to open

It's important that share extensions open quickly, since the user is performing a quick task. To enable this, we should try to only load in JavaScript code that's necessary for the share extension to run (i.e., the entry point should not be the same as the main app).

Share extensions are a separate executable

Share extensions, like all app extensions, are built as a separate executable. This means that, by default, they don't share code or data with the main application. But there are mechanisms for sharing information between the two, such as app group directories (described below), static linking of code (described in alternate solutions), and the NSUserDefaults API for smaller amounts of user data.

App groups share data between the app and its extensions

App groups can be configured to contain the app and its extensions, and provide shared storage. We can move data like the Onyx database to this location to allow both the app and its share extension access.

Sharing configuration between the app and its extensions

App extensions are configured in nearly the same way as the main app, which can mean duplicating configuration that's managed by the React Native template and the Podfile.

For example, in XCode, the deployment target is set separately for the main app and its extension, but it doesn't make sense for the app and the extension to have different deployment target values.

Share extensions increase total app size

Since app extensions are their own executable, by default they'll have each have their own copies of relevant code. We looked at whether we might benefit from optimizations, such as manually linking only needed native code, or refactoring JS to keep bundle size small, but impact was minimal. See "alternate solutions" section for details.

Detailed implementation of the solution

User experience details

The user experience will vary a little depending on the context, but the general steps are these:

1. User is navigated to the group creation page, and selects people to share to
2. User clicks the "Create group" button.
3. User sees new screen `ShareMessagePage` where they will see the shared content, and can edit an optional message
4. User clicks the "Share" button to send the attachment.

Unlike the main app, we will *not* show the splash screen for the iOS share extension. We've found it takes longer for the splash screen to render than the React Native UI.

The following sections will describe how the user experience can differ depending on context.

iOS vs. Android

On Android, the share flow will just be modal screens within the app, and when the user clicks "Share", the modal will be closed, and the user will be shown the report they shared to.

On iOS, after the user sends the attachment, the share extension will be dismissed and the user will be returned to the app they were using when they started the share process.

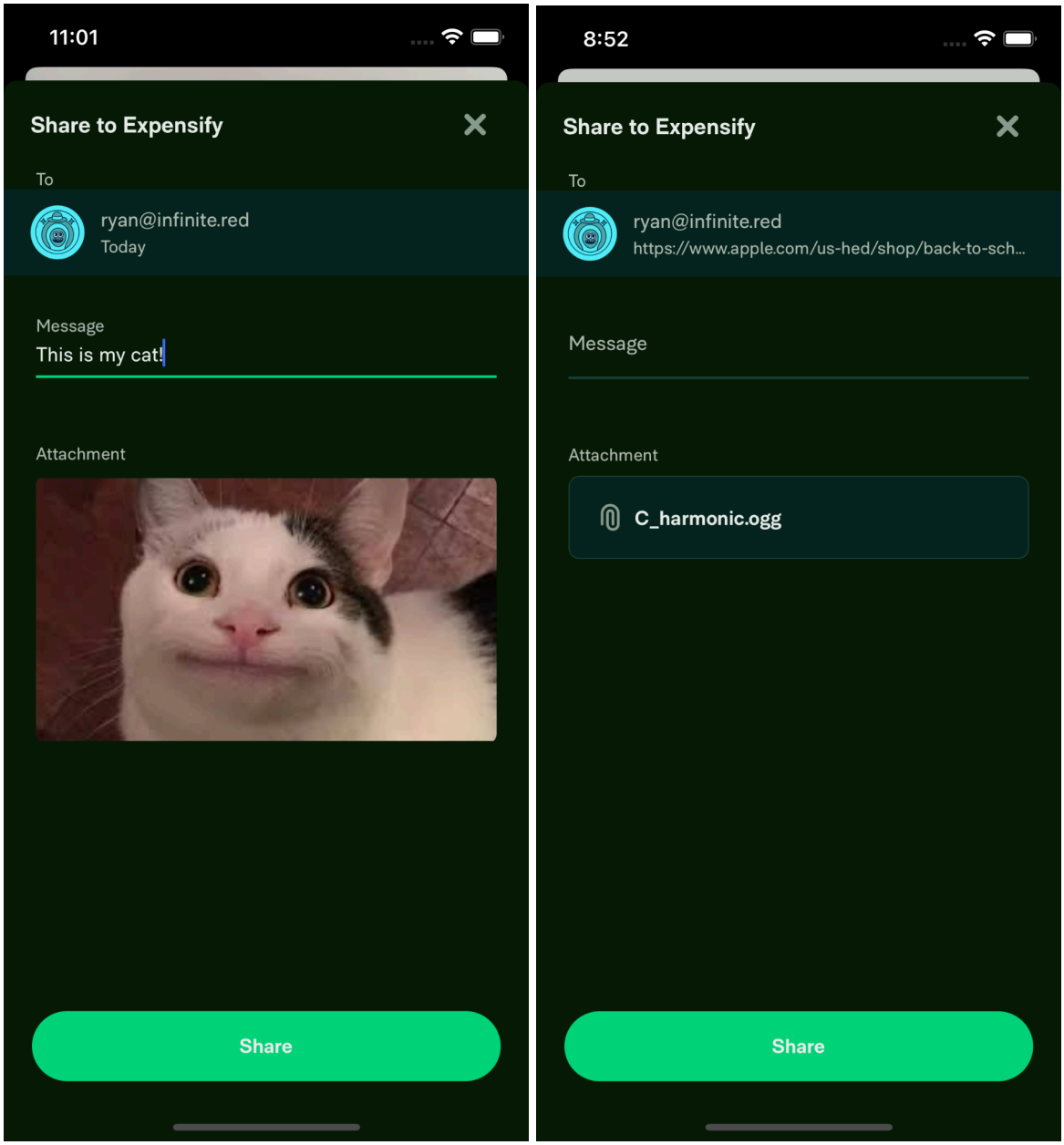
In addition, on iOS, the screens will be presented as a modal that allows the app that was shared **from** to be visible underneath, at the top of the screen.

