Application Design v1.1

CSCI-310 Software Engineering

Professor: **Nenad Medvidović**

TA: **Shushan Arakelyan**

Fall 2017

Dream Team (Team 4)

Aarav Malpani	8739-8865-03	malpani@usc.edu
Bowei Chen	5327-0036-21	boweiche@usc.edu
Prateek Bhatia	2013-5693-40	prateekb@usc.edu
Shatrujeet Naruka	2782-3669-61	naruka@usc.edu
Tushar Singhal	2897-7369-04	tsinghal@usc.edu

Preface **3**

Introduction **3**

Architectural Design **4**
- Overview 4
- Description 4
- Security Analysis 5

Detailed Design **6**
- Package Diagrams 6
  - Overview 6
  - Description 7
- Class Diagrams 11
  - Overview 11
  - Description 12
- Sequence Diagrams 30

Requirement Changes **43**
- Change 1 43
- Change 2 44

1. Preface

This document outlines the design elements for the Android app *Focus!*, from the broad architecture design to the individual methods and classes. This document also details how different classes interact with one another within a package and between packages. All information necessary to build all individual classes, packages, and components necessary to facilitate the build of this application are included in the scope of this document. This document has also been updated based on necessary changes in implementation details.

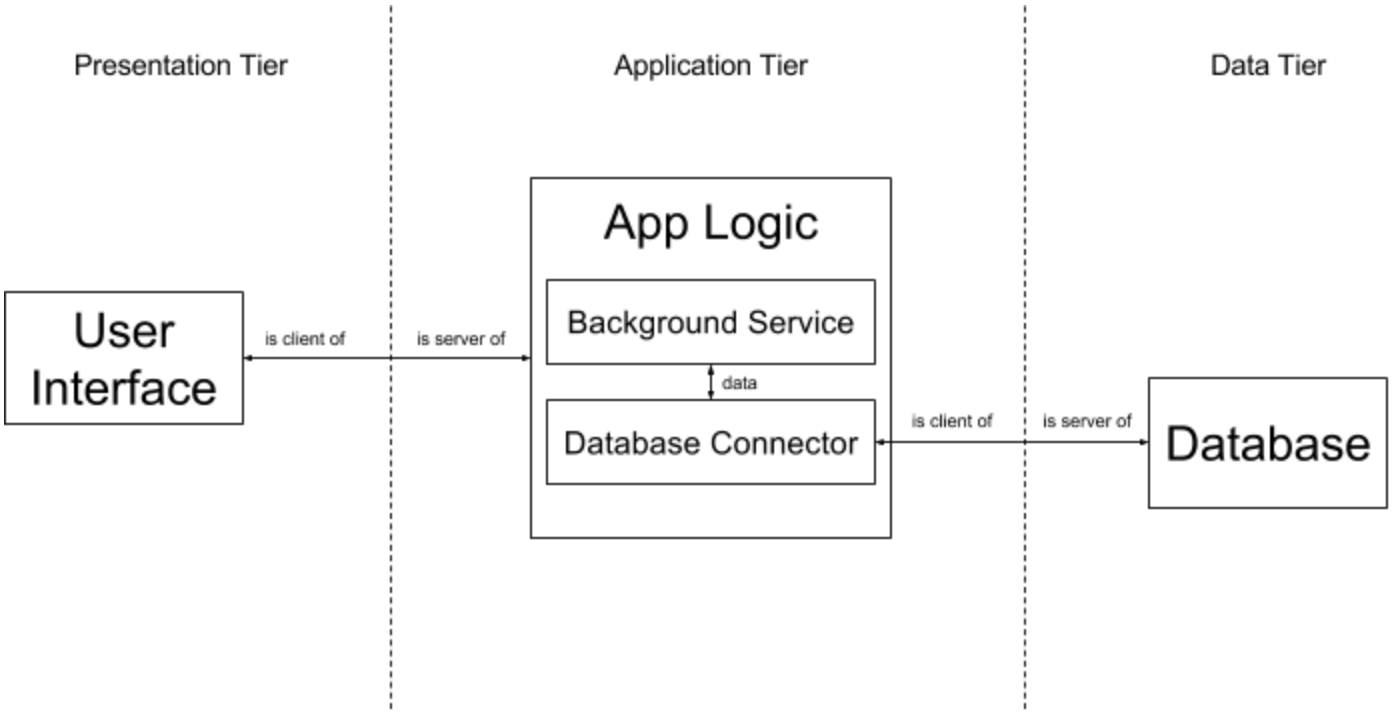
2. Introduction

Phones have constantly evolved and have today become an extension of people, notifying us about breaking news, social events, messages from friends, etc. Constant notifications are distracting and can make phone users unable to focus on the tasks at hand due to constant interference. *Focus!* seeks to alleviate this problem by blocking apps and notifications in a user-configured manner so that users can focus on their work and other activities.

Focus! is an Android application that allows users to block distracting applications and their notifications for a specified amount of time. The app allows users to define profiles to block certain applications and notifications at specified times and days.

3. Architectural Design

3.1. Overview



Focus! will follow a three-tier client-server design: the user interface will be the client to the application logic; the database interface then serves as a client to the database server. Application logic also serves as a server to user interface and client to database. Background service will have access to the database connector. This connector allows the background service to access the most recent copy of the data and make logic decisions based off of the information.

This app relies heavily on the application logic, from the app blocking feature to the notification listening feature. All of these features require a background Thread listening on Android's native APIs for changes and act on those changes accordingly. Without background Threads, i.e., some persistent logic running in the background, listening for changes in application state, the features required of this application will not be possible. Due to this constraint, making application logic the most integral component of this architecture allows for the best balance among dataflow efficiency, simplicity in design, and separation of concerns.

3.2. Description

The database connector in application logic and the database will have a client-server relationship. The database will hold the most up-to-date information and pass relevant information, when queried by controllers, to the relevant recipients. The database is the server as it holds all the

relevant data; the database connector is a client, as it holds no data. It only serves as a gateway between other application logic and the database itself.

The application logic component, in addition to being a client to the database, is also a server to the user interface; that is, application logic and the database are also in a client-server relationship. The user interfaces will query the application logic to pull the latest information from the database. The user interfaces will not be able to access the database directly. Rather, information in the database will be made available through the database connector in the application logic component, specified in the previous paragraph. The application logic then processes the raw data fetched from the database and passes it onto the UI components that made the request.

The UI component will have a unidirectional connection to the background service component, from the latter to the former, allowing background service to notify UI for any changes in internal states; the UI component will also have a bidirectional connection to the database, allowing for addition/modification of database be made by the UI component.

3.3. Security Analysis

This database will hold data related to profiles, schedules, and Android notifications. While profiles and schedules are not confidential information, they are considered sensitive and user information can be extracted. Profiles may be able to reveal partial lists of applications installed on the user's devices and may allow attackers to profile users based on broad scopes, such as gender and locality. Schedules can be more revealing, as they include temporal information of a user's usual schedule.

Android notifications, on the other hand, are considered confidential due to the nature of information they contain. Information in an Android notification typically includes the application name, the title of the notification, and the text of the notification. In the context of a messaging application, a message notification will usually include the sender's name and the message itself. Most users consider private messages confidential and therefore would want proper security measures to handle these information.

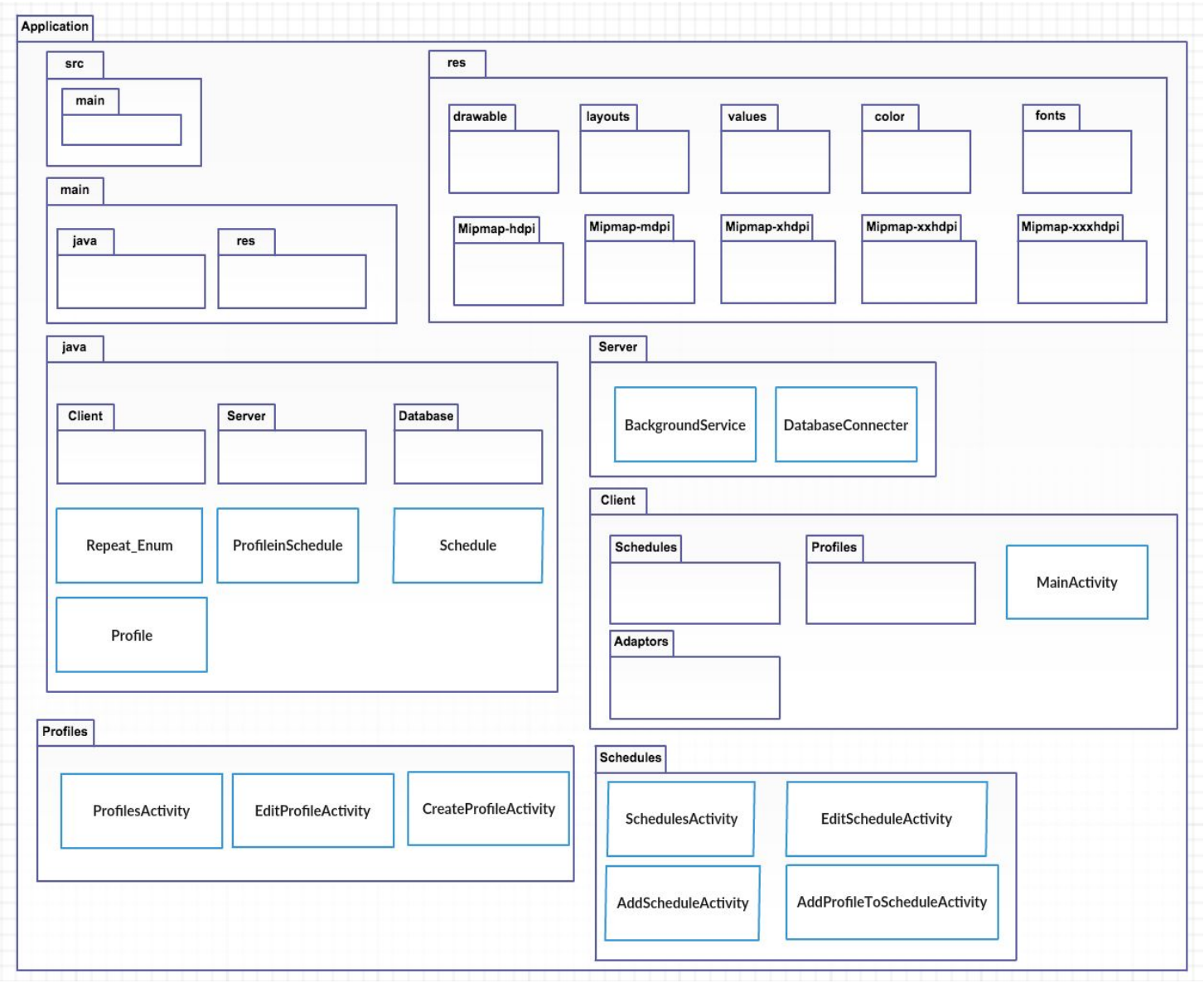
With the sensitive nature of the data this application processes, it is all the more critical to follow this architecture design and restrict database access in this application to only the database connector as to prevent unauthorized access, modification, and addition to the database.

