Deliverable You should submit a zip file containing the following items.
1. Your whole project (ALL parts needed to run your system)
2. README file to explain how to run your system if there are any extra steps
3. Your implementation document, which should include the information specified below. Your implementation document must have the following sections.
1. Project Title and Authors a. Your team name as appeared on D2L b. A list of all team members (names and USC ID numbers) 2. Preface 3. Introduction 4. Architectural Change a. What architectural changes did you make during implementation? b. What's the rationale behind the decision change? 5. Detailed Design Change a. What detailed design changes did you make during implementation? b. What's the rationale behind the decision change? 6.

Requirement Change Note: You do not need to implement these changes for this assignment. You may implement it in a later assignment. Your analysis of the impact on your design will be taken into account again when grading the future assignments. Please answer the following questions for each scenario regarding your product below. 1. Does this change your design? 2. If not, why not? 3. If so, what should be changed and how?

1. PROJECT TITLE AND AUTHORS

a. Team 25 (Focus!)
b. Team Members

1. Rowen Wu 3552815427
2. Stephanie Lee 7779992403
3. Avand Lakmazaheri 9977188081
4. Andy Ly 6330086001
5. Yuan Xu 5533333231

2. PREFACE

The purpose of the implementation document is to outline the implementation of the mobile application, Focus!. More specifically, the document provides detail on how the actual implementation differs from the proposed design, which was outlined in the design document. The developers will use this document to keep track of what changes were made in terms of the application's architecture and detailed design. Additionally, the developers will discuss how the application's architecture and design may need to be changed further in potential scenarios where the customer has added new requirements.

The original descriptions and diagrams from the design document are included, and the changes we have made in our implementation are noted in red.

3. INTRODUCTION

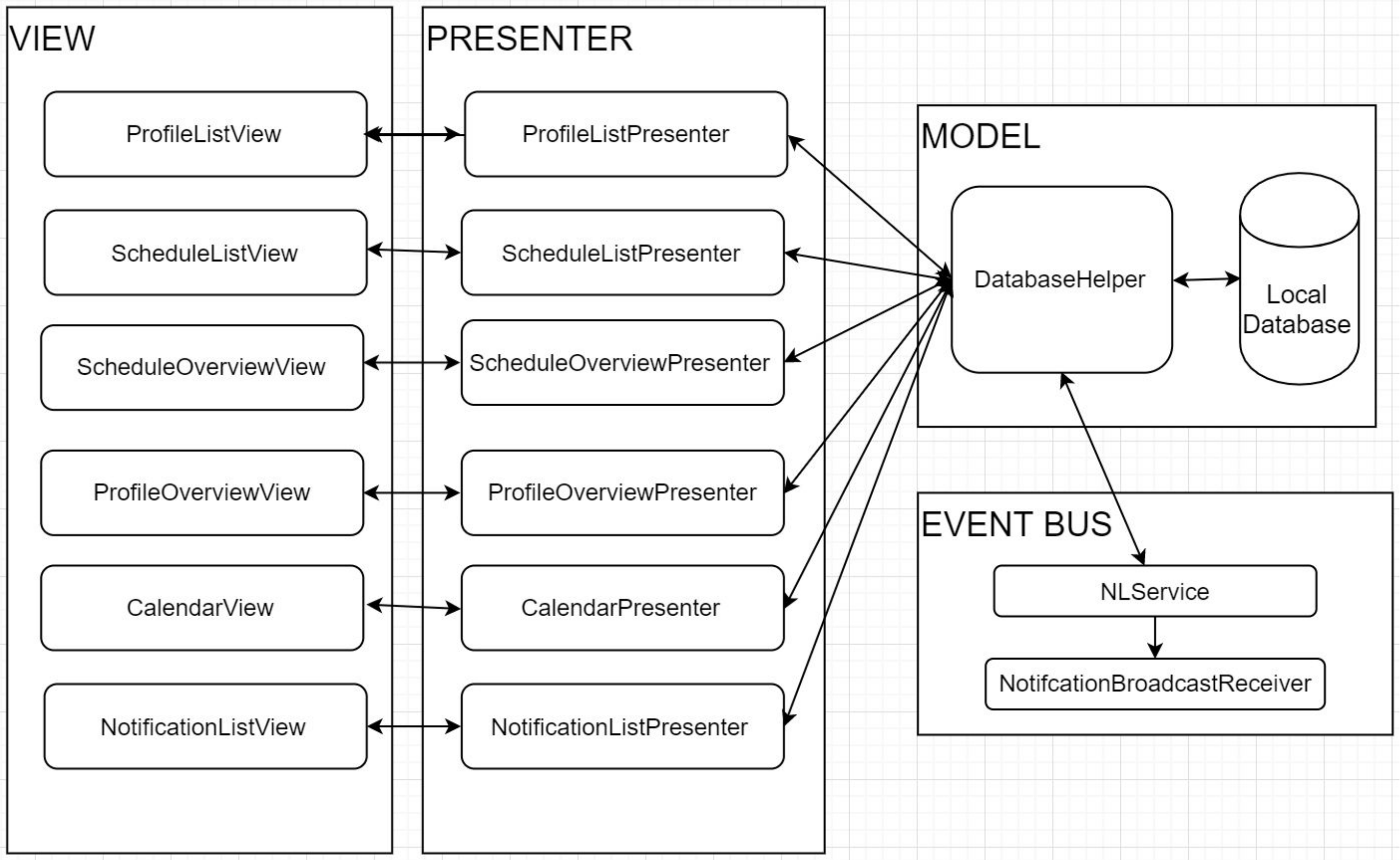
Focus! is designed to stop notifications on a mobile device from distracting a user. A user can create profiles that specify which applications to block. The application will allow the user to create multiple schedules, during which specific applications will be blocked. Schedules can block multiple profiles, so that users can easily modify what apps are blocked at what times of the week. When an application is blocked, no notifications will be sent to the user's phone. Once the time on a schedule has expired, the user will be provided information about the notifications that were received while Focus! was in use. Focus! is extremely customizable, allowing the user to specify which apps they want to block at what times. Schedules can be set to automatically activate at certain times throughout the week.

4. ARCHITECTURAL CHANGE

a. What architectural changes did you make during implementation?
b. What's the rationale behind the decision change?

High-Level Architecture

This diagram gives a general overview of the architecture of the Focus! App. The app will beleveraging the Model View Presenter (MVP) design pattern. MVP was chosen because it allows for better separation of concerns and greater app testabilityfor Android applications than Model View Controller. The three main layers ofthe architecture are: the Model, which holds and manages data; the View, which is a passive UI that displays data and routes actions to the Presenter; and the Presenter, which is an intermediary between the Model and View. The event bus includes underlying services and broadcast receivers. Services allow the app to perform actions even when it's not running, and broadcast receivers allow the app to receive system-wide broadcasts. These are not considered to be a part of the other MVP components. The inner elements are the main components of the app architecture that are involved in actual data presentation or manipulation. These inner elements may represent classes, but not all classes of the app are represented in this diagram.



Instead of using the MVP Architectural design we used a MVC Architectural design. Both the Views and Presenters were able to make calls to the database through the Room Persistence Framework. We changed it to MVC because there were cases where the layer of abstraction between the view and database was not necessary. We utilized the Adaptor classes as the presenters to populate the different fields of the view.

5. DETAILED DESIGN CHANGE

a. What detailed design changes did you make during implementation?
b. What's the rationale behind the decision change?

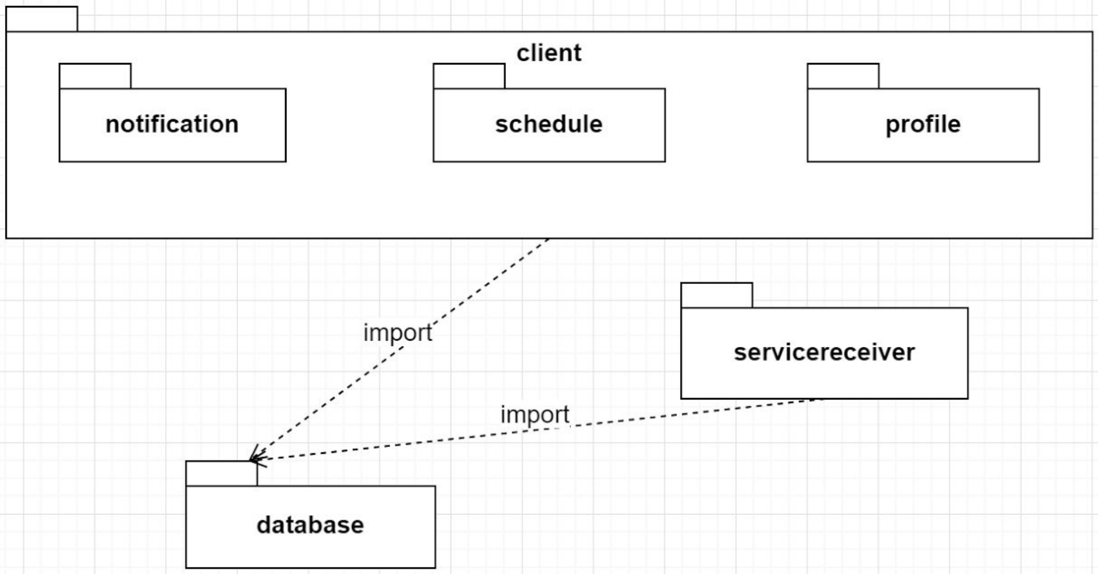
The following pages of the document show diagrams of the detailed design of the Focus! App. Changes made during implementation are listed in red. Listed below are the external libraries and frameworks we will be leveraging.