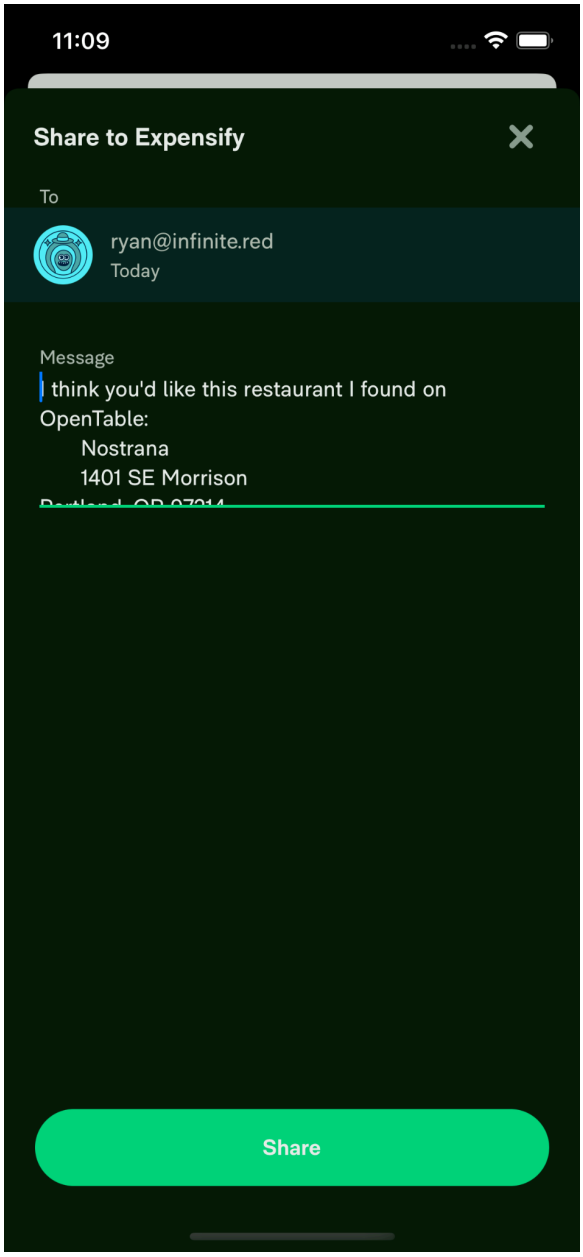
Content types

Content will be handled and displayed differently based on whether it falls into three different buckets: a previewable file, a non-previewable file, or text.

Whether a file is previewable is determined by the existing logic in the `AttachmentView` component, which we will re-use here.

If the shared item is text, such as a link or a message, it will be added to the message field instead, which will auto-grow for multiple lines.



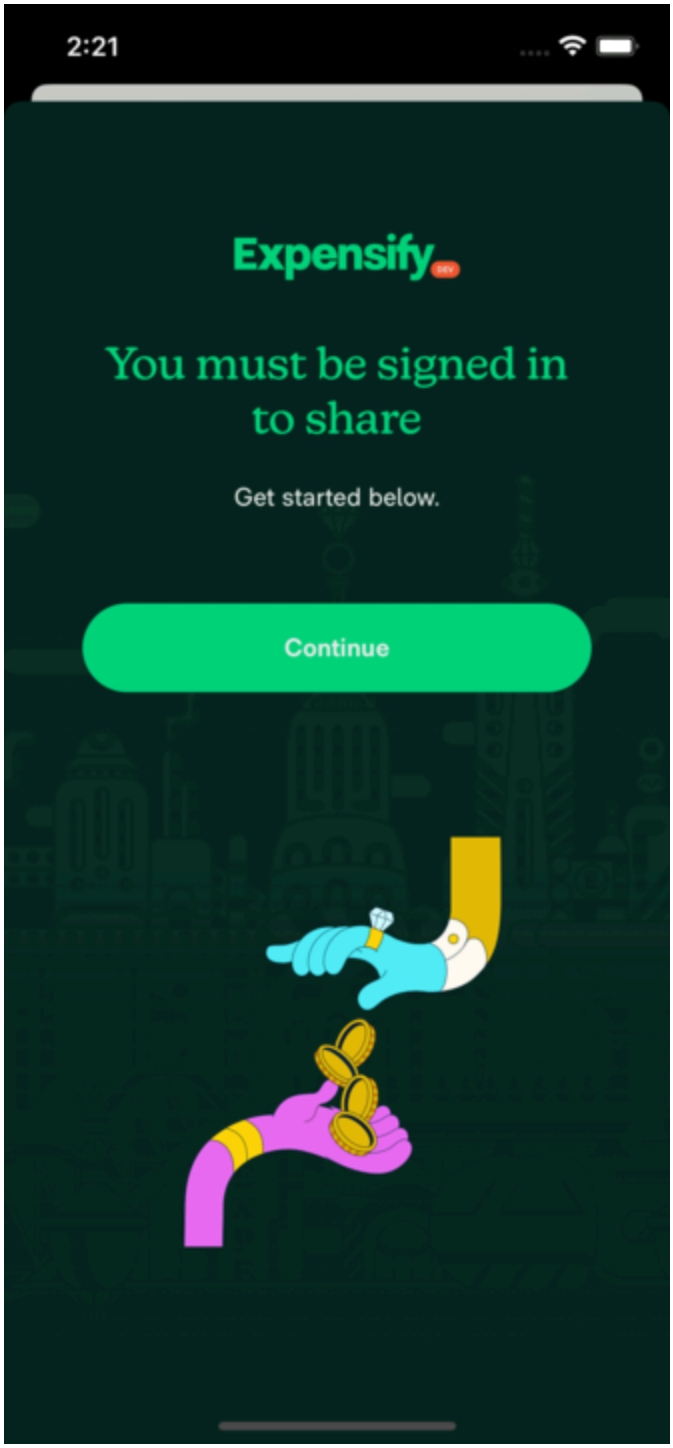


Unauthenticated user

If a user has not yet logged into Expensify when they try to share, we'll have them sign in first and try again afterwards.

For Android, they'll simply see the login page and sign in as usual. The user will then need to navigate back to the app they were attempting to share from and restart the process.

For iOS, they'll see this UI in the Share Extension, which will link them to the main app when they click "Continue":



They'll be able to sign in on the main app, then restart the share flow from the other app in order to try to share again.

Errors

Since we will be running the full app inside the iOS share extension (see “Entry point” section below), we will also have the Error Boundary available to catch and log unexpected errors.

Libraries

react-native-share-menu

The react-native-share-menu library is the best fit for our needs of any existing tools; it has existing support and documentation on how to add a custom React Native view to an iOS share extension, and tools for handling incoming share data to the Android app.

On iOS, it provides the native code needed to host a custom React Native view inside of a share extension. It also provides a way to pass data directly from the share extension to the main app, but we won’t be using that feature.