Overall, the architectural implementation did not diverge from its original design (it was only specified that the UI will have a client-server relationship with the background service component). However, we decided, as a team, that having background service handle all database transactions may place unnecessary burden to its threads.

4. Detailed Design

4.1. Package Diagrams

4.1.1. Overview



4.1.2. Description

The package diagrams of the app *Focus!* are modeled in sync with the system architecture i.e. three-tier client-server design. The package hierarchy begins with the Application package that comprises of all other packages that are useful in making the app function. The organization of various packages and classes is designed not only to follow the proposed architecture, but also to keep the code clean and organized, so that no member in the team has any problem in understanding and modifying the code during the app-building process. This will also enable the engineers to accommodate the changing needs of the customer during the development of the application.

Below is a breakdown of the package hierarchy along with brief explanation of what classes and/or files these packages will comprise of.

Application/**src**

- The top level package "`Application`" consists of a source package, namely "`src`". The package "`src`" has all the source code to run the app.

Application/src**/main [UPDATE]**

- The package "`src`" consists of a package, namely "`main`". The package "`main`" has all the java code to run the app and the resources that are needed in the app. This package will also have an `AndroidManifest.xml` file, which defines which theme/style every activity in the app will follow.

Application/src/main/**res**

- The package "`res`" contains all the resources : colors, layouts, mipmap for launcher icons, strings, fonts and other images.

Application/src/main/res/**drawable**

- The package "`drawable`" will contain Bitmap files (.png, .jpg, .gif), i.e., images needed in the app like title.png (show the title on main screen), add.png (image for add button), and clock.png (image to show clock is running).

Application/src/main/res/**layouts**

- The package "`layouts`" will include the XML files that define a user interface layout. Some of the layouts that *Focus!* will use are -

dropdown.xml	A layout for the drop down list with checkboxes that pops up on clicking the "Repeat" spinner and "Block apps" button.
profileListItem.xml	A layout for how a Profile will be listed for the user to view. There will be toggle button next to each profile name which the user can use to activate now/deactivate a profile.
showProfilesListItem.xml	A layout to show a list of all profiles with checkboxes to select multiple profiles.

scheduleListItem.xml	A layout for how a Schedule will be listed for the user to view. There will be a toggle button next to each schedule indicating active/inactive schedules.
calenderViewItem.xml	A layout for how profiles will be listed according to each day of the week. This will include a cross button to delete the profile, a label for the scheduled time, a time remaining label when profile is active and an indicator to depict whether a profile in the schedule is active.
notificationPop.xml	A layout of how notifications will be released to the user when a profile becomes inactive. This will include the name of the app and number of notifications that were missed.
timerPop.xml	A layout of time duration input will be asked from user when they want to activate the profile instantly. This will have a TimePicker to select minutes or hours - from 10 minutes to 10 hours.

[UPDATES]

Activity_main.xml	These layouts will define how each activity's UI will look like.
Activity_editProfile.xml	
Activity_editSchedule.xml	A layout to edit an existing Schedule in the Database. This layout is responsible for adding new Profiles to Schedule and removing Profiles in schedule as well. This will have a calendar like appearance with Days of the week written, under which the scheduled profile are also displayed. Each schedule profile item will have a remove button. The layout will also have three buttons for Saving changes, Discarding changes made and Deleting the schedule completely. The layout also has an editText field in case the user wishes to change the name of the schedule.
Activity_createProfile.xml	
Activity_createSchedule.xml	A layout to create a new Schedule object in the Database. This layout is responsible to take the user's request for a new schedule and show him how the schedule will look like before it is created. The layout also has all the items mentioned in Activity_editSchedule.xml as well with the same functionalities.
Activity_Schedules.xml	A layout with a list of all Schedules present the database along with a toggle switch for activating the schedule and deactivating it. This switch works like the repeat weekly button mentioned in earlier documents which was removed after talking to the customer as mentioned later in this document. The layout also has a button on the top-right hand corner to add a new Schedule that directs the user to Activity_createSchedule.xml.
Activity_Profiles.xml	

Application/src/main/res/**values**

- The package "`values`" contains XML files that contain simple values, such as Strings and Integers.

Application/src/main/res/**color**

- The package "`color`" contain an XML file that defines the RGB values of the colors that will be used in the app.

Application/src/main/res/**fonts**

- The package "`fonts`" contain an XML file that defines the fonts that will be used in the app.

Application/src/main/res/**Mipmap-hdpi**
Application/src/main/res/**Mipmap-mdpi**
Application/src/main/res/**Mipmap-xhdpi**
Application/src/main/res/**Mipmap-xxhdpi**
Application/src/main/res/**Mipmap-xxxhdpi**

- These packages contain the drawable files for different launcher icon densities.

Application/src/main/**java**

- The package "`java`" consists of all the Java classes i.e. the Java code to run the app. It is designed on the basis of the system architecture and hence comprises of three packages - "`Client`," "`Server`," and "`Database`," which correspond to the components of the three-tier client-server architecture.
- Alongside these three packages, "`java`" also includes four classes - `Repeat_enum.java`, `ProfileInSchedule.java`, `Profile.java` (a class that serves as a template for a single profile), and `Schedule.java` (a class that serves as a template for a single schedule). The interfaces (functions) and variables of these four classes will be used by all three packages that are in "`java`". Hence, these classes are placed higher up in the hierarchical order as compared to the other classes in the application.

Application/src/main/java/**Client**

- The package "`Client`" is designed in accordance with the "User Interface" component of the system architecture. This package consists of all the classes that cater to the UI of the app along with any other supporting classes.
- In Android, the user actually looks at activities on his screen as he/she navigates through the app. These activities are responsible for taking the user input and passing it on to the "Application Logic" component (which acts as a server to the UI). Hence, all java classes of these activities will come under the "`Client`" package.
- There will be `MainActivity.java` class responsible for the main screen and two other packages - Profiles and Schedules which will have activities for displaying all profiles and schedules made by the user.

Application/src/main/java/**Server**

- The package "`Server`" is designed in accordance with the "Application Logic" component of the system architecture. It comprises of two classes namely - `DatabaseConnector.java` and `BackgroundService.java`.
- The class `DatabaseConnector.java` is designed in accordance with the "Database Interface" component of the app logic, whereas `BackgroundService.java` is related to the

component “Background Thread”. These classes are placed in “`Server`” package because they are responsible for interacting both with the `Client` and `Database`.

Application/src/main/java/**Database**

- This package has the database in local storage that stores all the profiles and schedules that the user makes.

Application/src/main/java/Client/**Profiles**

- The package “`Profiles`” contains all the classes that are associated with profiles that the user creates, views and modifies. These classes are `ProfilesActivity.java`, `EditProfileActivity.java`, and `CreateProfileActivity.java` which are responsible for the UI of the features related to profiles.

Application/src/main/java/Client/**Schedules**

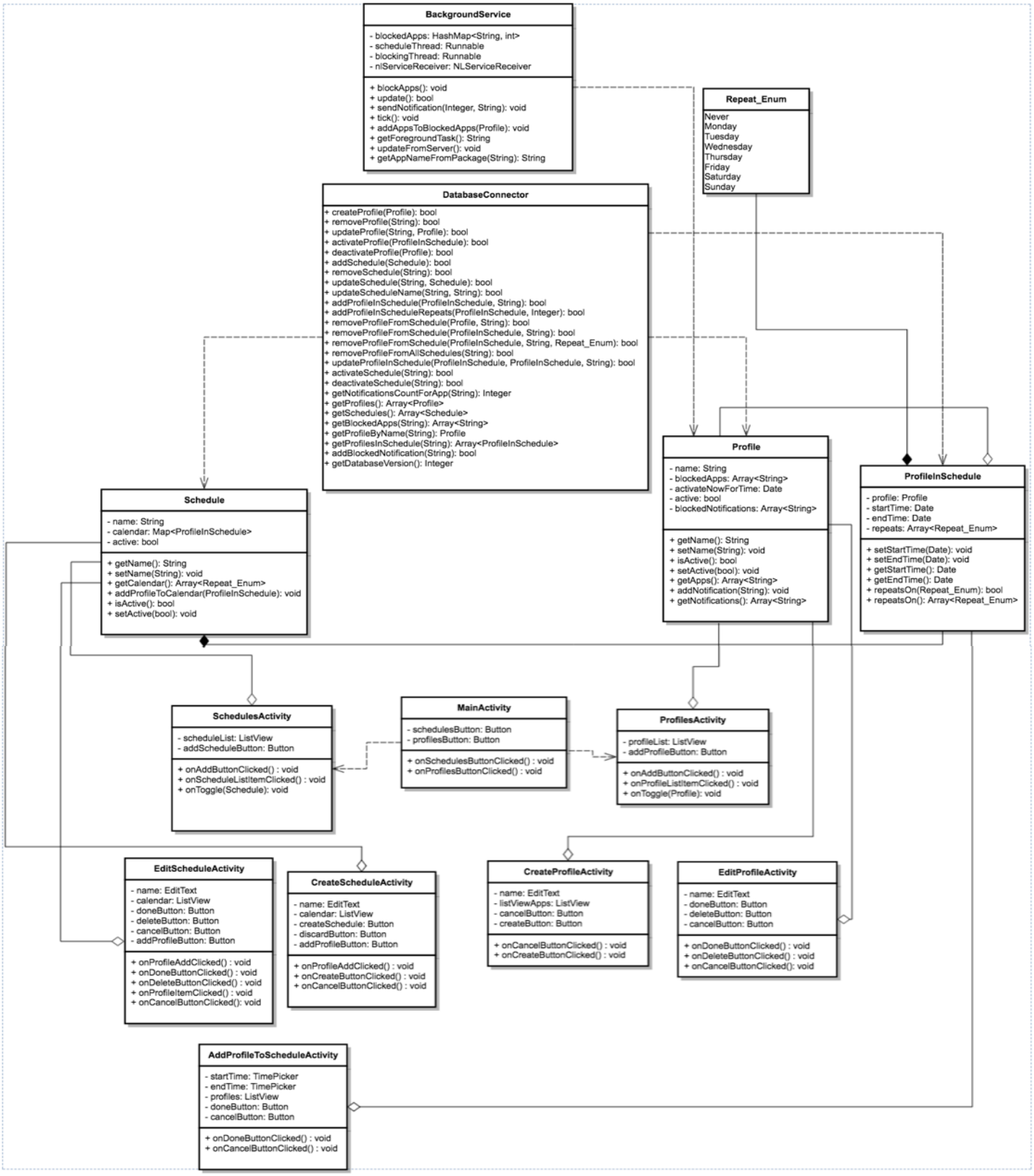
- The package “`Schedules`” contains all the classes that are associated with schedules that the user creates, views and modifies. These classes are `SchedulesActivity.java`, `EditScheduleActivity.java`, `AddScheduleActivity.java`, and `AddProfileToScheduleActivity.java` which are responsible for the UI of the features related to schedules.