Frameworks/Libraries:

- Google Maven repository
  This repository is published by Google to help Android developers create architecture components that follow Google's architectural recommendations. We will be importing these components to reuse in our app architecture.
  https://developer.android.com/topic/libraries/architecture/adding-components.html
- Android Week View
  This is a library that provides and implementation of a weekly or daily calendar view. We will be using this in our presenter components to display schedules.
  https://libraries.io/github/ribot/Android-Week-View
- Android Job
  This library allows users to avoid rewriting boilerplate code by combining functionalities of Android's AlarmManager, JobScheduler, and GCMNetworkManager. We'll be using this in our event bus components to schedule profile activation.
  https://github.com/evernote/android-job

Package Diagram

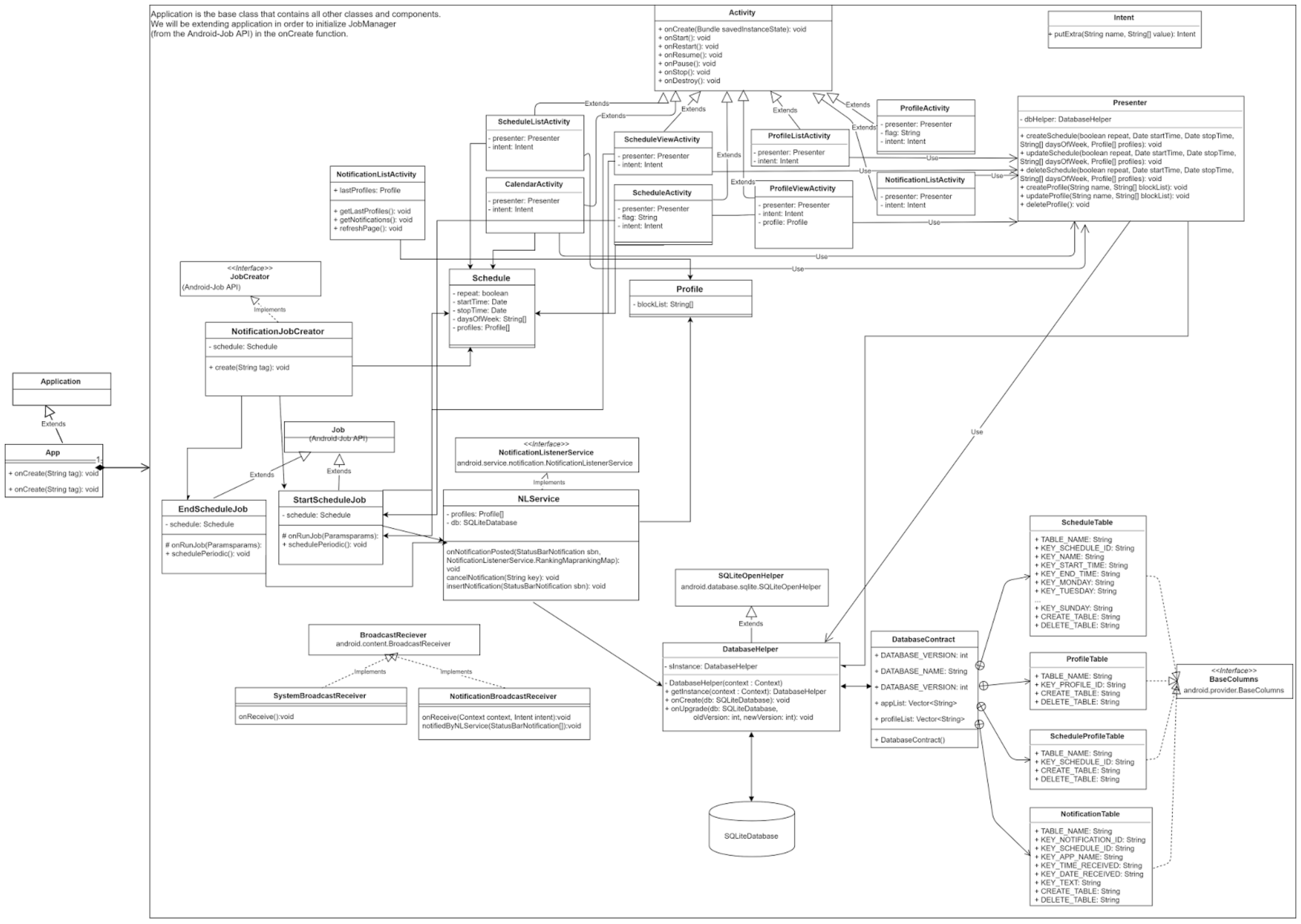
This diagram shows structure of the designed system at the level of packages. The diagram shows the various packages of the app and the import relationships between packages. Packageable elements are not shown, because these elements are the classes shown in the class diagram. The client package is composed of client-side packages, which correspond to the View and Presenter aspects of the MVP model outlined in the high-level architecture. The client package consists of three packages: notification, schedule, and profile, which contain the classes related to generating/.interacting with the UI for displaying notifications, schedules, and profiles. The package, servicereceiver, will contain the background services and broadcastreceivers that will be listening for and blocking notifications and sending push notifications during times when profiles are active. database is access through a database helper, as well as a database contract, which is a schema that defines how the database is set up.



The package hierarchy of our implementation is fairly consistent with our diagram, but we also have packages for entities and daos, which include the classes that communicate with the database through room persistence.

Class Diagram

This diagram shows the majority of classes of the app and their main implemented functions. A closer look at specific classes in the various MVP components of the app architecture are shown in pages 7-9. Some getter and setter functions and other boilerplate functions are not shown. In addition, some of the nested UI classes are not shown, such as fragments.
An unlabeled arrow represents a dependency of one class upon another.