On Android, it provides a Javascript API for listening for `SEND` intents, as a React Native event listener, and handling the data provided by the intents.

We were able to get it working with a few patches. However, it’s slightly out of date and hasn’t been maintained consistently. For example, there’s this open issue due to an API change from react-native. We have an open conversation with an existing maintainer for Expensify to adopt the project, and add maintainers from Expensify and Infinite Red so that we can get the library up-to-date.

We would also like to update the library with more features to support the desired sharing workflow in New Expensify. See the react-native-share-menu updates section for details.

react-native-fs

We’ll use react-native-fs, which is already included in the app, to handle the Onyx database location move needed for iOS. See Make Onyx accessible from the Share Extension for details.

Configuration

Content types

iOS

Using the `NSExtensionActivationRule`, we will be able to support files, text, and links:

```
Unset
<key>NSExtensionAttributes</key>
<dict>
```

```
        <key>NSExtensionActivationRule</key>
        <dict>
                <key>NSExtensionActivationSupportsImageWithMaxCount</key>
                <integer>1</integer>
                <key>NSExtensionActivationSupportsText</key>
                <true/>
                <key>NSExtensionActivationSupportsWebURLWithMaxCount</key>
                <integer>1</integer>
                <key>NSExtensionActivationSupportsFileWithMaxCount</key>
                <integer>1</integer>
        </dict>
</dict>
```

Android

We set Android to accept all file types for the `SEND` intent, which covers share requests of a single file or text string:

```
Unset
<activity
  ...
  android:documentLaunchMode="never">
  ...
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="*/*" />
  </intent-filter>

</activity>
```

Implementation

Both Android and iOS will use the main app entry point, adding conditions to react to whether shared data is present.

Receiving shared data from native layer

Android and iOS have different methods in the react-native-share-menu API in order to retrieve shared data from the native layer, but we can unify them:

```
JavaScript

// src/libs/Share/index.android.js
```

```
const registerListener = () => {
  // handles share attempts that occur while app is not running
  ShareMenu.getInitialShare(navigateToShare);
  // listens for share attempts while app is running
  return ShareMenu.addNewShareListener(navigateToShare);
};

// src/libs/Share/index.ios.js
const registerListener = () => {
  Navigation.isNavigationReady().then(() => {
    ShareMenuReactView.data().then(navigateToShare);
  });
  return {remove: () => {}};
};

// src/Expensify.js
function Expensify(props) {
  ...
  const shareListener = Share.registerListener();
  return () => {
    ...
    shareListener.remove()
  }
}
```

`navigateToShare` will take the shared data and pass it `NewSharePage` as a navigation param. Using navigation params makes it easy to manage share state without accidentally persisting it beyond the share attempt.

JavaScript

```
// src/libs/Share/navigateToShare.js
const formatShareData = (shared) => {
  const share = isArray(shared.data) ? shared.data[0] : shared;
  return {
    isTextShare: share.mimeType === 'text/plain',
    name: share.data.split('/').pop(),
    source: share.data,
    type: share.mimeType,
  };
};

const hasNoShareData = (share) => !share || !share.data || isEmpty(share.data);

const navigateToShare = (share) => {
  if (hasNoShareData(share)) return;
  Navigation.isNavigationReady().then(() => {
    Navigation.navigate(ROUTES.NEW_GROUP);
    Navigation.setParams({share: formatShareData(share)});
  });
};
```

Unified flow in JS

On the NewSharePage, the user will select who to share to, and pressing “Create group” will open the relevant report, and navigate to the new share page.

To accomplish this, we’ll add a new `navigateToAndOpenShare` action, mirroring the existing `navigateToAndOpenReport` action:

JavaScript

```
// src/libs/actions/Report.js
/**
* This will find an existing chat, or create a new one if none exists, for the
given user or set of users. It will then navigate to the share dialog.
*
* @param {Array} userLogins list of user logins to start a chat report with.
* @param {Object} share the share object to be passed to the share modal
*/
function navigateToAndOpenShare(userLogins, share) {
 let newChat = {};
 const formattedUserLogins = _.map(userLogins, (login) =>
OptionsListUtils.addSMSDomainIfPhoneNumber(login).toLowerCase());
 const chat = ReportUtils.getChatByParticipantsByLoginList(formattedUserLogins);
 if (!chat) {
  const participantAccountIDs =
PersonalDetailsUtils.getAccountIDsByLogins(userLogins);
  newChat = ReportUtils.buildOptimisticChatReport(participantAccountIDs);
 }
 const reportID = chat ? chat.reportID : newChat.reportID;


   // We want to pass newChat here because if anything is passed in that param
(even an existing chat), we will try to create a chat on the server
 openReport(reportID, userLogins, newChat);
 Navigation.navigate(ROUTES.SHARE_MESSAGE);
 Navigation.setParams({option: userLogins, share, reportID});
}
```

Then, on the `ShareMessagePage`, clicking the “Share” button will fire Report.addComment or Report.addAttachment, depending on the type of data shared.

JavaScript

```
        onPress={() => {
          if (isTextShare) {
            Report.addComment(reportID, message);
          } else {
            Report.addAttachment(reportID, share, message);
          }
          Share.dismiss();
         }}
```

End of flow behavior

The button will also call a `dismiss` function, which will have different implementations based on the platform, since iOS needs to dismiss the native share extension:

```
JavaScript
// src/libs/Share/index.ios.js
const dismiss = () => ShareMenuReactView.dismissExtension();

// src/libs/Share/index.android.js
const dismiss = () => Navigation.dismissModal();
```

Offline share support on iOS

No additional steps are needed to support offline sharing on Android, as it's the same as sharing any other file.

However, on iOS, we need to copy any shared files to a new location where the main app can access them later, as the temporary copy that the share extension uses does not last beyond the share extension's lifetime. We'll add methods for managing those files to react-native-share-menu, with methods to create a copy of a file and later delete that copied file.

Ensuring we clean up copied files

We need to make sure that we delete these copied files after the request to `Report.addAttachment` has either succeeded or failed (i.e., is no longer pending).

To accomplish this, we will add a new Onyx key, `tempFilesToDelete`, to both the success and failure data of `Report.addAttachment` that will contain the URIs of files to be deleted.