Application/src/main/java/Client/**Adaptors**

- The package “`Adaptors`” contains all the classes that are associated with adaptors that are used in list views in `SchedulesActivity.java`, `EditScheduleActivity.java`, `AddScheduleActivity.java`, `AddProfileToScheduleActivity.java`, `ProfilesActivity.java`, `EditProfileActivity.java` and `AddProfilectivity.java` which are responsible for the UI of the features related to schedules.

4.2. Class Diagrams

4.2.1. Overview



4.2.2. Description

In our software architecture and designs, classes involved in code implementation are divided into various packages based on our three-tier client-server architecture to facilitate an organized development of the application. There are many relationships between classes which follow the UML design. Below is the description of these relationships among classes.

The class DatabaseConnector.java has a dependency relationship i.e. depends on classes `Profile.java`, `Schedule.java` and `ProfileInSchedule.java`. This relationship is of dependency as the function parameters of `DatabaseConnector.java` use `Profile.java`, `Schedule.java` and `ProfileInSchedule.java`. `DatabaseConnector.java` does not declare instances of these three classes hence the relation is not associative.

`BackgroundService.java` depends on `Profile.java` as its functions use `Profile.java` as a parameter and the class itself doesn't declare instances of `Profile.java`.

`MainActivity.java` depends on `SchedulesActivity.java` and `ProfilesActivity.java` because both these activities are transitioned into from the `MainActivity.java`.

`Schedule.java` has a composition relationship with `ProfileInSchedule.java`. This is because the containing object- `Schedule.java` contains `ProfileInSchedule.java` object in a map which contains all profiles in a schedule. The relationship is such because `ProfileInSchedule.java` is of no use if `Schedule.java` doesn't exist.

`ProfileInSchedule.java` has a composition relationship with `Repeat_Enum.java`. This is because the containing object- `ProfileInSchedule.java` contains `Repeat_Enum.java` object in an array to select the days when a profile must be repeated. The relationship is such because `Repeat_Enum.java` is of no use if `ProfileInSchedule.java` doesn't exist.

`ProfileInSchedule.java` has an aggregation relationship with `Profile.java`. ProfileInSchedule.java define an instance of the associated class `Profile.java` within its class scope. This is why `ProfileInSchedule.java` aggregates `Profile.java` because `Profile.java` can exist on its own even if `ProfileInSchedule.java` doesn't exist.

`SchedulesActivity.java` has an aggregation relationship with `Schedule.java`. `SchedulesActivity.java` define many instances of the associated class `Schedule.java` within its class scope to show a list of schedules. This is why `SchedulesActivity`.java aggregates `Schedule.java` because `Schedule.java` can exist on its own even if `SchedulesActivity`.java doesn't exist.

`AddScheduleActivity`.java has an aggregation relationship with `Schedule.java`. `AddScheduleActivity.java` define an instance of the associated class `Schedule.java` within its

class scope when the user input is received to add a schedule. This is why `AddScheduleActivity.java` aggregates `Schedule.java` because Schedule.java can exist on its own even if `AddScheduleActivity.java` doesn't exist.

`EditScheduleActivity.java` has an aggregation relationship with `Schedule.java`. `EditScheduleActivity.java` define an instance of the associated class `Schedule.java` within its class scope to edit that schedule object. This is why `EditScheduleActivity.java` aggregates `Schedule.java` because `Schedule.java` can exist on its own even if `EditScheduleActivity.java` doesn't exist.

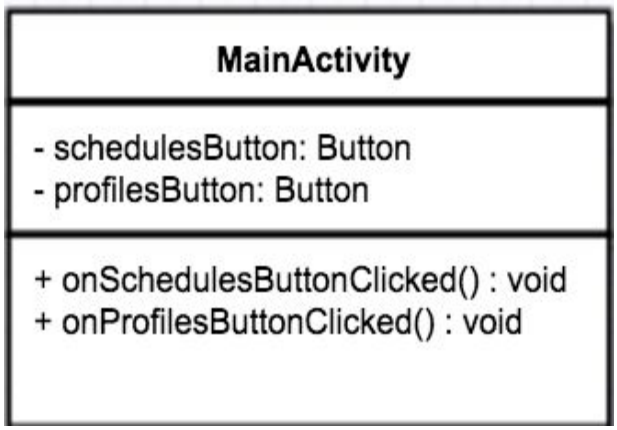
`ProfilesActivity.java` has an aggregation relationship with `Profile.java`. `ProfilesActivity.java` define many instances of the associated class `Profile.java` within its class scope to display all profiles. This is why `ProfilesActivity.java` aggregates `Profile.java` because `Profile.java` can exist on its own even if `ProfilesActivity.java` doesn't exist.

`CreateProfileActivity.java` has an aggregation relationship with `Profile.java`. `CreateProfileActivity.java` define an instance of the associated class `Profile.java` within its class scope when user input is received to add a profile. This is why `CreateProfileActivity.java` aggregates `Profile.java` because `Profile.java` can exist on its own even if `CreateProfileActivity.java` doesn't exist.

`EditProfileActivity.java` has an aggregation relationship with `Profile.java`. `EditProfileActivity.java` define an instance of the associated class `Profile.java` within its class scope when user edits a profile. This is why `EditProfileActivity.java` aggregates `Profile.java` because `Profile.java` can exist on its own even if `EditProfileActivity.java` doesn't exist.

`AddProfileToScheduleActivity.java` has an aggregation relationship with `ProfileInSchedule.java`. `AddProfileToScheduleActivity.java` define an instance of the associated class `ProfileInSchedule.java` within its class scope to add a profile to the schedule. This is why `AddProfileToScheduleActivity.java` aggregates `ProfileInSchedule.java` because `ProfileInSchedule.java` can exist on its own even if `AddProfileToScheduleActivity.java` doesn't exist.

Below are detailed breakdowns of UML class diagrams of each class.



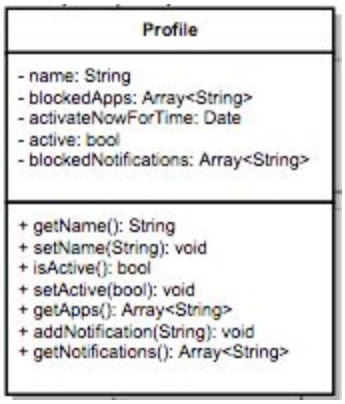
MainActivity: This class is an Android activity class that will be the launch-page/main-screen of the app. It will consist of the app's logo and two buttons: one to display all the schedules and the other for profiles that the user has made. This screen will require no login from the user.

- Private Members:

schedulesButton - (Button)	button to direct the user to the `SchedulesActivity` (class that shows list of all schedules)
profilesButton - (Button)	button to direct the user to the `ProfilesActivity` (class that shows list of all profiles)

- Member Functions:

void OnSchedulesButtonClicked()	This will act as a listener to the schedulesButton. It will be responsible for directing the user to `SchedulesActivity` when button is clicked.
void OnProfilesButtonClicked()	This will act as a listener to the profilesButton. It will be responsible for directing the user to `ProfilesActivity` when button is clicked.



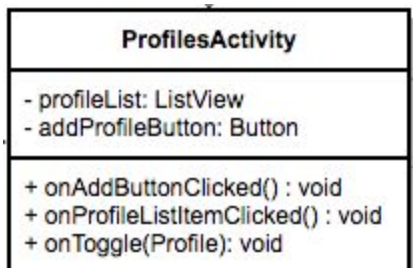
Profile: This class will model profile objects that users will make on `CreateProfileActivity` class. Each instance of the profile class will encapsulate name, list of apps to be blocked, activateNow duration time and its current state (active/passive) for that profile. This class will also be connected to the `DatabaseConnector` class as the `DatabaseConnector` class will store `Profile.java` objects in the database.