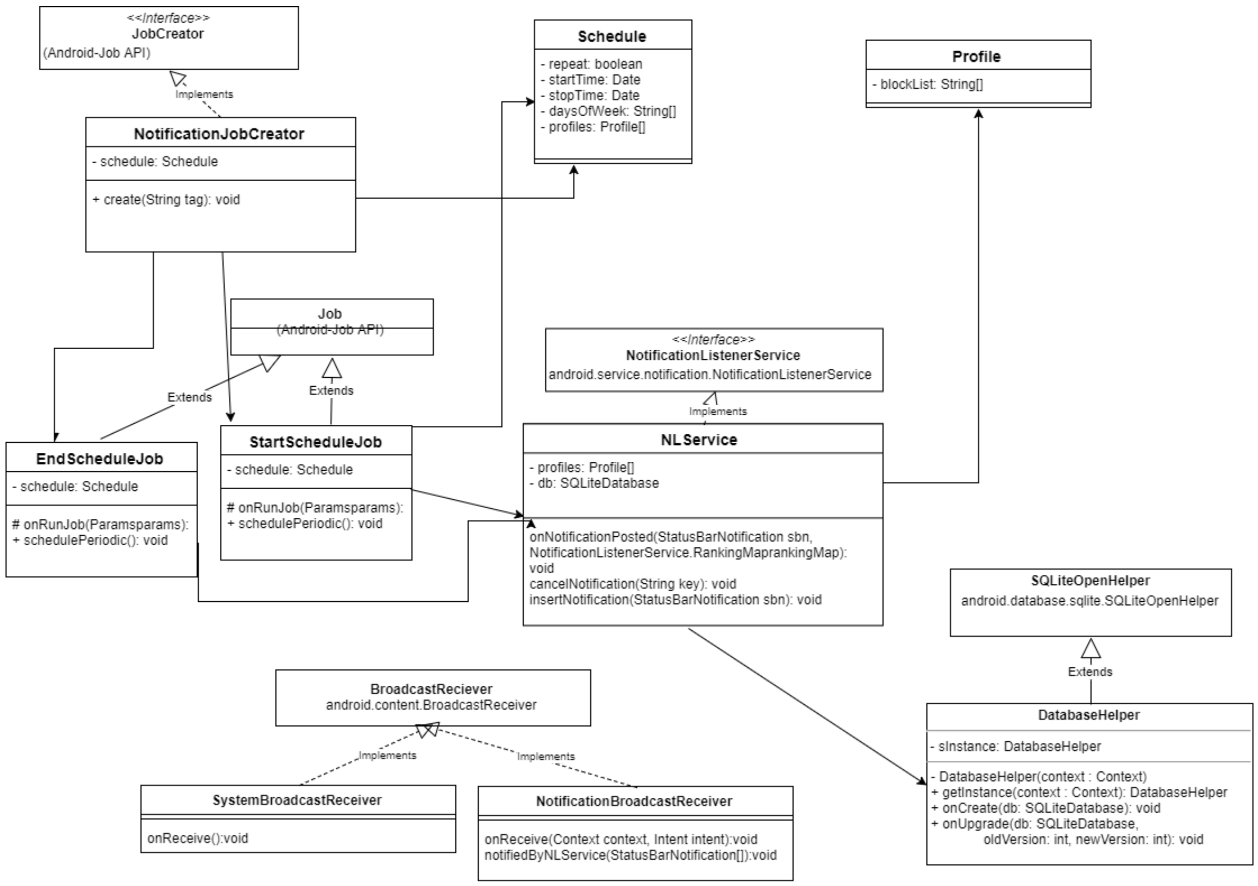


Application is the base class that contains all other classes and components.
We will be extending application in order to initialize JobManager
(from the Android-Job API) in the onCreate function.
Activity
+ onCreate(Bundle savedInstanceState): void
+ onStart(): void
+ onRestart(): void
+ onResume(): void
+ onPause(): void
+ onStop(): void
+ onDestroy(): void
Intent
+ putExtra(String name, String[] value): Intent
Extends
Use
Implements
ScheduleListActivity
- presenter: Presenter
- intent: Intent
ScheduleViewActivity
- presenter: Presenter
- intent: Intent
ProfileListActivity
- presenter: Presenter
- intent: Intent
ProfileActivity
- presenter: Presenter
- flag: String
- intent: Intent
NotificationListActivity
+ lastProfiles: Profile
+ getLastProfiles(): void
+ getNotifications(): void
+ refreshPage(): void
CalendarActivity
- presenter: Presenter
- intent: Intent
ScheduleActivity
- presenter: Presenter
- flag: String
- intent: Intent
ProfileViewActivity
- presenter: Presenter
- intent: Intent
- profile: Profile
NotificationListActivity
- presenter: Presenter
- intent: Intent
Presenter
- dbHelper: DatabaseHelper
+ createSchedule(boolean repeat, Date startTime, Date stopTime, String[] daysOfWeek, Profile[] profiles): void
+ updateSchedule(boolean repeat, Date startTime, Date stopTime, String[] daysOfWeek, Profile[] profiles): void
+ deleteSchedule(boolean repeat, Date startTime, Date stopTime, String[] daysOfWeek, Profile[] profiles): void
+ createProfile(String name, String[] blockList): void
+ updateProfile(String name, String[] blockList): void
+ deleteProfile(): void
<<Interface>>
JobCreator
(Android-Job API)
Schedule
- repeat: boolean
- startTime: Date
- stopTime: Date
- daysOfWeek: String[]
- profiles: Profile[]
Profile
- blockList: String[]
NotificationJobCreator
- schedule: Schedule
+ create(String tag): void
Application
App
+ onCreate(String tag): void
+ onCreate(String tag): void
1
Job
(Android-Job API)
<<Interface>>
NotificationListenerService
android.service.notification.NotificationListenerService
EndScheduleJob
- schedule: Schedule
# onRunJob(Paramsparams):
+ schedulePeriodic(): void
StartScheduleJob
- schedule: Schedule
# onRunJob(Paramsparams):
+ schedulePeriodic(): void
NLService
- profiles: Profile[]
- db: SQLiteDatabase
onNotificationPosted(StatusBarNotification sbn, NotificationListenerService.RankingMaprankingMap): void
cancelNotification(String key): void
insertNotification(StatusBarNotification sbn): void
BroadcastReciever
android.content.BroadcastReceiver
SystemBroadcastReceiver
onReceive():void
NotificationBroadcastReceiver
onReceive(Context context, Intent intent):void
notifiedByNLService(StatusBarNotification[]):void
SQLiteOpenHelper
android.database.sqlite.SQLiteOpenHelper
DatabaseHelper
- sInstance: DatabaseHelper
- DatabaseHelper(context : Context)
+ getInstance(context : Context): DatabaseHelper
+ onCreate(db: SQLiteDatabase): void
+ onUpgrade(db: SQLiteDatabase, oldVersion: int, newVersion: int): void
SQLiteDatabase
DatabaseContract
+ DATABASE_VERSION: int
+ DATABASE_NAME: String
+ DATABASE_VERSION: int
+ appList: Vector<String>
+ profileList: Vector<String>
+ DatabaseContract()
ScheduleTable
+ TABLE_NAME: String
+ KEY_SCHEDULE_ID: String
+ KEY_NAME: String
+ KEY_START_TIME: String
+ KEY_END_TIME: String
+ KEY_MONDAY: String
+ KEY_TUESDAY: String
...
+ KEY_SUNDAY: String
+ CREATE_TABLE: String
+ DELETE_TABLE: String
ProfileTable
+ TABLE_NAME: String
+ KEY_PROFILE_ID: String
+ CREATE_TABLE: String
+ DELETE_TABLE: String
ScheduleProfileTable
+ TABLE_NAME: String
+ KEY_SCHEDULE_ID: String
+ CREATE_TABLE: String
+ DELETE_TABLE: String
NotificationTable
+ TABLE_NAME: String
+ KEY_NOTIFICATION_ID: String
+ KEY_SCHEDULE_ID: String
+ KEY_APP_NAME: String
+ KEY_TIME_RECEIVED: String
+ KEY_DATE_RECEIVED: String
+ KEY_TEXT: String
+ CREATE_TABLE: String
+ DELETE_TABLE: String
<<Interface>>
BaseColumns
android.provider.BaseColumns

Class Diagram - Event Bus Classes

This diagram shows a closer look at the classes that are part of the event bus (as shown in the high-level app architecture diagram) or are closely depended on by said classes. These classes are mainly responsible for scheduling jobs to block notifications and listening for notifications to block.

An unlabeled arrow represents a dependency of one class upon another.



<<Interface>>
JobCreator
(Android-Job API)
Implements
NotificationJobCreator
- schedule: Schedule
+ create(String tag): void
Schedule
- repeat: boolean
- startTime: Date
- stopTime: Date
- daysOfWeek: String[]
- profiles: Profile[]
Profile
- blockList: String[]
Job
(Android-Job API)
Extends
Extends
<<Interface>>
NotificationListenerService
android.service.notification.NotificationListenerService
Implements
EndScheduleJob
- schedule: Schedule
# onRunJob(Paramsparams):
+ schedulePeriodic(): void
StartScheduleJob
- schedule: Schedule
# onRunJob(Paramsparams):
+ schedulePeriodic(): void
NLService
- profiles: Profile[]
- db: SQLiteDatabase
onNotificationPosted(StatusBarNotification sbn, NotificationListenerService.RankingMaprankingMap): void
cancelNotification(String key): void
insertNotification(StatusBarNotification sbn): void
SQLiteOpenHelper
android.database.sqlite.SQLiteOpenHelper
Extends
BroadcastReciever
android.content.BroadcastReceiver
Implements
Implements
SystemBroadcastReceiver
onReceive():void
NotificationBroadcastReceiver
onReceive(Context context, Intent intent):void
notifiedByNLService(StatusBarNotification[]):void
DatabaseHelper
- sInstance: DatabaseHelper
- DatabaseHelper(context : Context)
+ getInstance(context : Context): DatabaseHelper
+ onCreate(db: SQLiteDatabase): void
+ onUpgrade(db: SQLiteDatabase, oldVersion: int, newVersion: int): void