Then in the share library module, we will use `react-native-fs` to delete the files listed in that key, and then update its value to be empty again once the files are deleted:

```
JavaScript
// src/libs/actions/Share.js

// constructs share-specific data for use in `addAttachment`
const cleanUpActions = (file) => {
    if (!file || !file.source.includes(appGroupPath)) return [];
    return [
        {
            onyxMethod: Onyx.METHOD.MERGE,
            key: ONYXKEYS.TEMP_FILES_TO_DELETE,
            value: [file],
        },
```

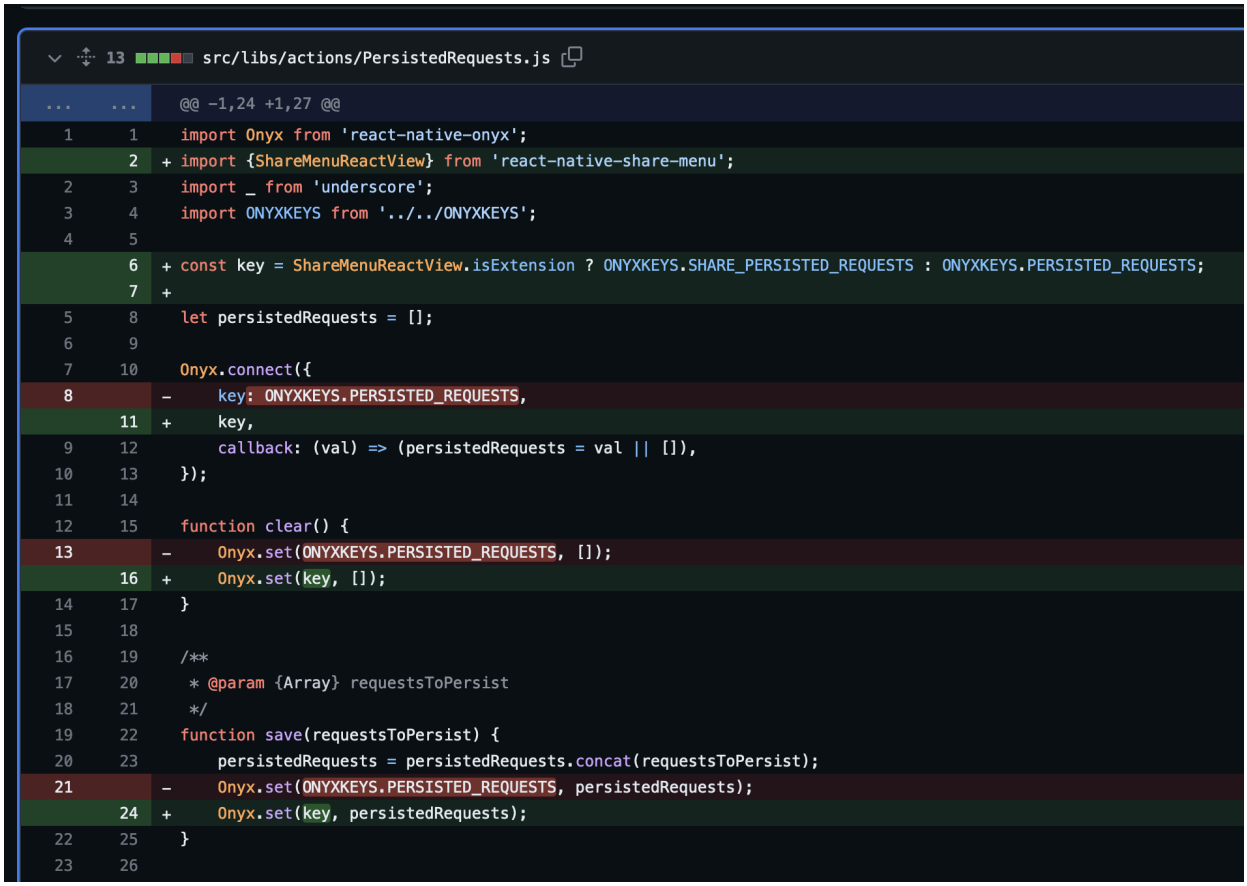
```
    ];
};

let isCleaningUpTempFiles = false;
Onyx.connect({
  key: ONYXKEYS.TEMP_FILES_TO_DELETE,
  callback: (val) => {
   if (!val || isCleaningUpTempFiles) return;
   isCleaningUpTempFiles = true;
   val.forEach((file) => {
    exists(file.source).then((fileExists) => {
     if (!fileExists) return;
     unlink(file.source);
    });
   });
  Onyx.set(ONYXKEYS.TEMP_FILES_TO_DELETE, []);
  isCleaningUpTempFiles = false;
  },
});
```

Making PersistedRequests reload when the main app is foregrounded

Currently, the app is not designed to (reliably) reload persisted requests from the database when the main app resumes. It has previously relied on all relevant requests being available in memory if the app was running.

To make the iOS app read any requests the share extension may have persisted, we'll create a new Onyx key ,`shareExtensionNetworkRequestQueue`, where the share extension requests will be stored. This will also be the key used by `PersistedRequests` if we're running in the share extension:



Then, on foregrounding, when the `flush` method in `SequentialQueue` is called in the main app, it will then ensure that the share extension's queue is added to the in-memory value used by `PersistedRequests`:

JavaScript

```
// src/libs/Network/SequentialQueue.js
function flush() {
  if (Share.flushAppExtensionQueue(flush)) {
   return;
  }
  ...
}

// src/libs/actions/Share.js

let isAppExtensionQueueFlushed = false;
const flushAppExtensionQueue = (callback = () => {}) => {
    if (isAppExtensionQueueFlushed) return false;
    const connectionID = Onyx.connect({
        key: ONYXKEYS.SHARE_PERSISTED_REQUESTS,
        callback: (val) => {
            Onyx.disconnect(connectionID);
```

```
            // save to in-memory value
            PersistedRequests.save(val || []);
            isAppExtensionQueueFlushed = true;
            Onyx.set(ONYXKEYS.SHARE_PERSISTED_REQUESTS, []);
            callback();
        },
    });
    return true;
};

AppState.addEventListener('change', (appState) => {
    if (ShareMenuReactView.isExtension) return;
    if (appState === CONST.APP_STATE.ACTIVE) return;
    // queue is reset to "not flushed" in memory in the main app if the app is
anything but foregrounded
    isAppExtensionQueueFlushed = false;
});
```

Ensuring Onyx's in-memory cache is updated

When the user accesses a report, Onyx stores a copy of that report's data in memory, cached. That report is typically only updated by the instance of Onyx in the main app, so Onyx has always known when that cached value needs updating. However, this changes with the introduction of the share extension, which is running its own instance of Onyx in its own process. Any running instance of the main app would not be aware that it needs to update any in-memory copy it may have.
So we need a way for the main app's Onyx instance to re-read the values from the database.