- Private Members:

name - (String)	name of the profile
blockedApps - (Array<String>)	array of names of all blocked apps in a profile
activateNowForTime - (Time)	duration for activateNow option for profile (10mins - 10hours)
Active - (bool)	saves the current state of the profile - active/passive
blockedNotifications - (Array<String>)	array of app names of whose notifications missed by the user due to the blocked apps in this profile

- Member Functions:

String getName()	getter for the name of the profile
void setName(String)	setter for the name of the profile
bool isActive()	gives current status of profile - true if active, false if inactive
void setActive(bool)	change the status of profile to the Boolean value passed
Array<String> getApps()	getter for the list of blocked apps under this profile
void addNotification(String)	method to add a missed notification to a profile object when a notification for a blocked application arrives
Array<String> getNotifications()	getter for missed notifications by the user from the set of blocked apps in this profile



ProfilesActivity: This class will be an Android activity class that will be responsible for the display of all the 'Profiles' that the user has made. The class will not modify any profiles directly but will only read `Profile.java` objects from the database for display. This activity enlists all the profiles, hence it will be it will be updated by the `BackgroundService.java` when changes occur to the database.

- Private Members:

profileList - (ListView)	a listview display layout used to list profiles. This ListView will be populated by each profile item that will follow the layout provided by ProfileListItem.xml.
addProfileButton - (Button)	a button to create a new profile
~~backButton - (Button)~~	~~a button to navigate back to the home which is the `MainActivity` (class that displays the main screen)~~

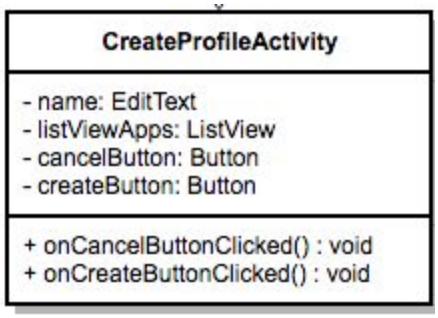
Note: This button was not needed as it's embedded in the device by default.

- Member Functions:

~~void onBackButtonClicked()~~	~~This function will act as a listener the back button. It will be responsible to direct the user to the home page i.e `MainActivity`.~~

Note: This button was not needed as it's embedded in the device by default.

void onAddButtonClicked()	This function will act as a listener to the addProfileButton. It will be responsible to direct the user to the `CreateProfileActivity` (class that defines the activity in which the user can create a new profile).
void onProfileListenItemClicked()	This function will act as a listener to the profieList(List View). It will be responsible to direct the user to `EditProfileActivity` (class that defines the activity in which the user can edit a profile).
void onToggle(Profile)	This function will act as a listener to each toggle button that is next to a profile. Once the state of the toggle button is changed, the profile will become active if button is on or inactive if button is off. If the toggle is turned from off to on, the user will get a pop up asking to select a duration of time from 10 min to 10 hours for the profile to be active. This pop up will follow timerPop.xml layout.



CreateProfileActivity: This activity class will create and manage the `Profile` class and connect the user inputs to the backend. Each successful interaction with this activity will lead to creating a new Profile (`Profile.java` object), addition of the profile on the `ProfileActivity` and on the database using the `DatabaseConnector.java`.

- Private Members:

name - (EditText)	a textbox for the name of the profile to be created
~~blockAppsButton - (Button)~~	~~a button that will display a list of all apps installed on device that the user can select to block~~

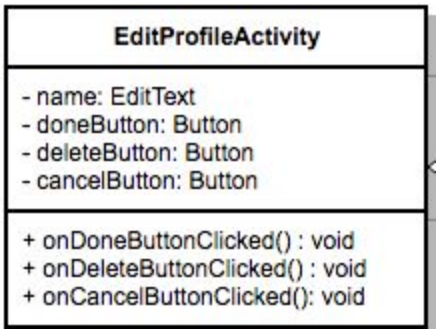
Note: This is now handled by ListView of apps on device.

listViewApps - (ListView)	a list view of all apps on device
cancelButton - (Button)	a button to dismiss the `CreateProfileActivity` which will navigate the user back to the `ProfilesActivity`.
createButton - (Button)	a button to finish and submit the Profile details

- Member Functions:

void onCancelButtonClicked()	This function will act as a listener to the cancelButton. It will be responsible to cancel the current activity without saving any user input and direct the user back to the `ProfilesActivity`.
void onCreateButtonClicked()	This function would act as a listener to the createButton. It will be responsible to make a new object of Profile class and make a function call to the `DatabaseConnector` to add this profile to the database. However, if user already has 20 profiles, an alert message will tell the user to delete a profile before adding this new profile.
~~void onBlockAppsButtonClicked()~~	~~This function will act as a listener to the blockAppsButton. It will be responsible to display a list of all apps in a dropdown list with checkboxes next to each app for user to select the apps to be blocked. This list will follow the dropdown.xml layout.~~

Note: The blocked apps button was changed to listViewApps to better User Experience by showing on the same page



EditProfileActivity: This activity class will manage and edit a pre-existent profile object and will be able to take user inputs to alter the profile. Each successful interaction with this activity will lead to a modified `Profile` object, modification of the profile on the `ProfileActivity` and on the database using the `DatabaseConnector.java`.

- Private Members:

name - (EditText)	a textbox with the name of the profile. Default value will be the existing name but can be modified to desired name if needed
~~blockAppsButton - (Button)~~	~~a button that will display a list of all apps installed on device that the user can select to block~~

Note: blocked apps button has been replaced by list view to enhance user experience

listViewApps - ListView	a list view of all apps on device
doneButton - (Button)	a button to submit the changes made to the profile
deleteButton - (Button)	a button to delete the profile currently being edited
cancelButton - (Button)	a button to cancel editing of the profile

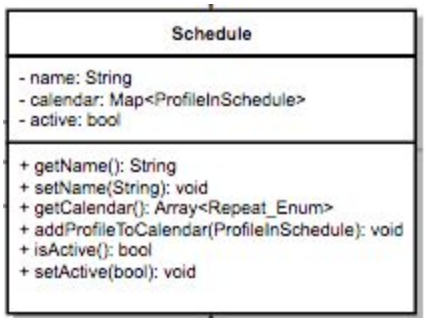
- Member Functions:

void onDoneButtonClicked()	This function will act as a listener to the doneButton,it will be responsible to submit the changes made to an existing profile.
~~void onBlockAppsButtonClicked()~~	~~This function will act as a listener to the blockAppsButton. It will be responsible to display a list of all apps in a dropdown list with checkboxes next to each app for user to select the apps to be blocked. This list will follow the dropdown.xml layout.~~

Note: blocked apps button has been removed and replaced by list view to enhance user experience

void onDeleteButtonClicked()	This function will act as a listener to the deleteButton. It will be responsible to delete the profile currently being edited i.e. object of `Profile` class and make a function call to the `DatabaseConnector` to delete this profile from the database.

void onCancelButtonClicked()	This function will act as a listener to the cancelButton. It will be responsible to cancel the editing of the profile and take the user back the `ProfilesActivity`.



Schedule: This class will model schedule objects that users will make on `AddScheduleActivity` class. Each instance of the schedule class will encapsulate name, map of profiles in the schedule, a repeatWeekly yes/no and its current state (active/passive) for that profile. This class will also be connected to the `DatabaseConnector` class as `DatabaseConnector` will store `Schedule.java` objects in the database.

- Private Members: **[Update]**

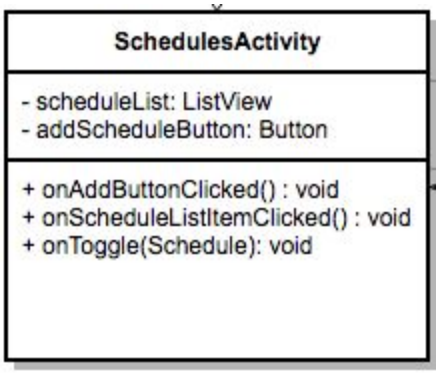
name - (String)	name of the schedule
calender - (Map<ProfileInSchedule>)	map of `ProfilesInSchedule.java` objects i.e list of profiles inside a schedule
~~repeatWeekly - (bool)~~	~~saves the user input to repeat schedule or not~~

Note: After speaking to the customer, we realized that when the state of a schedule is active, it by default repeats weekly, hence we longer need an additional check as to whether the user wants the schedule to repeat weekly or not.

active - (bool)	saves the current status of the schedule - true if active, false if inactive

- Member Functions:

String getName()	getter for name of schedule
void setName(String)	setter for name of schedule
void addProfileToCalender(ProfileInSchedule)	add a profile (`ProfileInSchedule.java` object) selected by the user to a schedule
bool isActive()	gives current status of schedule - true if active, false if inactive
void setActive(bool)	change the status of schedule - from active to inactive or vice-versa



SchedulesActivity: This activity class will be responsible for the display of 'Schedule'. The class will not modify any schedule directly but will only read schedule objects for display. This activity enlists all the schedules, hence it will be it will be updated by the `BackgroundService.java` when changes occur to the database.

- Private Members:

scheduleList - (ListView)	a listview display layout used to list schedules. This ListView will be populated by each schedule item that will follow the layout provided by scheduleListItem.xml.
addScheduleButton - (Button)	a button to create a new schedule
~~backButton - (Button)~~	~~a button to navigate to the home page~~

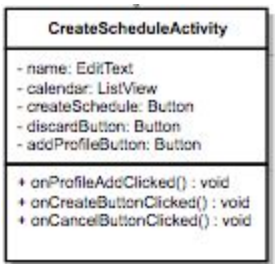
Note: This button was not needed as it's embedded in the device by default.

- Member Functions:

~~void onBackButtonClicked()~~	~~This function will act as a listener to the backButton. It will be responsible to navigate the user to the home page i.e.~~ `MainActivity` ~~(class that displays the main screen) once pressed.~~

Note: This functions was not needed as it's handled by the device by default.

void onAddButtonClicked()	This function will act as a listener function to the addScheduleButton. It will be responsible to direct the user to the `AddScheduleActivity` (class that defines the activity in which the user can add a schedule).
void onScheduleListenItemClicked()	This function will act as a listener to the profileList(ListView). It will be responsible to direct the user user to the `EditScheduleActivity` page (class that defines the activity in which the user can edit a schedule).
void onToggle(Schedule)	This function will act as a listener to each toggle button that is next to a schedule. Once the state of the toggle button is changed, the schedule will become active if button is on or inactive if button is off.