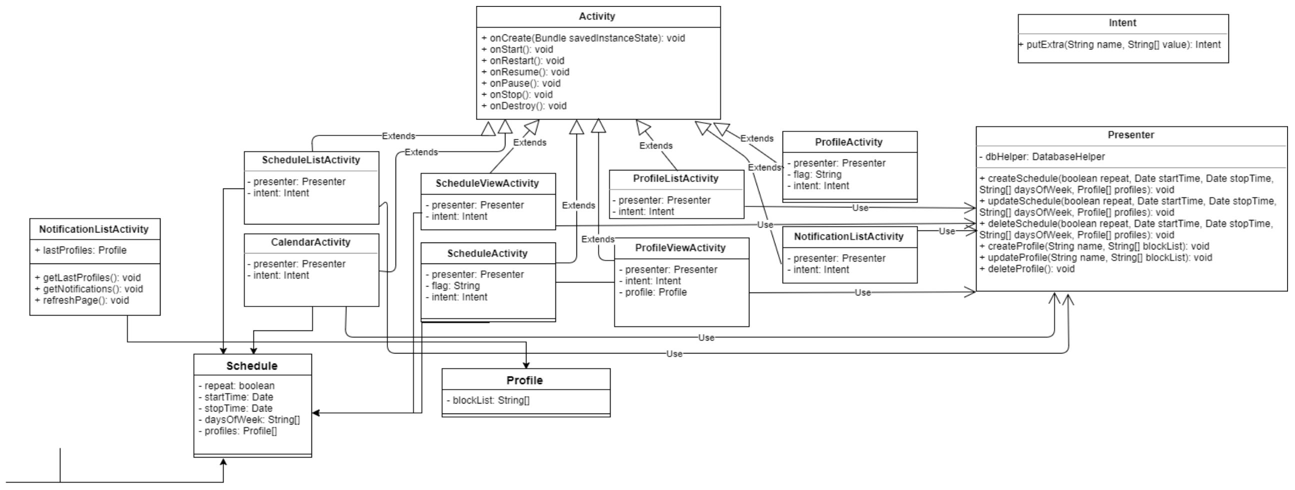
These classes differed fairly significantly in our implementation, because we did not end up using the Android-Job API to create and start schedules. Instead, we leveraged the native Android library AlarmManager, which also schedules alarms to go off at a certain time. Originally, we intended to use Android-Job because it appeared to be newer, and it offers other features, such as integration with Android’s other scheduling libraries (JobScheduler and GCMNetworkManager), but we found that we actually will not need this functionality. AlarmManager was a simpler way to sufficiently meet our scheduling needs, and so we didn’t need to create many extra classes. The scheduling is all done through the static ProfileScheduler class, which controls most of the AlarmManager logic and communicates with NLService.

Class Diagram - Presenter/View Classes

This diagram shows a closer look at the classes that are part of the presenter and view. The Presenter class processes data from the UI elements and communicates with the database to provide functionality for user interactions. The Intent class is used to send data and switch between different activities. All activities extend the Activity class and overwrite the methods shown. ScheduleListActivity represents the view for displaying a comprehensive list of the user's schedules. ScheduleViewActivity allows the user to view a specific schedule in daily or weekly calendar form. This view will also show a list of the profiles currently active for that schedule. CalendarActivity will display all of the user's active schedules in daily or weekly calendar form. ScheduleActivity is displayed when the user is creating, editing, or deleting a schedule. The flag string identifies which action the user wishes to take, so that input fields can be populated/disabled when necessary. Similarly, ProfileActivity is used for creating, editing, or deleting a profile. ProfileListActivity shows a comprehensive list of the user's profiles. This list can be filtered to show only active profiles. ProfileViewActivity shows the details of a specific profile, including a list of the apps in it. NotificationListActivity allows the user to view the notifications blocked by their profiles.

In addition, we will be using the Android Week View library to create the layouts for displaying the user's schedules. This library allows us to create weekly and daily calendar views with custom styling, which fulfills our needs for displaying the user's schedules in weekly and daily calendar formats.

Some of the presenter classes were removed from the design. Adaptor classes were used in their place to populate the UI with appropriate data. This change was made partly because the project did not need to asynchronously retrieve data from the database. This change was also made due to our use of the Room Persistence Library to create our database. By using the Room Persistence Library, we significantly simplified the processes of database creation and data retrieval, and thus eliminated the need for a separate presenter layer to handle communication with the database. Instead, the activities can now directly communicate to the database.



Class Diagram - Model Classes

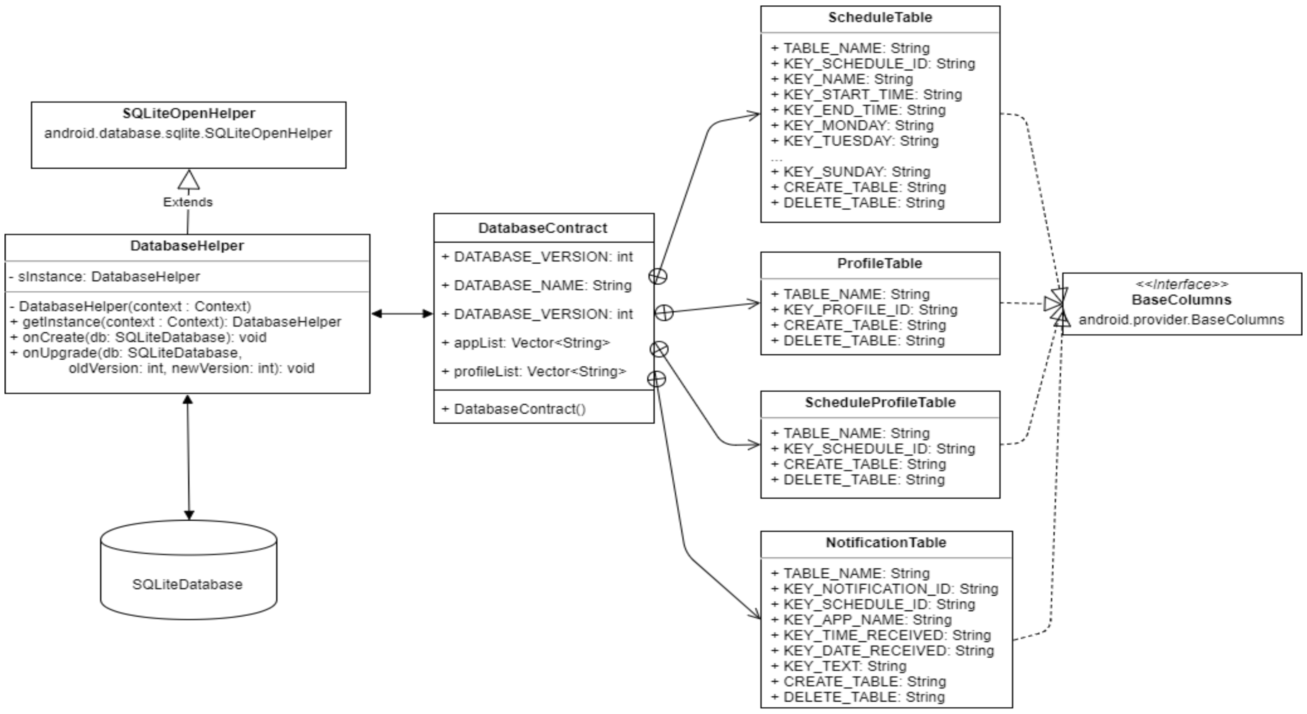
This diagram shows how the access of database is managed in our application. The DatabaseHelper class extends the native SQLiteOpenHelper class, which is used to access the SQLiteDatabase. The getInstance method of the class is static and synchronized, which guarantees that only one instance of DatabaseHelper can be created, and only one part of the application can access the database at a particular time. This Singleton structure will help manage memory and prevent memory leak. The DatabaseContract class is the database schema, or guidelines of how the database is set up. The DatabaseContract class is abstract and will never be instantiated. Rather, it's only for referring to the constants and variables defined in it. Each different table implements the native BaseColumns interface, which allows them to inherit a primary key field called _ID that some Android classes such as cursor adaptors would expect it to have. Any class within the application that needs to access the information in the data would call DatabaseHelper.getInstance(), at which point a reference to the DatabaseHelper would be returned, and if the database was not created it would be created at that instance. The DatabaseHelper reference can then be used to retrieve the SQLiteDatabase itself using .getWriteableDatabase() or .getReadableDatabase(), depending on the purpose.

We have completely changed the way we approach the database (model) part of our application. We decided to use the new Room Persistence Library that Android provides. The Room Persistence Library essentially provides an extra abstraction layer over SQLite, which allows the app to interact with the database more easily and fluently. There are four parts to our implementation of the database:

(a) AppDatabase, which is the main access point to the database. The annotations define the entities in the database, and the class itself defines the list of data access objects in the database.
(b) Entity: each entity represents a class that holds a database row, and the AppDatabase creates a database table for each of the entity that is defined.
(c) DAO: Data Access Objects (DAOs) are responsible for defining the methods that access the database. Each entity has a corresponding DAO, along with methods defined that allows the app to retrieve, change, or store information related that particular entity.
(d) TypeConverters: A type converter is defined for every field of an entity that is not a primitive type. Used alongside the Gson library that Google provides, we can convert any member variables, including lists of objects, into a String, and vice versa. This simple process allows one-to-many relationships to exist in the database.

Previously, we had a DatabaseHelper class that extended the SQLiteOpenHelper, alongside a DatabseContract interface that defined the structures of different tables as well as the way information was retrieved. However this structure proved itself not very intuitive, and difficult to implement, especially since the nature of our application requires a lot of one-to-many relationships.