We'll add a method to Onyx that allows us to force a given key or keys to be read from the database and updated in memory, which we'll then call when the app is foregrounded. Since there are only a small number of keys that require this, this is unlikely to have a significant impact on performance. If the number of keys required to load on app foregrounding grows, this approach is also scalable: that method can instead be called internally by Onyx. To reduce the amount of data that needs to be read from disk, processes that make changes can flag, in a new database table, what keys they changed and when, to let other processes know they should invalidate their own in-memory cache.

UI added/changed

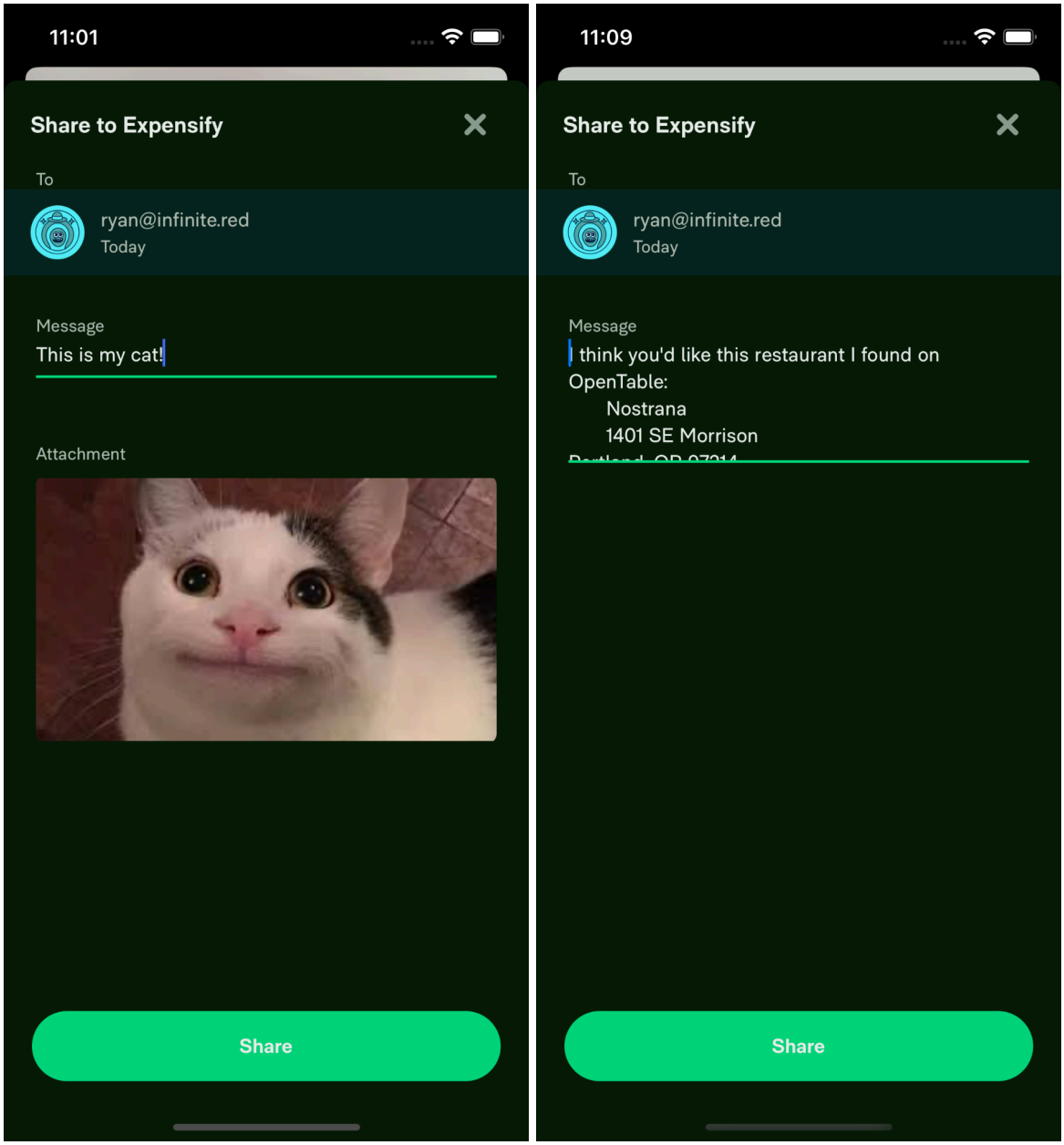
ShareMessagePage

This new page will use the following components:

- OptionsRowLHNData to display who the share is being sent to
- TextInput with autoGrowHeight to accommodate multi-line messages. Many shared pieces of text from other apps may include multi-line messages, e.g., a referral message and link.
- AttachmentView, if the shared item is a file

- Button, inside a FixedFooter, to send the share

ShareMessagePage UI:



NewSharePage (NewGroupPage/NewChatPage)

We'll add a screen called `NewSharePage`, that, like `NewGroupPage`, is a thin wrapper around `NewChatPage` with the `isGroupChat` parameter and a new `isShare` parameter.

We'll add two conditions to `NewChatPage` based on whether there is share data present in state: the "Create group" button behavior to navigate to ShareMessagePage and update the title bar to render "Share to Expensify" and a close button.

JavaScript

```
- <HeaderWithBackButton title={props.isGroupChat ?
props.translate('sidebarScreen.newGroup') :
props.translate('sidebarScreen.newChat')} />
+ <HeaderWithBackButton
+   title={share ? props.translate('newChatPage.shareToExpensify') :
props.translate(props.isGroupChat ? 'sidebarScreen.newGroup' :
'sidebarScreen.newChat')}
+   shouldShowBackButton={!share}
+   shouldShowCloseButton={!!share}
+   onCloseButtonPress={Share.dismiss}
+  />
```

SigninPage

The sign-in page will have a condition to render a prompt to go to the main app and sign in, if it is shown inside of the iOS share extension.

iOS Share Extension configuration

Setting up the Share Extension

Largely following the guide from the basic iOS setup, and then for the custom React Native view for the share extension.