AddScheduleActivity: This activity class will create a schedule object and will be able to take user inputs to initiate a schedule object. Each successful interaction with the activity will lead to a new Schedule (`Schedule.java` object), addition of the schedule on the `SchedulesActivity` and on the database using the `DatabaseConnector.java`.

Note: This activity is now called CreateSchedule as we had to differentiate between creating a new Schedule and editing a schedule. Just a change of name but functionalities are maintained.

- Private Members:

name - (EditText)	a textbox that will get the name of the schedule through user input
~~repeatWeekly - (bool)~~	~~saves the user input to repeat schedule or not~~

Note: After speaking to the customer, we realized that when the state of a schedule is active, it by default repeats weekly, hence we longer need an additional check as to whether the user wants the schedule to repeat weekly or not.

calendar - (ListView)	a listview display layout to display the profiles in the schedule with the time and day they are supposed to run. This view will follow the calenderViewItem.xml mentioned in the “layouts” package.

Note: Instead of using a single ListView, one ListView was needed for each day of the week to sort and display the profiles in a schedule dynamically even after a profile was added to the schedule temporarily before clicking the doneButton.

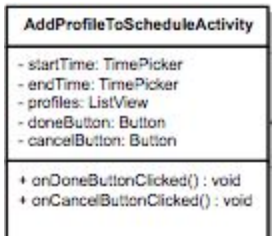
createScheduleButton - (Button)	a button used to create the schedule
cancelButton - (Button)	a button that takes the user back to `SchedulesActivity` without adding the schedule being made by the user

Note: Called DiscardButton now. Same functionalities.

addProfileButton - (Button)	a button used to add a new profile to the schedule to run as a part of the schedule.

- Member Functions :

void onProfileAddClicked()	This function would act as a listener to handle the addProfileButton. It will be responsible to direct the user to `AddProfiletoScheduleActivity` (class that defines the activity which allows the user to add a new profile to the schedule).
void onCreateButtonClicked()	This function would act as a listener to the createButton. It will be responsible to make a new object of Schedule class and make a function call to the `DatabaseConnector` to add this schedule to the database. However, if user already has 20 schedules, an alert message will tell the user to delete a schedule before adding this new schedule.
void onCancelButtonClicked()	This function would act as a listener to a the cancelButton mentioned above. It will be responsible to navigate the user to the `SchedulesActivity` once pressed.



AddProfileToScheduleActivity: This activity class will help the user to add a profile inside a schedule so that the profile can be activated automatically as scheduled.

- Private Members:

startTime - (TimePicker)	time picker to choose the start time for a profile to activate
endTime - (TimePicker)	time picker to choose the end time for a profile to deactivate
~~days - (Spinner)~~	~~multi choice spinner that follows dropdown.xml to give user choice of days of when the profile should be repeated. This spinner will take the list of days from `Repeat_Enum.java`.~~

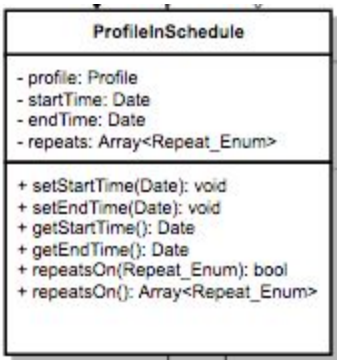
Note: Changed from a Spinner to a series of switches corresponding to each day as we wanted the user to be able to add the profiles to multiple days in the schedule and not just one day. This was done to add functionality.

profiles - (ListView)	a listview to display all the profiles made by the user for him/her to choose which need to be added to the schedule. This listview will follow showProfilesListItem.xml layout as it gives checkboxes for user to choose multiple profiles.
doneButton - (Button)	a button to submit the user input and add profile(s)

cancelButton - (Button)	a button to cancel the user input and go back to the `AddScheduleActivity` (class that defines the activity which allows the user to add a new schedule).

- Member Functions:

void onDoneButtonClicked()	This function would act as a listener to the doneButton. It will be responsible to make a new object of ProfileInSchedule class and make a function call to the `DatabaseConnector` to add this profile to the schedule in the database. The changes will also be updated on `SchedulesActivity` which displays all schedules.
void onCancelButtonClicked()	This function would act as a listener to a the cancelButton mentioned above. It will be responsible to navigate the user to the `AddScheduleActivity` once pressed.



ProfileInSchedule: This class will model profile objects inside schedules that users add on `AddProfileToScheduleActivity` class. Each instance of the ProfileInSchedule class will encapsulate a profile object, start and end time for a profile to activate/deactive, and a list of days when profile will be repeats. This class will also be connected to the `DatabaseConnector` class as `DatabaseConnector` will store `ProfileInSchedule.java` objects in schedule objects in the database.

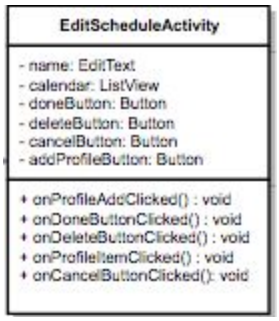
- Private Members:

profile - (Profile)	a profile object that is added to a schedule if the user decides to add it
startTime - (Time)	the start time for profile to activate
endTime - (Time)	the end time for profile to deactivate
repeats - (Array<Repeat_Enum>)	This array will hold the list of days which were chosen by the user for the profile to repeat. These days will be picked from `Repeat_Enum.java`.

- Member Functions:

void setStartTime(Time)	setter for the start time of the profile to activate

void setEndTime(Time)	setter the end time of the profile to deactivate
Time getStartTime()	getter for the start time of the profile to activate
Time getEndTime()	getter the end time of the profile to deactivate
bool repeatsOn(Repeat_Enum)	return yes if the profile repeats on any given day from `Repeat_Enum.java`
Array<Repeat_Enum> repeatsOn()	returns the list of days chosen by user for the profile to repeat.



EditScheduleActivity: This activity class will manage and edit a pre-existent schedule object and will be able to take user inputs to alter the schedule object. Each successful interaction with the activity will lead to a modified Schedule(`Schedule.java` object), modification of the schedule on the `SchedulesActivity` and on the database using the `DatabaseConnector.java`.

- Private Members:

name - (EditText)	a textbox with the name of the schedule. Default value will be the existing name but can be modified to desired name if needed.
~~repeatWeekly - (ToggleButton)~~	~~a button to allow a schedule to run weekly or not~~

Note: After speaking to the customer, we realized that when the state of a schedule is active, it by default repeats weekly, hence we longer need an additional check as to whether the user wants the schedule to repeat weekly or not.

calender - (ListView)	a listview display layout to display the profiles in the schedule with the time and day they are supposed to run

Note: Instead of using a single ListView, one ListView was needed for each day of the week to sort and display the profiles in a schedule dynamically even after a profile was added to the schedule temporarily before clicking the doneButton.

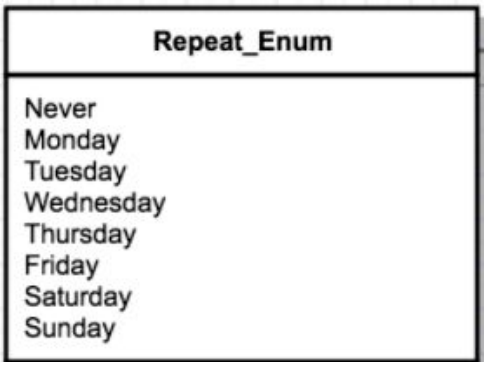
doneButton - (Button)	a button used to submit the changes made to a schedule
cancelButton - (Button)	a button that takes the user back to `SchedulesActivity` without saving the changes made by the user
deleteButton - (Button)	a button used to delete the schedule being edited

addProfileButton - (Button)	a button used to add a new profile to the schedule to run as a part of the schedule

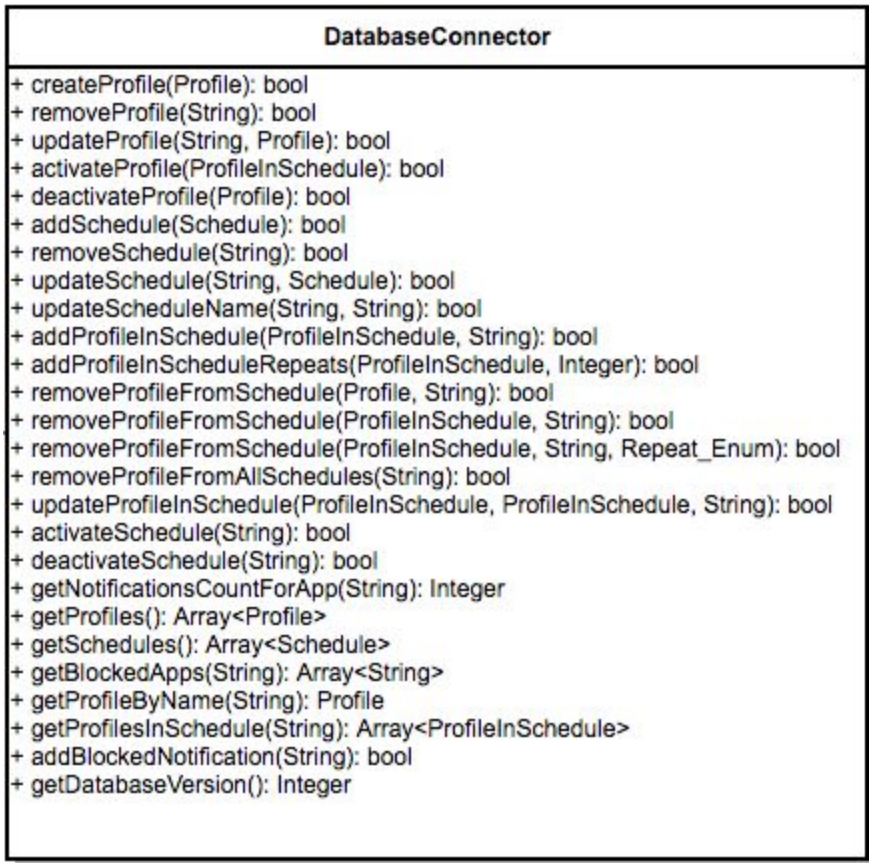
- Member Functions :