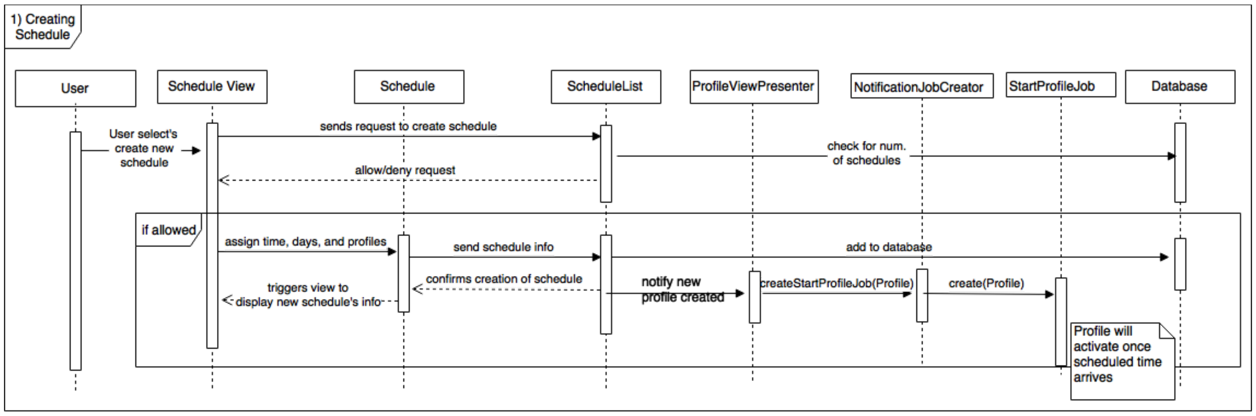
The main advantages of using Room Persistence Library is that we do not have to worry about the structuring of different tables to support our one-to-many relationships of different classes. In addition, Room Persistence Library supports LiveData, which allows us to change variables of different classes and have that change reflected in the database simultaneously.

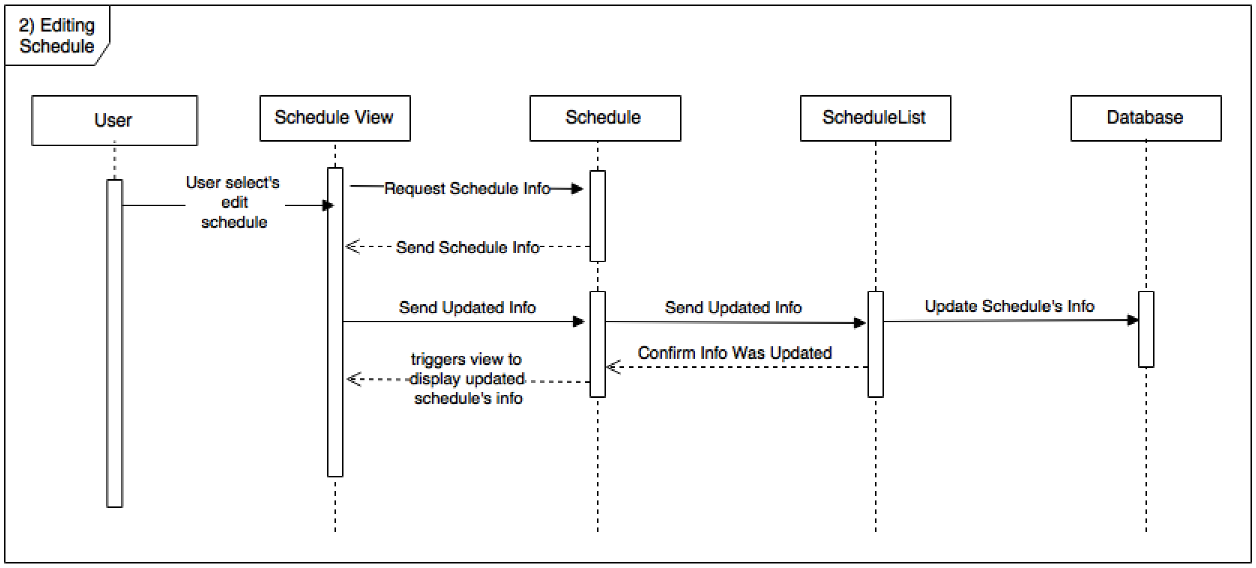


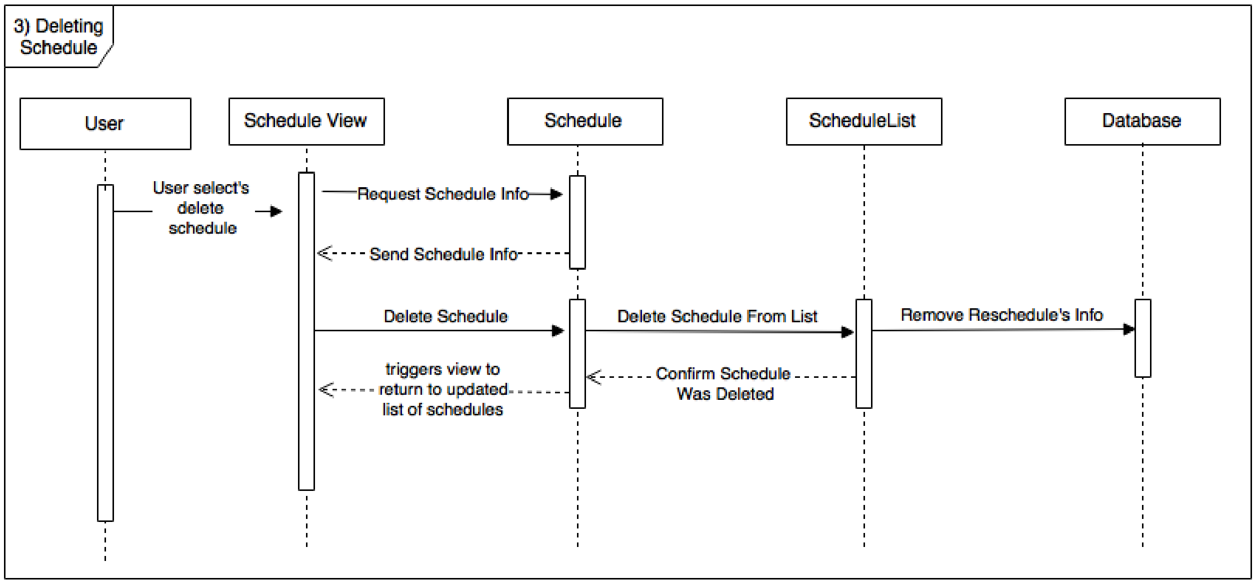
Sequence Diagrams

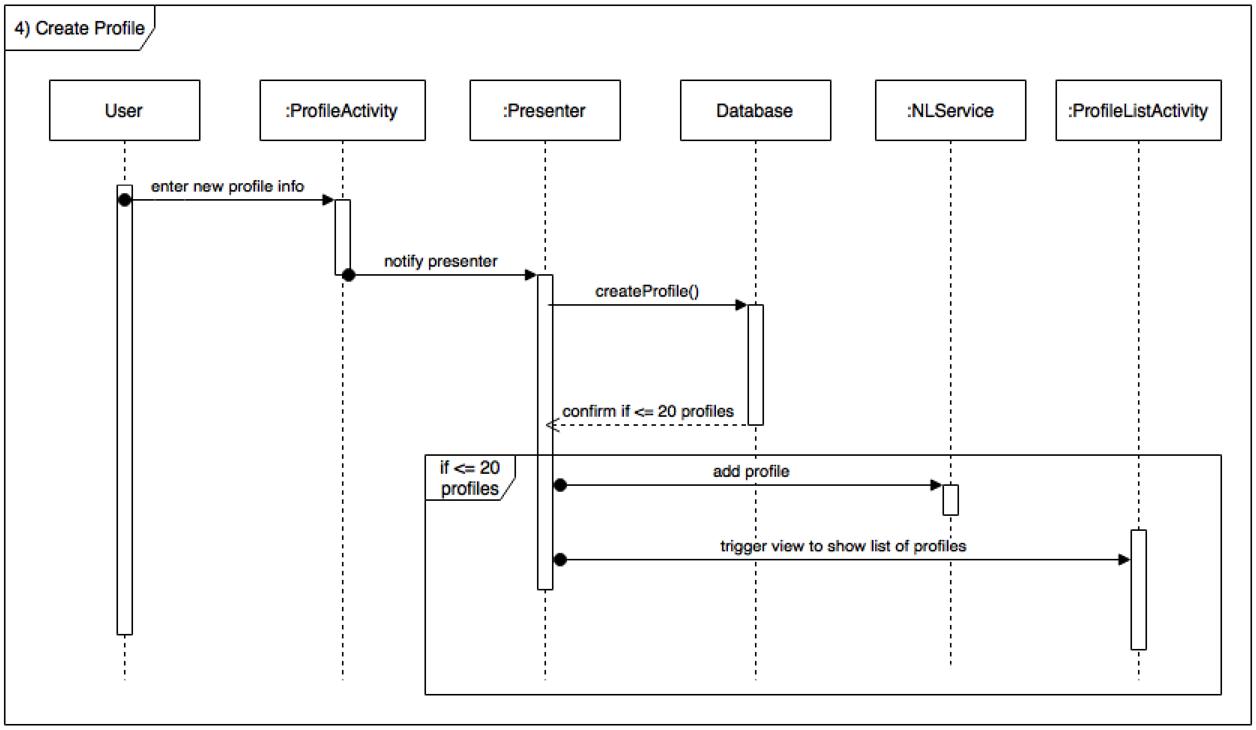
This diagram shows the flow of interaction between objects for different parts of the architecture of the Focus! App.Diagram (1) shows the order of creating a new schedule and its interaction with the database and the NotificationJobCreator which enables profile to wait till their scheduled time to activate, it also checks if the maximum number of schedules has already been created.Diagram (2) and (3) similarly show how to edit and delete profiles by connecting with the database.Diagram (4) shows the creation of a profile, the profile will not be linked to the NotificationJobCreator until it has been added to a schedule or manually activated.Diagram (5) and (6) shows process of editing and deleting profiles.Diagram (7) shows the steps resulting from the user manually turning on a profile that is no tied to an active schedule.Diagram (8) shows the reverse steps of when the user manually ends a profile.Diagram (9) shows how the user will receive a push notification when they enter an app that is currently being blocked.Diagram (10) shows what happens when a user checks the list of notifications that were received when a schedule was active. Diagram (11) shows the progress of when a notification was sent from an app that was in an active profile.