In brief, we:

1. Create a new share app extension, using XCode, and configure, similar to the main app:
    a. Code signing
    b. The Podfile to manage the share extensions code linking and settings that are also managed for the main app
2. We add react-native-share-menu's `ShareViewController` to the share extension, and then its `ReactShareViewController` as well, to host the React Native UI
3. We create an App Group that contains the main app and the share extension so that they can share data, such as the Onyx database.
4. Add any natively linked assets (i.e., fonts) to the share extension.

Share Extension React Native view

The react-native-share-menu library extends `SLComposeServiceViewController` and creates a new `RCTRootView` for `ShareMenuModuleComponent`. To use it, we just need to re-register the app under this name, and conditionally navigate based on whether there is share data:

JavaScript

```
// index.js
AppRegistry.registerComponent('ShareMenuModuleComponent', () => App);
```

```
AppRegistry.registerComponent(Config.APP_NAME, () => App);

// src/Expensify.js

function Expensify() {
  ...
  useEffect(() => {
    Navigation.isNavigationReady().then(() => {
      ShareMenuReactView.data().then(navigateToShare);
    });
    return {remove: () => {}};
  }, []);
...
}
```

Make Onyx accessible from the Share Extension

Since iOS application processes are sandboxed, and application share extensions run in a separate process, we will need to update react-native-quick-sqlite to allow us to store Onyx data in a location accessible by both the application and the share extension (in app group storage). We've already merged a PR to add this option to the library.

We'll add an Onyx migration to move the location of the database on iOS to the app group directory, using react-native-filesystem (which is already used elsewhere in the app). This should have no impact on user experience.

Make the share extension maintainable

The share extension has much of the same configuration as the main app, but it is specified separately. The share extension also shares components with the main app. Ensuring updates to the main app's configuration and code are also made, as appropriate, to the share extension's configuration will be important for long-term maintenance.

To ensure this, we:

1. Use `inherit! :complete` in the share target in the Podfile to ensure the configuration is applied to both the main app and the share extension, unlike the react-native-share-menu instructions, which instruct developers to configure it manually.
2. Using the same entry point for the iOS app and share extension. Sharing app initialization and context as much as possible between the main app and share extension helps ensure that it is obvious the share extension exists and must be considered when changes are made to app architecture and context.

react-native-share-menu updates

Fixes

Updates for latest React Native changes

1. `jsBundleUrl` arguments must be changed (issue link).
2. Update Android compileSdkVersion (issue link).
3. `ShareMenuModule.java` needs to implement NativeEventEmitter methods to avoid warnings (issue link).

Crash

1. There have been intermittent crashes on iOS due to issues with the view delegate state handling. We have a basic fix for this, but there are related issues we should look into:
    a. Sharing multiple times with `continueInApp` (which is NOT an API Expensify will be using) crashes (issue)
    b. `viewDidDisappear` is not consistently called, so extension cleanup does not happen consistently and can lead to memory issues (issue)

Mime-types on iOS reported incorrectly

We've found that on iOS, non-image files are returning mimeTypes of "text/plain" to JavaScript even when they aren't text files.

We might fix this, but if it's too complex, might just remove mimeType support and check for the presence of a `file://` URL to distinguish files from text shares.

New features

Determining if JS code is running inside the share extension

The iOS share extension is an exceptional context, and requires different behavior of otherwise reusable components (e.g., SignInPage displaying a page to redirect users back to the main app to sign in). To support this, we need to add a method to the native module that can tell the JavaScript layer if it is running inside the share extension.

Managing shared files past the share extension's lifetime

This is to support offline sharing. The files that the share extension uses are temporary copies that typically go away when the share extension is dismissed. In order to support the offline use case and allow them to be uploaded later, after the user has reconnected, we need to:

1. Have a way to copy the files to a non-temporary location
2. A method to clear out these copies after they've been sent to the Expensify API.

Allowing custom native setup

In order to support the ErrorBoundary component in the app, we have to initialize Firebase in the native code setting up the share extension.

We would like to modify react-native-share-menu's setup instructions to have users create their own native file based on a template, instead of linking to a file within react-native-share-menu.

This would make future upgrades for library consumers a bit more involved, but shouldn't be too complicated, as the files involved are minimal.

Future work

This section outlines work we won't be tackling in the scope of this project, but will be important to plan for in the future for this library.

Privacy manifest
Apple is going to start requiring that libraries and apps that use specific features register their "reason" for doing so in a new privacy manifest. This includes UserDefaults, which react-native-share-menu uses to store shared data to pass to the main app. Its usage of UserDefaults is fully compliant with the allowed "reason" ("Declare this reason to access user defaults to read and write information that is only accessible to the app itself"), so we don't need to change anything.

This will only be possible to add starting with XCode 15, so we can't do it right away, and will need to track this required change in the future.

New Architecture support
React-native-share-menu, being a bit out of date, does not have support for the New Architecture. It should not be difficult to add, but we should make the API changes we need above first or in tandem with updating for the New Architecture.

This will require converting the module to a Turbo Module. There are no native components that are used **from** React Native, only native components that host React Native, so we don't expect Fabric to be a consideration, although there is a possibility that using Fabric from within an app extension may have unexpected issues.

Manual tests

User is online and already signed into the Expensify app

Share a single image to New Dot

1. In another app, find an image to share. Choose the share option, then choose New Expensify from the list of apps.
2. The "New Group" UI will be shown. On Android, this will be shown inside the app. On iOS, this will be shown inside a near-full-height modal screen, displayed over the app where the share was initiated.
3. Choose a person to share with by searching for them in the text box, or selecting them in the list shown beneath. Click "Create Group" to continue.
4. The Share UI will be shown. You should see:

    a. The person(s) the share will be sent to, with a preview of the last message in the conversation if applicable
    b. A text field where you can add an optional message. In this case, it should be empty.
    c. A preview of the image.
    d. A “Share” button.
5. Add a message to the message field.
6. Click “Share”.
    a. On Android, you should be redirected to the conversation you shared to, and should see the shared image and message in that conversation.
    b. On iOS, the Share UI should close, and you will see the app where you started the share from. If you open the app and go to the conversation you shared the image to, you should see the message and image attachment you sent.

Share a single, non-image file to New Dot

What’s different from the image file test case? There will not be an image preview of the file; instead, there will be an attachment component that displays the file name.