void onProfileAddClicked()	This function will act as a listener to the addProfileButton. It will be responsible to direct the user to the `AddProfiletoScheduleActivity` (class that defines the activity which allows the user to add a new profile to the schedule).
void onDoneButtonClicked()	This function will act as a listener to the doneButton. It will be responsible to modify the object of `Schedule` class and make a function call to the `DatabaseConnector` to modify this schedule in the database.
void onDeleteButtonClicked()	This function will act as a listener to the deleteButton. It will be responsible to delete the schedule currently being edited i.e. object of `Schedule` class and make a function call to the `DatabaseConnector` to delete this schedule from the database.
void onCancelButtonClicked()	This fuction will act as a listener to the cancelButton. It will be responsible to cancel the editing of the schedule and take the user back the `SchedulesActivity`.
void onProfileItemClicked()	This function will act as a listener event listener to the calendar(ListView). It will be responsible to navigate the user back to the `AddProfileToScheduleActivity` where the user can edit the profile selected in the schedule. (Note: The actual profile is not being activated in this case. Only the profile scheduled time and day can be changed)
Void onProfileItemRemoveClicked()	This functions will act as a listener for the remove button next to every profile item to remove the profile in schedule.

Note: This function was added to boost functionality for the user so that the user could delete single items rather than deleting the whole schedule or all the instances of that particular profile.



Repeat_Enum: This is an enumeration for calendar days when a profile will repeat. This class will be used in most classes in both `Client` and `Server`, which is why it is placed in package "`java`".



DatabaseConnector: This class will act as an interface between the client (UI) and the database as it will add `Profile` and `Schedule` objects made with user input into the database. It will work on the commands of the `BackgroundService.java` class which will call functions in `DatabaseConnector` to perform necessary function in database.

- Member Functions:

bool createProfile(Profile)	add a new profile object that was created by user to the database and return true on success.
bool removeProfile(String)	remove a profile object that was deleted by user from the database and return true on success.
bool updateProfile(String, Profile)	update a profile object that was modified by user in the database and return true on success.
bool activateProfile(ProfileInSchedule)	activate a profile that was selected by user that is present in the database and return true on success.
bool deactivateProfile(Profile)	deactivate a profile that was selected by user that is present in the database and return true on success.
bool addSchedule(Schedule)	add a new schedule object that was created by user to the database and return true on success.
bool removeSchedule(String)	remove a schedule object that was deleted by user from the database and return true on success.

bool updateSchedule(Schedule)	update a schedule object that was modified by user in the database and return true on success.
bool updateScheduleName(String, String)	update the name of an existing schedule and return true on success

Note: Added this function in case a user changes just the name of the schedule without any other modifications. This will help avoid passing the entire Schedule object when only the schedule name has changed, making the function call faster and cheaper.

bool addProfileInSchedule(ProfileInSchedule, String)	add a ProfileInSchedule object (profile) to the schedule user has selected and return true on success.
bool addProfileInScheduleRepeats(ProfileInSchedule, Integer)	add a ProfileInSchedule object in the Repeats Table based on its ID and return true on success

Note: Added this function to store the days a profileInSchedule repeats on. This became essential because the database was modified to store the day a profileInSchedule repeats on in a different table so it calls this function after adding the profileInSchedule with the new ID.

bool removeProfileFromSchedule(ProfileInSchedule, String)	remove a ProfileInSchedule object (profile) from the schedule user has selected and return true on success.
bool removeProfileFromSchedule(Profile, String)	remove a user selected profile from the schedule and return true on success

Note: Overloaded removeProfileFromSchedule function when user wants to delete all profile instances from a schedule. This function call is essential when a profile is deleted so all instances of that profile is deleted from the given schedule.

bool removeProfileFromSchedule(ProfileInSchedule, String, Repeat_Enum)	remove the user selected profileInSchedule from a schedule for a particular day and return true for success

Note: Overloaded removeProfileFromSchedule function when user wants to delete a profile instance from a particular day from a schedule. This function call is essential since the user deletes a profileInSchedule from a particular day and not from the entire schedule.

bool removeProfileFromAllSchedules(String)	removes profileInSchedule instances of a profile from all schedules in the database

Note: This function call is essential in the case when a user deletes a profile. Instances of that profile are deleted from all schedules in the database.

bool updateProfileInSchedule(ProfileInSchedule, ProfileInSchedule, String)	update a user selected ProfileInSchedule from the particular schedule and return true

Note: Added this function so the user can update a profileInSchedule object in a schedule. This is called by the client whenever a user edits a profileInSchedule and saves changes to update the database.

bool activateSchedule(String)	activate a schedule that was selected by user that is present in the database and return true on success.
bool deactivateSchedule(String)	deactivate a schedule that was selected by user that is present in the database and return true on success.

~~HashMap<String name,Integer number> getNotificationsForProfile(Profile)~~
~~returns a map of name of all applications and the number of notifications missed for a given profile~~
~~HashMap<String name,Integer number> getNotificationsForSchedule(Schedule)~~
~~returns a map of name of all applications and the number of notifications missed for a given schedule~~

Note: Deleted because the backgroundThread uses getNotificationsCountForApp(String) function instead to get the notifications count for a profile/schedule. This is done to move the logic for this function to the backgroundThread as it is doing the blocked notifications logic, and the backgroundThread then makes the desired function calls already implemented in the databaseConnector. DatabaseConnector now just works with the data like it is supposed to.

Integer getNotificationsCountForApp(String)	get the count of notifications the user missed for a particular app

Note: Added this so the backgroundThread can query the database and get the count of missed notifications for a particular app. This is essential as the backgroundThread calls this when an app is deactivated to retrieve the missed notifications from the blocked apps from a particular profile.

Array<Profile> getProfiles()	returns an array of all the profiles present in the database.
Array<Schedule> getSchedules()	returns an array of all the schedules in the database.

Array<String> getBlockedApps(String)	returns an array of all the apps a profile would block
Profile getProfileByName(String)	returns a Profile object for the given profile name

Note: Added this so the server can receive a Profile object by name.

Array<ProfileInSchedule> getProfilesInSchedule(String)	returns an array of all profileInSchedule in a given schedule

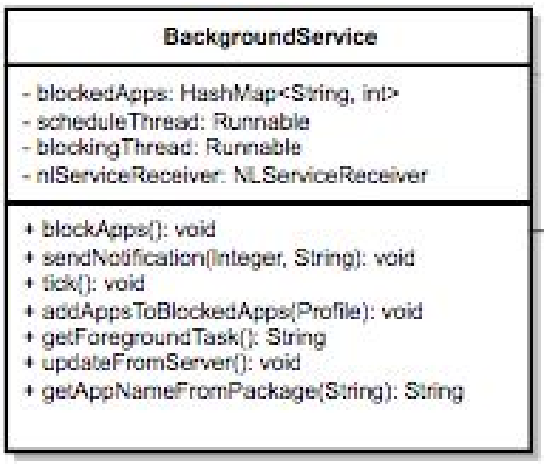
Note: Added to receive all profileInSchedules in a given schedule. This is essential for both client and backgroundThread to retrieve all profilesInSchedules from a specific schedule, especially to retrieve currently active profiles that are stored as profileInSchedules.

bool addBlockedNotification(String)	adds a blocked app notification and returns true to success

Note: Added so the backgroundThread can add a blocked notification to the database. The backgroundThread sends the notification received that the app is supposed to block and sends it out to the database to add it in the existing blocked notifications.

Integer getDatabaseVersion()	returns the current database version

Note: Added so the background thread can see if the database has been updated. This fastens the processes of the app as the background thread only looks through the database for data if the database version is changed.



BackgroundService: This will be the thread running in the background to update our data using `DatabaseConnector.java` and will instantaneously change our `User Interface` accordingly. It will keep track of the time and update the schedules and profiles accordingly both in the UI and database through `DatabaseConnector.java`. It will also be responsible for activating/deactivating the apps and blocking/releasing notifications.

- Private Members: **[UPDATE]**

blockedApps - (HashSet<String appName>)	a set to store a list of all the apps that are blocked currently on the system.

~~unreleasedNotifications - (HashMap<String name,Integer counter>)~~

~~a map to store the number of pushed notifications for particular apps for the duration of time they are blocked.~~

Note: We no longer need unreleased Notifications hashmap as the notification missed count will be stored on the database rather than being stored locally in backgroundService. This aligns more with the architecture of storing data on database while also protecting notification count from unnecessary overrides.

scheduleThread - (Runnable)	A thread that runs every 5 seconds to check for changes in database and update the blockedApps accordingly and inform UI if any profile in schedule is starting/ending or if any instant profile activation is ending.

Note: We added this as we needed another thread to run separate to the UI thread that runs in the background indefinitely to check for changes in the database.

blockingThread - (Runnable)	Added a thread that runs every second to block apps that are stored in blockedApps.

Note: We added this as we needed another thread to run separate to the UI thread that runs in the background indefinitely to block the apps. This thread is different to scheduleThread as we want to block apps to be faster when user tries to open a blocked app.

nlServiceReceiver - (NLServiceReceiver)	An instance of the inner class NLServiceReceiver used to broadcast messages between activities and read notifications posted/removed in status bar.

Note: We need this inner class to work with notification listener service in Android to read notifications that are posted or removed on the phone by any app on the device.