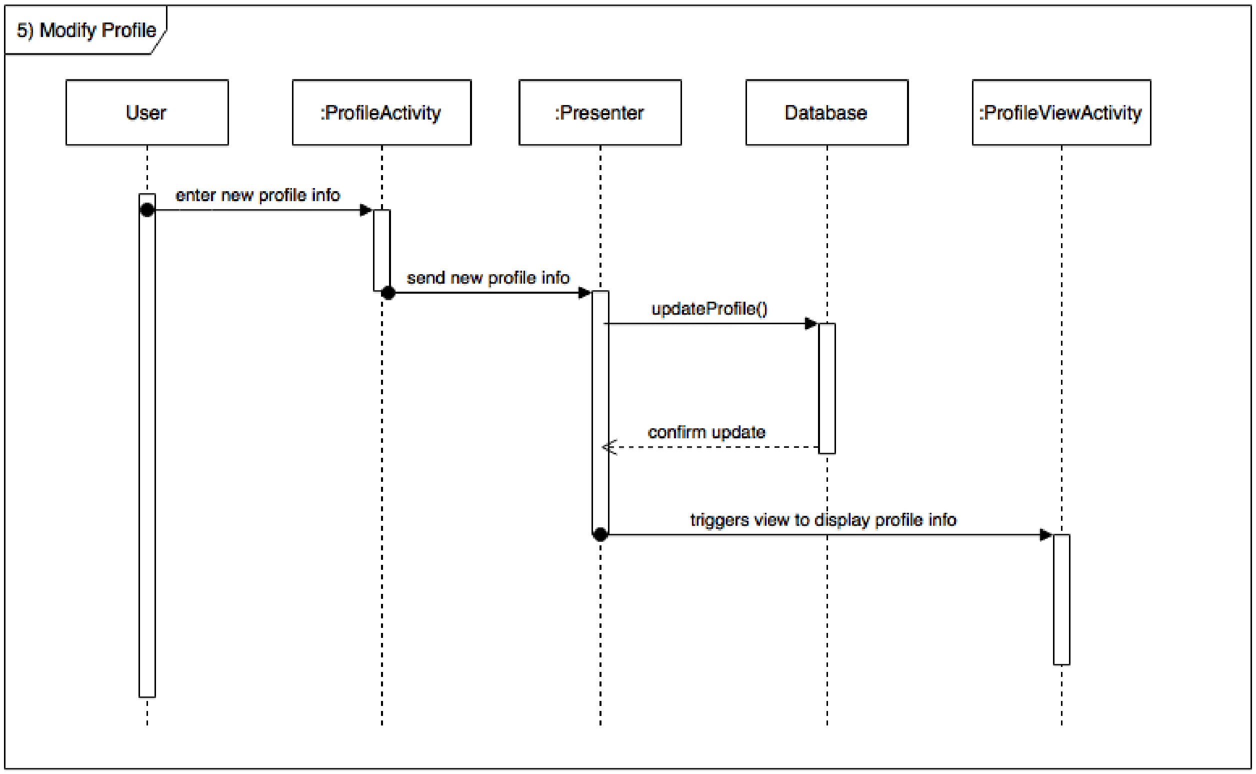
The sequence diagrams, for the most part, are still reflective of our implemented design. The user interactions and database are consistent with our intended design; however, as previously mentioned, some of the presenter classes and Android-Job related classes were removed from the design. Some of the presenters were superfluous because we used Android Room Persistence for our databases, which greatly reduced the amount of interaction needed to present the UI and react to changes. Android-Job wasn't used because we substituted AlarmManager in instead. Some parts of our design were not realized, so they do not exist in our implementation, such as editing/deleting a profile/schedule (diagrams 2, 3, 5, 6) and sending a notification when a user enters an app that is currently being blocked (diagram 9).

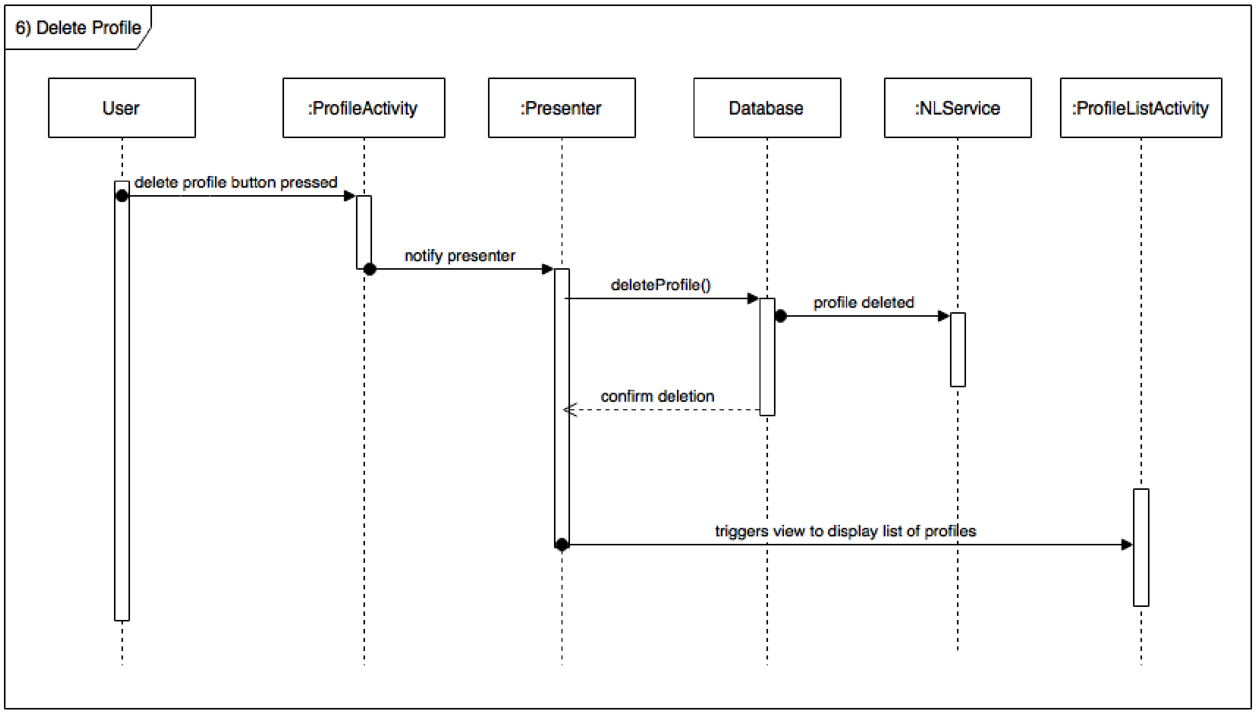

1) Creating Schedule
User
Schedule View
Schedule
ScheduleList
ProfileViewPresenter
NotificationJobCreator
StartProfileJob
Database
User select's create new schedule
sends request to create schedule
check for num. of schedules
allow/deny request
if allowed
assign time, days, and profiles
send schedule info
add to database
confirms creation of schedule
triggers view to display new schedule's info
notify new profile created
createStartProfileJob(Profile)
create(Profile)
Profile will activate once scheduled time arrives