1. In another app, find a file that isn’t an image (e.g., an mp3) to share. Choose the share option, then choose New Expensify from the list of apps.
2. The “New Group” UI will be shown. On Android, this will be shown inside the app. On iOS, this will be shown inside a near-full-height modal screen, displayed over the app where the share was initiated.
3. Choose a person to share with by searching for them in the text box, or selecting them in the list shown beneath. Click “Create Group” to continue.
4. The Share UI will be shown. You should see:
    a. The person(s) the share will be sent to, with a preview of the last message in the conversation if applicable
    b. A text field where you can add an optional message. In this case, it should be empty.
    c. A file attachment component showing the file’s name.
    d. A “Share” button.
5. Add a message to the message field.
6. Click “Share”.
    a. On Android, you should be redirected to the conversation you shared to, and should see the shared file and message in that conversation.
    b. On iOS, the Share UI should close, and you will see the app where you started the share from. If you open the app and go to the conversation you shared the file to, you should see the message and file attachment you sent.

Share text to New Dot

What's different from the image or file test cases? There will be no preview or attachment component, and the text will be automatically inserted in the message field.

1. In another app, highlight and share some text, then choose New Expensify from the list of apps to share to.
2. The "New Group" UI will be shown. On Android, this will be shown inside the app. On iOS, this will be shown inside a near-full-height modal screen, displayed over the app where the share was initiated.
3. Choose a person to share with by searching for them in the text box, or selecting them in the list shown beneath. Click "Create Group" to continue.
4. The Share UI will be shown. You should see:
    a. The person(s) the share will be sent to, with a preview of the last message in the conversation if applicable
    b. A text field, pre-filled with the text that was shared. You should be able to edit this text.
    c. A "Share" button.
5. Update the message in the message field.
6. Click "Share".
    a. On Android, you should be redirected to the conversation you shared to, and should see the message you wrote in that conversation.
    b. On iOS, the Share UI should close, and you will see the app where you started the share from. If you open the app and go to the conversation you shared to, you should see the message you sent.

Share a link to New Dot

This test case is the same as sharing text. The link will not be clickable, highlighted, or previewed, just treated as text.

Share a file to multiple other users

In all cases listed above, you should be able to select multiple users before clicking "Create group" and the share will go to that group.

Share a file to a conversation/group that did not exist before

You should be able to share to both conversations and groups that did exist before, and conversations and groups that didn't. If you choose a conversation or group that did not exist before, a new one will be created after selecting "Create group".

Try to share multiple files to New Dot

1. Go to an app that allows selecting and sharing of multiple files (e.g., a photo gallery app).
2. Select and share multiple files.

3. You should not see New Expensify listed as an option among the applications in the system share dialog.

Cancel flow before pressing send

Both platforms

The user can abandon the share process. If they have pressed “Create group” already, and that group did not exist prior to the share attempt, that group will remain created.

iOS, using close button

1. Share a file to New Expensify.
2. Create a new group to share to, press “Create group”.
3. On the next screen, press the “X” button in the top right corner to close out the share modal. The share extension should close, and you should see the app that was shared from.
4. Go to New Expensify. The newly created group should be in the conversation list, but with nothing shared to it.

Android, using close button

1. Share a file to New Expensify.
2. Create a new group to share to, press “Create group”.
3. On the next screen, press the “X” button in the top right corner to close out the share modal. The Share UI should close, but the app should remain open.

Android, using back button

1. Share a file to New Expensify.
2. Create a new group to share to, press “Create group”.
3. Press the back button once to go back to the group creation screen, and then back again. The result depends on whether the app was previously opened:
    a. If the New Expensify app was already open and running in the background, the back button will navigate to the screen that was visible prior to the share attempt
    b. If the New Expensify app was not in the background prior to the share attempt, the app will exit.

User is offline and already signed into the Expensify app

User experience should be the same for offline users, except they will see indicators that they are offline at the bottom of each screen.

When looking at the conversation that they shared to, users who have remained offline will see the same UI as when sending a message or attachment while offline otherwise.

1. Turn on airplane mode on phone

2. Choose a file to share and share to New Expensify
3. Choose a group to share to
4. Add a message and press send
5. While offline, navigate to the group you shared to
6. The offline message pending state should be shown
7. Turn off airplane mode on phone
8. Message should be sent, and attachment and message should become visible in the conversation

User is not signed into the Expensify app

Android

1. Log out in the New Expensify app if already signed in.
2. Go to another app and try to share a file to New Expensify
3. The login form will be displayed. Sign in.
4. The sign in process continues as normal. The user is not redirected to the share flow.
5. Go back to the other app and try one of the share flows listed above. They should work as described.

iOS

1. Log out in the New Expensify app if already signed in.
2. Go to another app and try to share a file to New Expensify
3. A message explaining you will need to go to the main app and sign in will be displayed. Click the button to continue to the main app.
4. The share modal will be closed and the New Expensify app will be opened. The login form will be displayed.
5. The sign in process continues as normal. The user is not redirected to the share flow.
6. Go back to the other app and try one of the share flows listed above. They should work as described.

Automated tests

We do not plan on adding automated tests. We looked to see if the share extension could easily be tested with an automated tool, since that has the highest risk of breaking, but support for this in existing tools is poor.

Alternate solutions (detailed)

Alternative libraries

There is another library react-native-share-extension to use RN in a share extension but it's even less maintained than react-native-share-menu.

We could also write a new library but despite the maintenance status of react-native-share-menu it works well with the latest RN with a few small patches.

iOS-specific solutions

Optimizing native code size in the share extension

If needed, dynamic linking (a.k.a. embedded frameworks) can be used to share native code between the main app and share extension, instead of copying it, but in tests we found the potential benefits of this to be less than we expected.

Dynamic linking impact on app size

Below are the sizes of New Expensify with the share extension when built with static linking, versus when we configure most of the dependencies to use dynamic linking. (Not all dependencies were compatible with static linking.)

	Static linking	Dynamic linking
App Executable	84 MB	9 MB
Share Executable	91 MB	15 KB
Frameworks	139 MB	237 MB
JS Bundle	13 MB	13 MB
Other	1 MB	1 MB
Total .app size	328 MB	275 MB
Total .ipa size (compressed)	62 MB	59 MB

Drawbacks of dynamic linking

Dynamic linking is not usually tested and supported by React Native libraries. Even core React Native tends to have issues with `use_frameworks!` that need fixing often. Switching to frameworks adds a significant, long-term maintenance burden to an app that shows up during React Native upgrades for this reason..