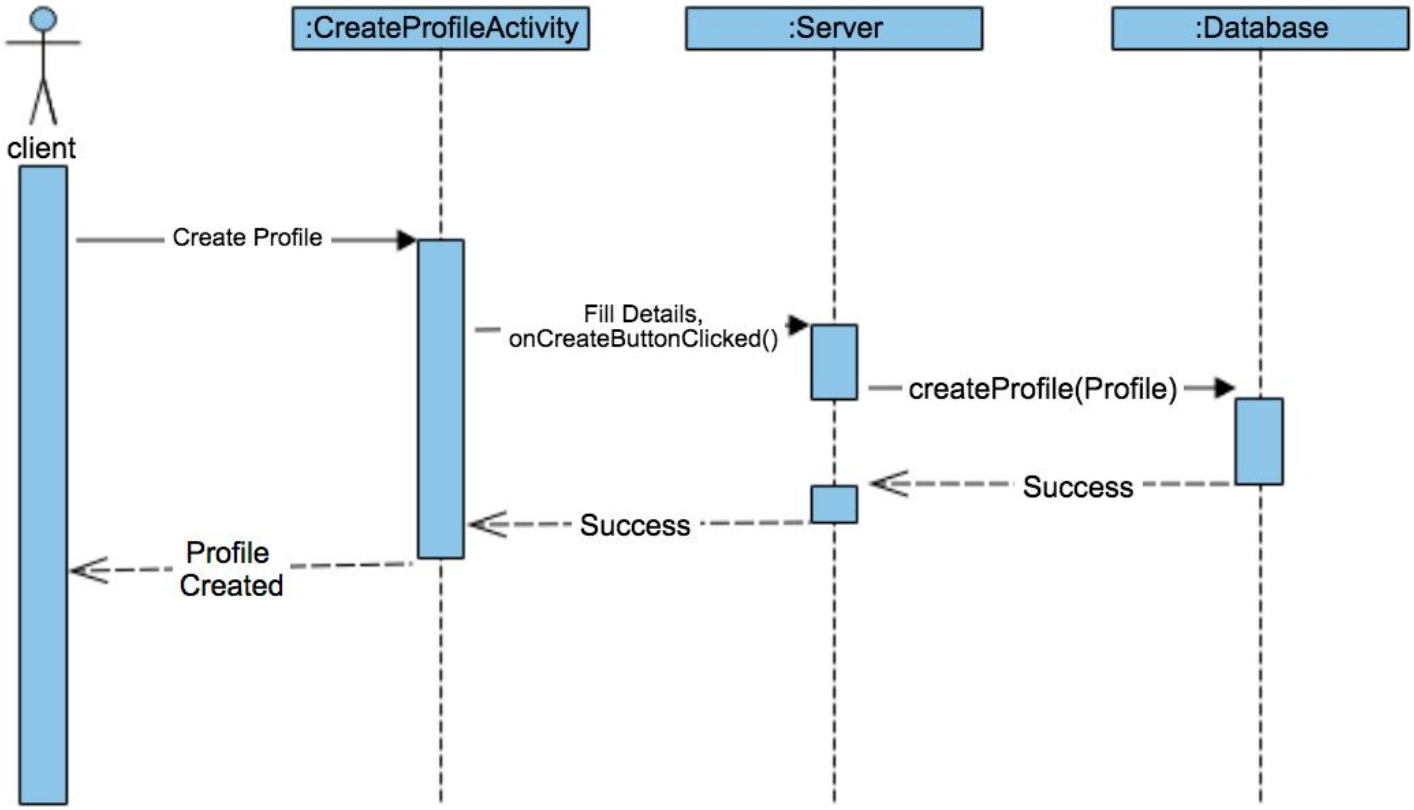
- Member Functions: **[UPDATE]**

void blockApps(~~String appName~~)	This function will go through the blockedApps hashSet and block apps according to their status.
~~void unblockApp(String appName)~~	~~Update the blockedApps(HashMap) to unblock particular apps from user. This function also sends a notification to the user to let him know that this particular application has been unblocked.~~
~~void displayMessage()~~	~~Whenever a user tries to open a blocked app, this function would display an error message to the user to indicate the status of the app.~~
~~bool updateFunction()~~	~~This function will go through the blockedApps hashMap and activate/deactivate apps according to their status. Another functionality of this function would be to send the blocked notifications to unreleasedNotifications(HashMap) and then call releaseNotifications() for all the apps that are unblocked. Returns true on success.~~

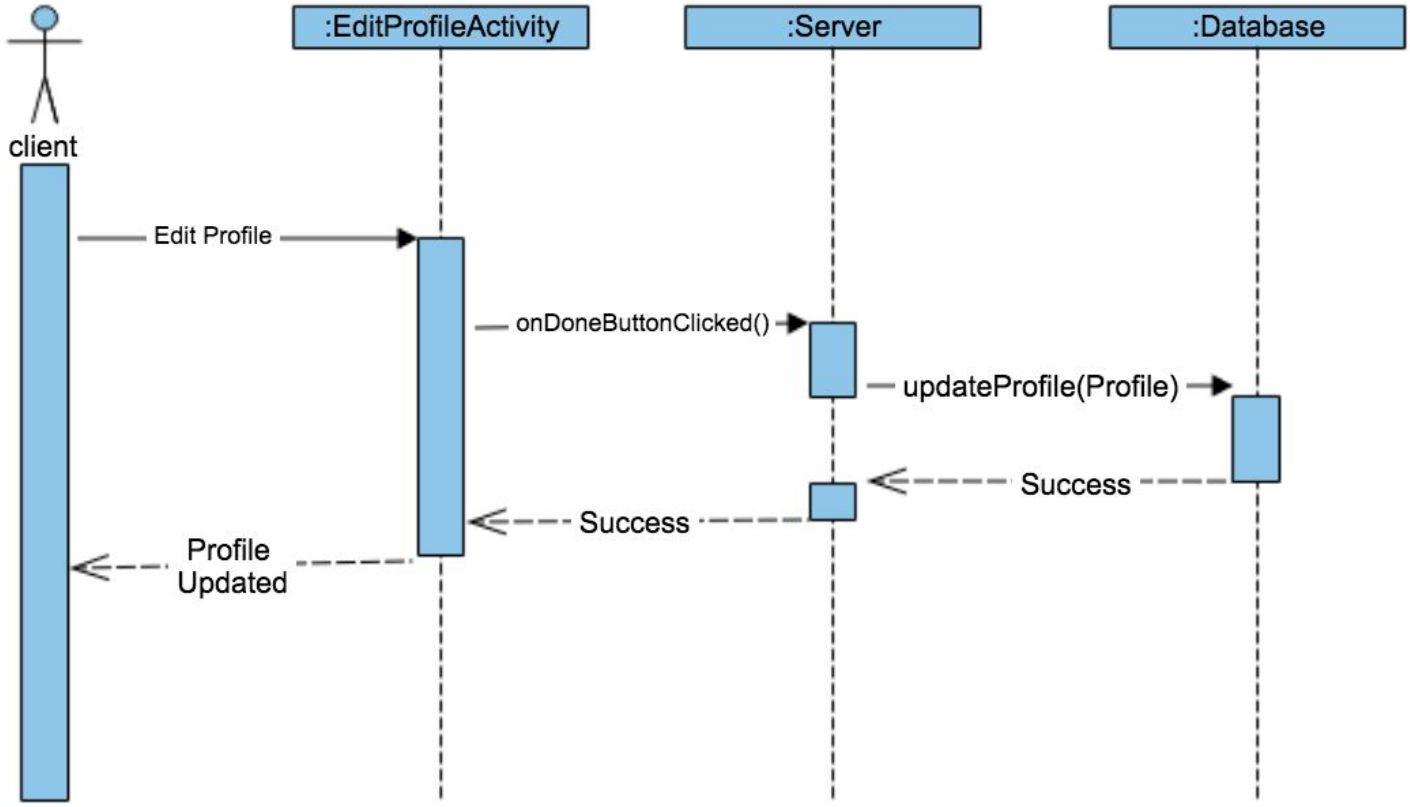
~~bool hasStarted(Profile name)~~	~~If the profile's start time is greater than the current time, this function would return true else return false.~~
~~bool hasNotEnded(Profile name)~~	~~If the profile's end time is greater than the current time, this function would return true else return false.~~
~~bool releaseNotifications(Profile p)~~	~~Whenever this function is called, it will give a popup dialogue box with the app names and the number of missed notifications from each. This pop up will follow notificationPop.xml layout. Will return true on success.~~
void sendNotification(Integer, String)	function used to send any notification to the user as a status bar notification. Integer parameter takes in the notification id and string takes in the notification message
void tick()	function which reads through all the schedules or profiles and check if their start/end time is reached on the correct day of the week when they are scheduled to run.
void addAppsToBlockedApps(Profile)	function to add a profile's blocked apps to the local hash set of blocked apps
String getForegroundTask()	returns the name of the package that is currently in the foreground in the phone
void updateFromServer()	when the database version is changed, this function pulls the schedules from the database to read through the changes
String getAppNameFromPackage(String)	gets the app name when the package name is given to the function

Note: In general, our interface for background service still remains the same with the main change being in function signatures. The addition of some new functions is only to assist the main functions that were listed out in the previous version. They have been specified here to be more detailed.