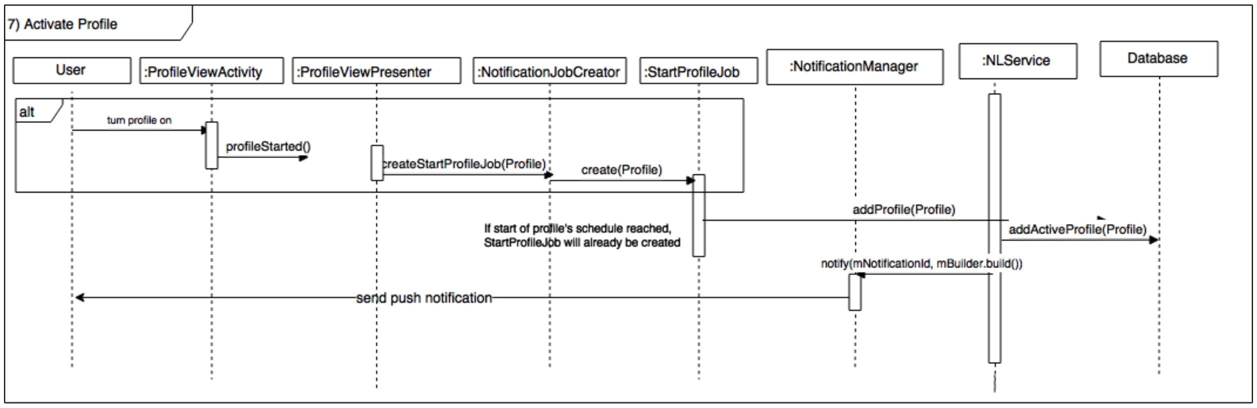

2) Editing Schedule
User
Schedule View
Schedule
ScheduleList
Database
User select's edit schedule
Request Schedule Info
Send Schedule Info
Send Updated Info
Send Updated Info
Update Schedule's Info
Confirm Info Was Updated
triggers view to display updated schedule's info

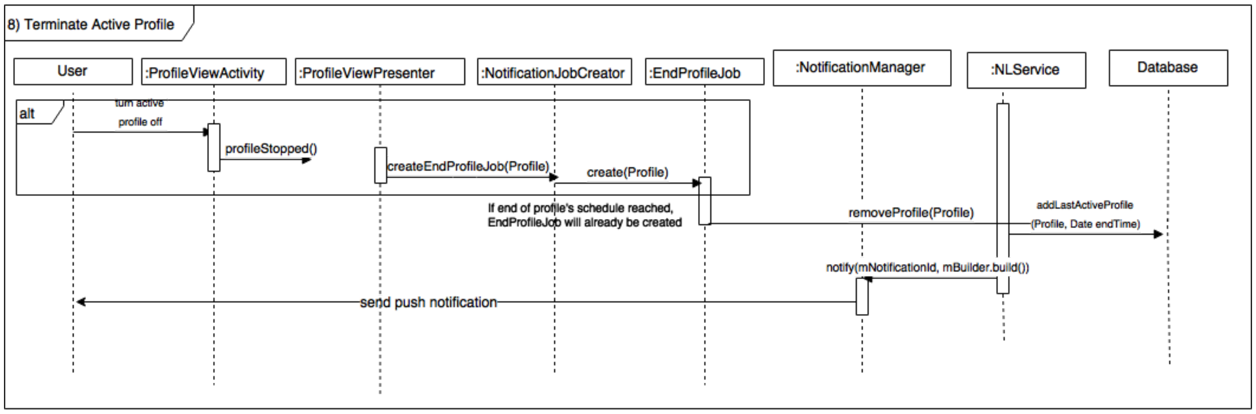

3) Deleting Schedule
User
Schedule View
Schedule
ScheduleList
Database
User select's delete schedule
Request Schedule Info
Send Schedule Info
Delete Schedule
Delete Schedule From List
Remove Reschedule's Info
Confirm Schedule Was Deleted
triggers view to return to updated list of schedules

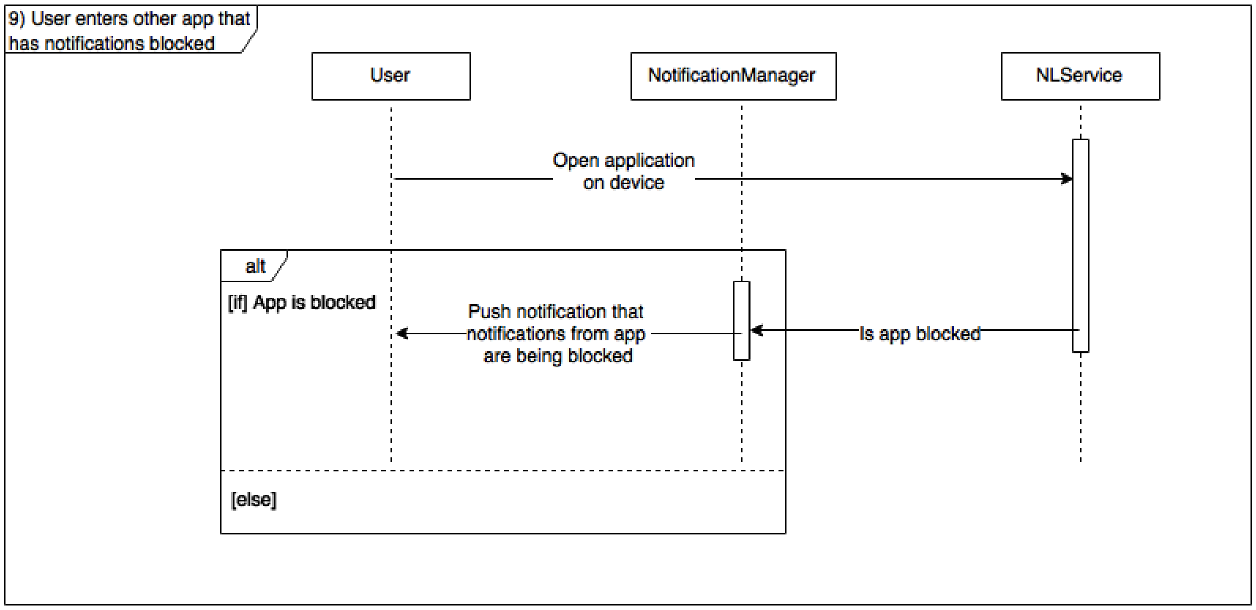

4) Create Profile
User
:ProfileActivity
:Presenter
Database
:NLService
:ProfileListActivity
enter new profile info
notify presenter
createProfile()
confirm if <= 20 profiles
if <= 20 profiles
add profile
trigger view to show list of profiles