Additionally, linking some libraries statically and some dynamically is non-standard for a React Native app and might be challenging for most React Native developers to maintain. As a final note, while it might be possible to update dependencies to work with dynamic linking, it would likely be a major investment. The following are the list of libraries that caused issues with dynamic linking:

```
C/C++
@oguzhnatly/react-native-image-manipulator
@onfido/react-native-sdk
@react-native-firebase/analytics
@react-native-firebase/app
@react-native-firebase/crashlytics
@react-native-firebase/perf
onfido-sdk-ui
react-native-dev-menu
react-native-quick-sqlite
react-native-reanimated
```

Manually linking pods

Manually linking a list of pods to the share extension is another possible strategy to reduce app size. However, if we are sharing component implementations across the main app and the share extension, manually linking pods increases risks of a share extension crash if a shared component is changed to require new native code and the share extension's configuration isn't properly updated to include it.

In this case, automated testing would be crucial to ensure PRs did not break the share extension.

Creating a separate, minimal bundle for the iOS share extension

Since the share extension needs relatively few screens, we explored the possibilities of extracting reusable components or reimplementing components in order to keep the share extension small, fast, and simple. However, we found that both of these options were much more complicated than using the same entry point as the main app, and performance and size were still good with this approach.

Reusable components

The "reusable components" approach means finding which existing components we need to use in the share extension, and constructing a new entry point that includes only those components. For example, this approach would have its own navigator that did not include screens from the main app that the share extension does not use.

However, we found that the screens and components needed were difficult to extract, and that entry point configuration would involve a lot of duplication: for example, a similar, but not identical, Navigator setup; a similar, but not identical, list of context providers; and so on. Components are particularly sensitive to having the correct context providers available, and not having them available is difficult to identify until runtime testing. This duplication increases the risk that future changes might get applied to components and the main app configuration, but miss the share extension until the QA stage. We believe that a unified entry point means earlier, easier discovery of code paths that impact Native Share.

Reimplementation

The “reimplementation” approach means re-implementing the required components so that they are less coupled to the entire context of the main application. This would take a lot of time, and potentially require revisiting a lot of existing architectural decisions, so we decided this approach was too inefficient.

Potential impact on size

As a quick test, we measured the size of a new React Native app that only included `react-native` as a dependency, nothing else. That only saved about ~30 MB compared to the Expensify executable, and combined with the JS bundle size, that’s ~40-50 MB. From what we’ve seen in our other tests, this might save one megabyte in the .ipa file, so it’s likely not worth the effort involved.

Separate app and share extension databases

We could negate the need to update the DB location by treating the application and share extension data as completely separate. This would require fully syncing and authenticating within the share extension and would duplicate data on the device.

Supporting offline share

A more complex refactor of Onyx and the storage layer

While we’re fairly confident that we can make changes to clear the appropriate in-memory caches of the storage layer in a fairly straightforward way, it’s possible that there will be edge cases we didn’t anticipate. If that is the case, we may need to create a new proposal for how to refactor and change the code without disrupting any current functionality.

Send user to main app to share

In the worst case scenario, if the required changes to the Onyx and storage layers prove intractable, we can avoid using a share extension to host the entire share flow, and instead redirect users to the main app. React-native-share-menu has existing support for this flow, and it would not be hard to implement, as it should then be able to re-use all of the existing work for Android.

Embedding files into the stored request instead of copying files

We found that base64-encoding files and embedding their data directly into the `addAttachment` command worked to support offline requests, rather than having to copy files to a new temporary location and later delete them when we are done uploading them.

However, this approach will both limit potential file size and have potential negative impacts on memory usage for large files. Since this introduces surprising limitations and potential performance issues, we opted not to use this solution.

Reading persisted requests in the main app

Below are a few of the options we considered for how to handle reading persisted requests in the main app. Original discussion of options (link).

Reload everything from disk when foregrounding app

We could try to reload everything from the database when we foreground the main app again, but this is likely to cause performance issues: either making the UI slow to load, or memory consumption.

Refactor to avoid in-memory persisted request object

It could be possible to refactor the storage layer code in the app to avoid this problem, either by:

1. Avoiding using an in-memory object instead of the database, to instead read from disk, at least when the app is foregrounded, if not always.
2. Changing the timing of when persisted requests are loaded. PersistedRequests does read from the database, but potentially too late for this particular scenario.

Ultimately though, changing this code layer substantially creates high risk for what could otherwise be a tiny change.

System requirements

We'll need to work with the internal Expensify team on how we want to handle react-native-share-menu updates and the new release of the library.

Plan of action / Rollout plan

1. We'll make changes to react-native-share-menu where necessary, and make a new release of the library,
2. We'll pull in the updated RNSM release to our feature branch.
3. In parallel, we'll implement the share feature for iOS and Android in one PR.

Communication Plan

A thorough Communication Plan ensures our internal (help docs, Zingtree) and external resources (Community posts and marketing items) are completed on time when the product changes. The leader of this design doc is responsible for the below tasks.

1. **Create a Launch/Guides task GH** using this template, a #launch team member will be automatically assigned after you create the GH. The Launch team member will be responsible for organizing any sales campaign related to the project.
2. **Create an External Resource GH** using this template, a #Resource-Management team member will be automatically assigned after you create the GH and responsible for organizing the updates.

3. **Create an Internal Resource GH** using this template, a #Resource-Management team member will be automatically assigned after you create the GH and responsible for organizing the updates.
    a. Reach out in #resource-management if you have any questions about making these GHs.
4. **Marketing Comms Chore GH** using this template a #marketing team member will be automatically assigned after you create the GH and responsible for organizing the updates
5. **Wrap up CAP Sheet** entries. As soon as your project(s) go live, please note your Launch Date(s) on the CAP Sheet.

Detailed implementation reviewed by

Authors:
Please make sure to **get the correct number of reviews from each G&R tier** before beginning to implement the detailed portion. Please follow this SO to guarantee reviews by applying the DesignDocReview label to your tracking issue. Thanks!

Reviewers:
After you have thoroughly reviewed this doc, add your name and date in the section that corresponds to your Growth and Recognition tier.

Expensifiers + Graduates (Need at least 2)
2023-09-01 *- Jules - looking good, very detailed explanation!*
2023-09-20 - Ariel

Project Managers (Need at least 2)
20223-10-04 - Steph E
2023-10-04 - Greeny

Product Managers or Generalists (Need at least 4 engineers and 4 non-engineers)
2023-09-04 - puneet
2023-09-07 - Yuwen
2023-10-03 Anu
2023-10-24 - Rocio

Project wrap up

Once the project is finished, update the CAP Sheet project status and launch date. Then, complete this section and email its contents to strategy@. What went well? What could we have done better? What did we learn?