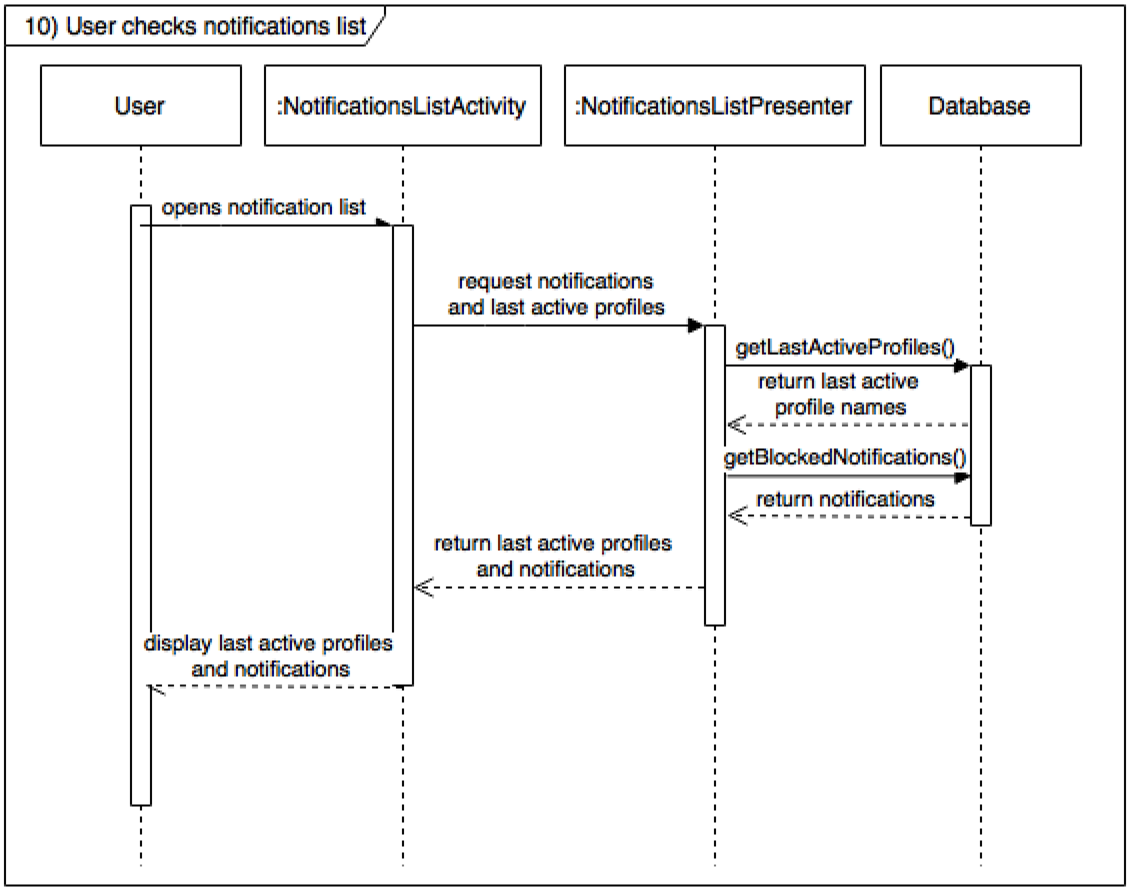

5) Modify Profile
User
:ProfileActivity
:Presenter
Database
:ProfileViewActivity
enter new profile info
send new profile info
updateProfile()
confirm update
triggers view to display profile info

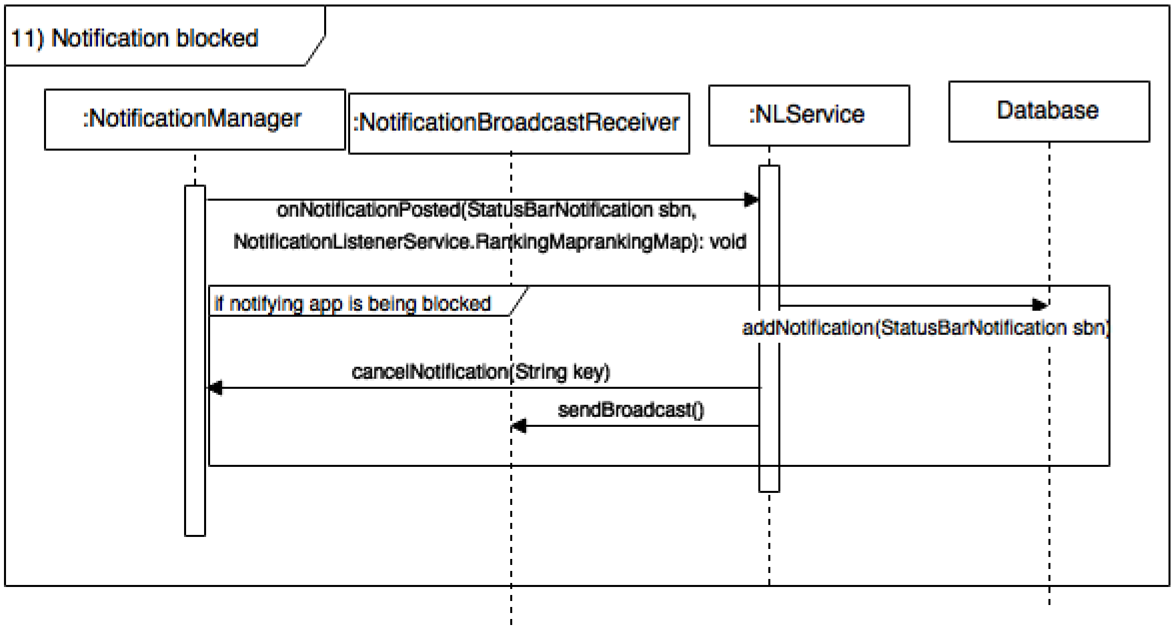

6) Delete Profile
User
:ProfileActivity
:Presenter
Database
:NLService
:ProfileListActivity
delete profile button pressed
notify presenter
deleteProfile()
profile deleted
confirm deletion
triggers view to display list of profiles


7) Activate Profile
User
:ProfileViewActivity
:ProfileViewPresenter
:NotificationJobCreator
:StartProfileJob
:NotificationManager
:NLService
Database
alt
turn profile on
profileStarted()
createStartProfileJob(Profile)
create(Profile)
If start of profile's schedule reached, StartProfileJob will already be created
addProfile(Profile)
addActiveProfile(Profile)
notify(mNotificationId, mBuilder.build())
send push notification


8) Terminate Active Profile
User
:ProfileViewActivity
:ProfileViewPresenter
:NotificationJobCreator
:EndProfileJob
:NotificationManager
:NLService
Database
alt
turn active profile off
profileStopped()
createEndProfileJob(Profile)
create(Profile)
If end of profile's schedule reached, EndProfileJob will already be created
removeProfile(Profile)
addLastActiveProfile (Profile, Date endTime)
notify(mNotificationId, mBuilder.build())
send push notification


9) User enters other app that has notifications blocked
User
NotificationManager
NLService
Open application on device
alt
[if] App is blocked
Push notification that notifications from app are being blocked
Is app blocked
[else]


10) User checks notifications list
User
:NotificationsListActivity
:NotificationsListPresenter
Database
opens notification list
request notifications and last active profiles
getLastActiveProfiles()
return last active profile names
getBlockedNotifications()
return notifications
return last active profiles and notifications
display last active profiles and notifications


11) Notification blocked
:NotificationManager
:NotificationBroadcastReceiver
:NLService
Database
onNotificationPosted(StatusBarNotification sbn,
NotificationListenerService.RankingMaprankingMap): void
if notifying app is being blocked
addNotification(StatusBarNotification sbn)
cancelNotification(String key)
sendBroadcast()

6. REQUIREMENT CHANGE

This section describes two scenarios in which the requirements for the application have been changed. The following questions will be answered for each scenario:

1. Does this change your design?
2. If not, why not?
3. If so, what should be changed and how?

(1) *Assume, that the user wants to be able to keep history of their uses of Focus! Your application should show the summary of uses during past several days, weeks or months. This summary should visualize statistics such as total hours focused, total hours per profile, total number of weeks used per schedule, total number of uninterrupted Focus!-ed intervals, the profile with maximum/minimum number of uninterrupted intervals, etc. Users will need to be able to back up and download this data as csv files and upload it to Focus on a different device. Users will also want to be able to sync their data between their Android devices via their Google accounts.*

Implementing the above functionality will require us to significantly modify our current design. First, we will need to modify our Schedule and Profile classes to be able to provide the requested statistics. The Schedule class will need member variables to track the total hours and weeks used for a schedule, which can be summed for each of the user's schedules to get a total number of hours/weeks used across the whole application. This sum would essentially be the total time the user has "focused" by using the application. The Profile class will need member variables to store the total hours used (which can be retrieved by summing the total hours for each schedule the profile has been assigned to), as well as the number of uninterrupted intervals. The profile with the minimum/maximum number of uninterrupted intervals can be found by comparing this member variable across all of the user's profiles. The UI will be expanded to display these statistics.

We will also need to add login functionality for the application, since it will no longer be used exclusively on one device for each user. This will require us to expand our database to store account information for all users, including the statistics discussed in the previous paragraph. Additionally, we will need to implement networking and deploy a server so that users can log onto Focus from different devices.

(2) *Assume, that Focus! has been commercialized and now it is used by schools to ensure their students stay present and Focus!-ed during class time. You will need to connect Focus! to user's Google account to authorize them. Focus should have functionality for its users to create "courses" and become an "instructor" for that course. Instructors will specify what applications they will need in their classes and hence they will allow using those, everything else is in blocked list. Instructors should also specify at what times and days the class takes place. Instructors can invite students by their email address (which will be entered manually) to join the course. The students can reject this invitation if it was not for them or was sent by mistake. If the invitation is accepted, the application on student's phone will create a schedule for the course and activate it. The instructor should be able to see the list of students who accepted invitation.*

Much of this functionality can be implemented by expanding upon our current design. Courses can be implemented as subclasses of the Profile class, with additional member variables to store the instructor, the list of students in the course, and the list of the course’s authorized applications. As before, users can assign schedules to profiles, so the instructor can create a schedule in the application that reflects the course schedule, and then assign it to their course. Additionally, instructor- and student-specific functionality can be implemented by creating Instructor and Student classes from a base User class.

As in the previous scenario, we will need to add login functionality for the application via the user's Google account. We will also need to expand the UI to have forms that allow instructors to set up courses and invite students by email address. Additionally, we will need to implement networking to allow multiple users to invite each other and share profiles and schedules.