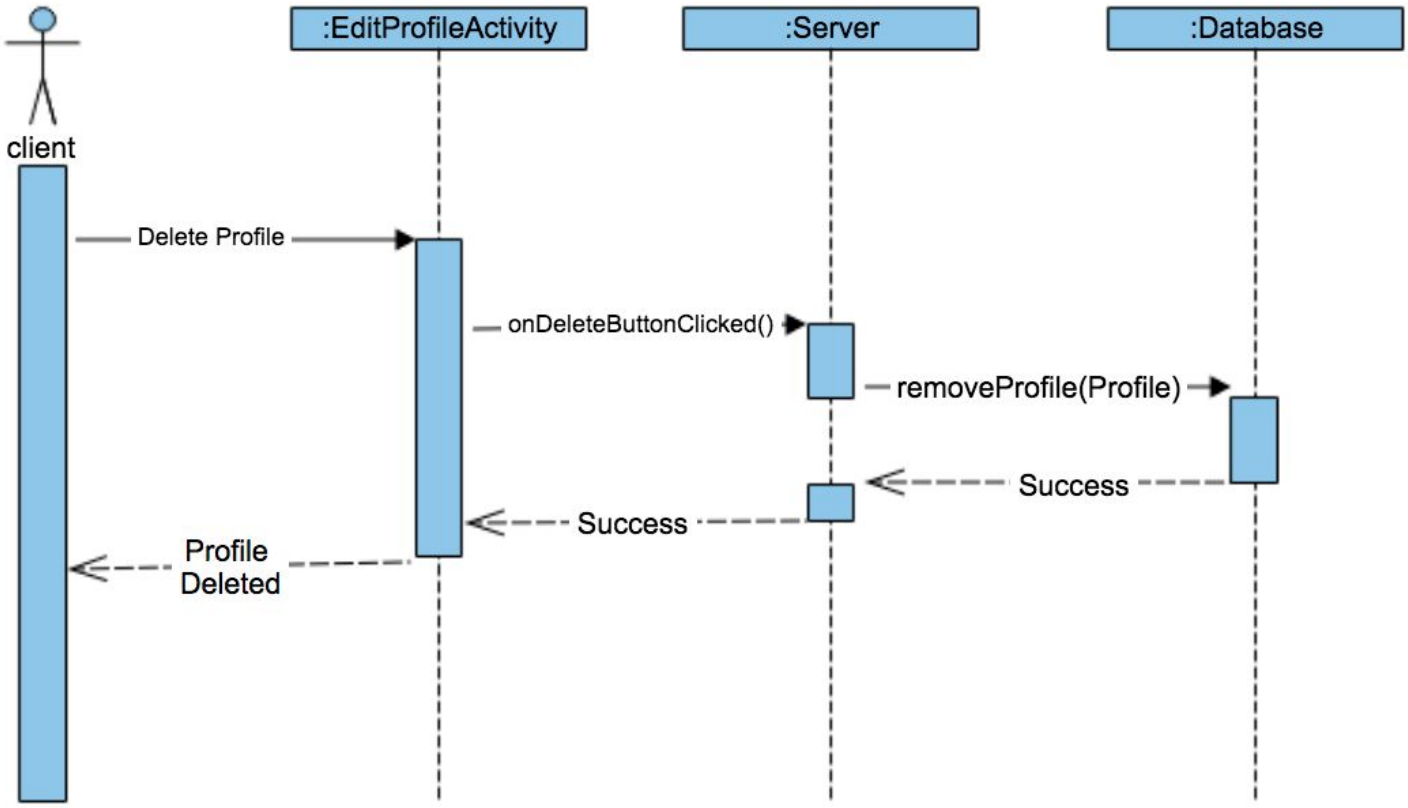
4.3. Sequence Diagrams



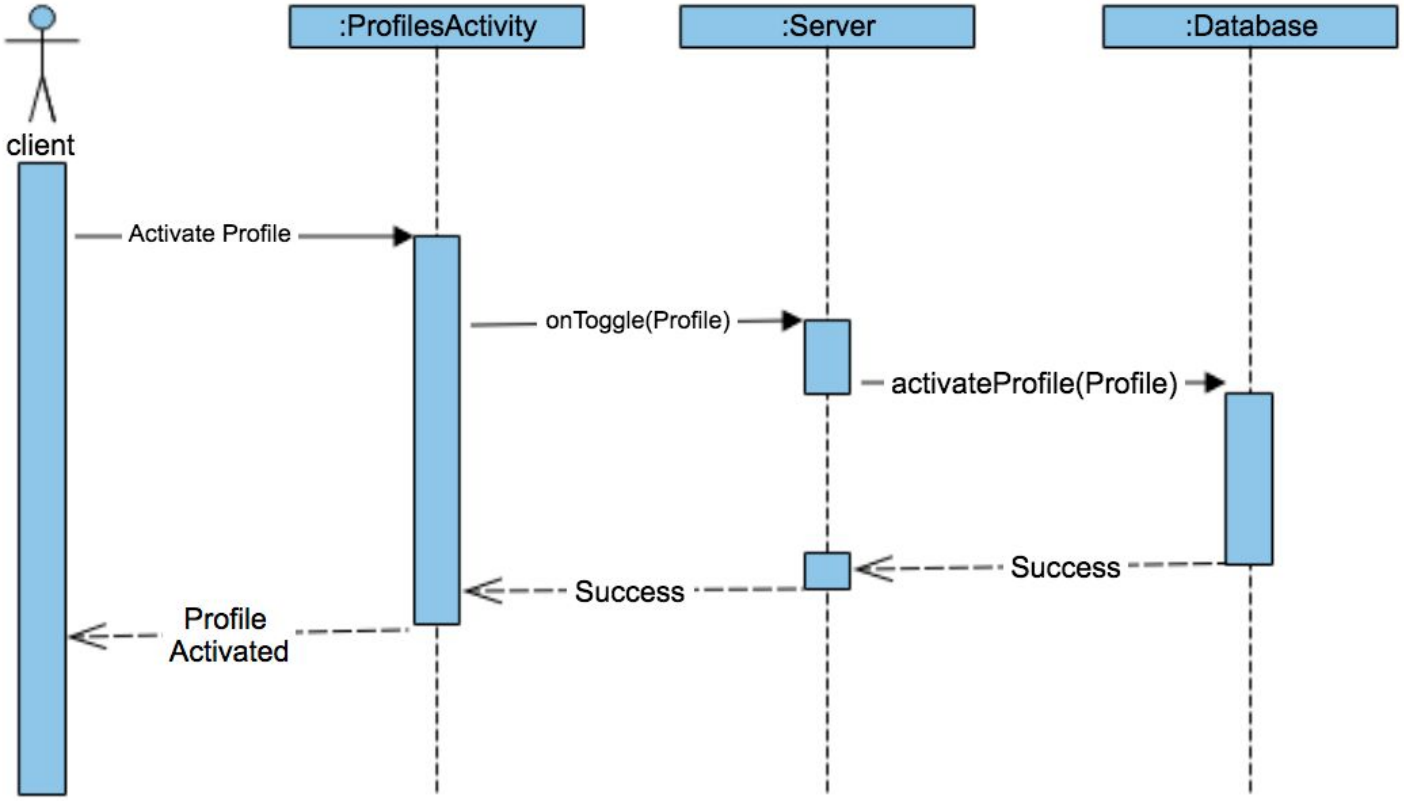
The progress of creating a new profile: the user clicks on the create new profile button and is directed to the `CreateProfileActivity`. Here, after the user fills in the required details and clicks the createButton, the information is sent to the server which makes a Profile object and adds it to the database. A successful entry would return the user a new Profile which the user can schedule or activate from the `ProfilesActivity`.



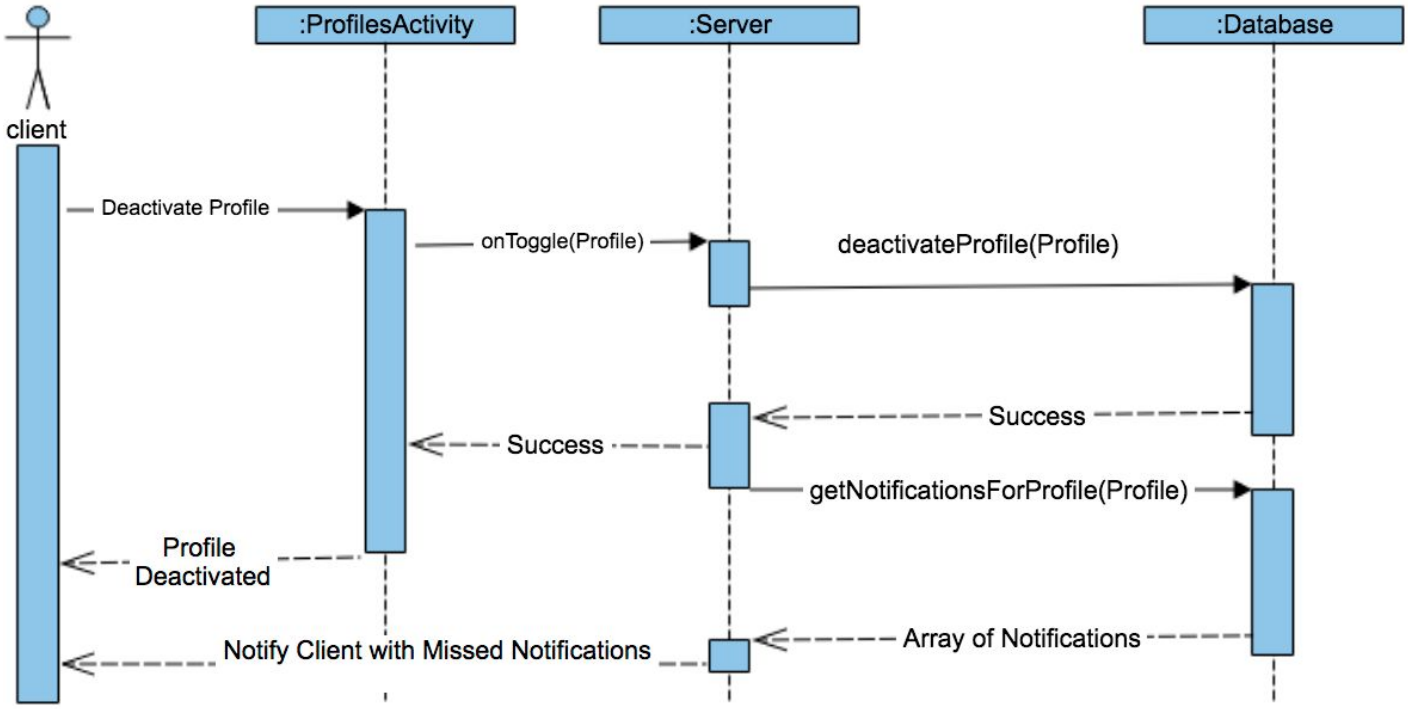
The progress of editing an existing Profile: the user clicks on the profile to be edited and is directed to the `EditProfileActivity` where all the information about the profile can be edited. The user changes the profile name and clicks the doneButton to save these changes. The information is then sent to the Server and the server updates this information in the database. The user can now see the new name for the Profile.



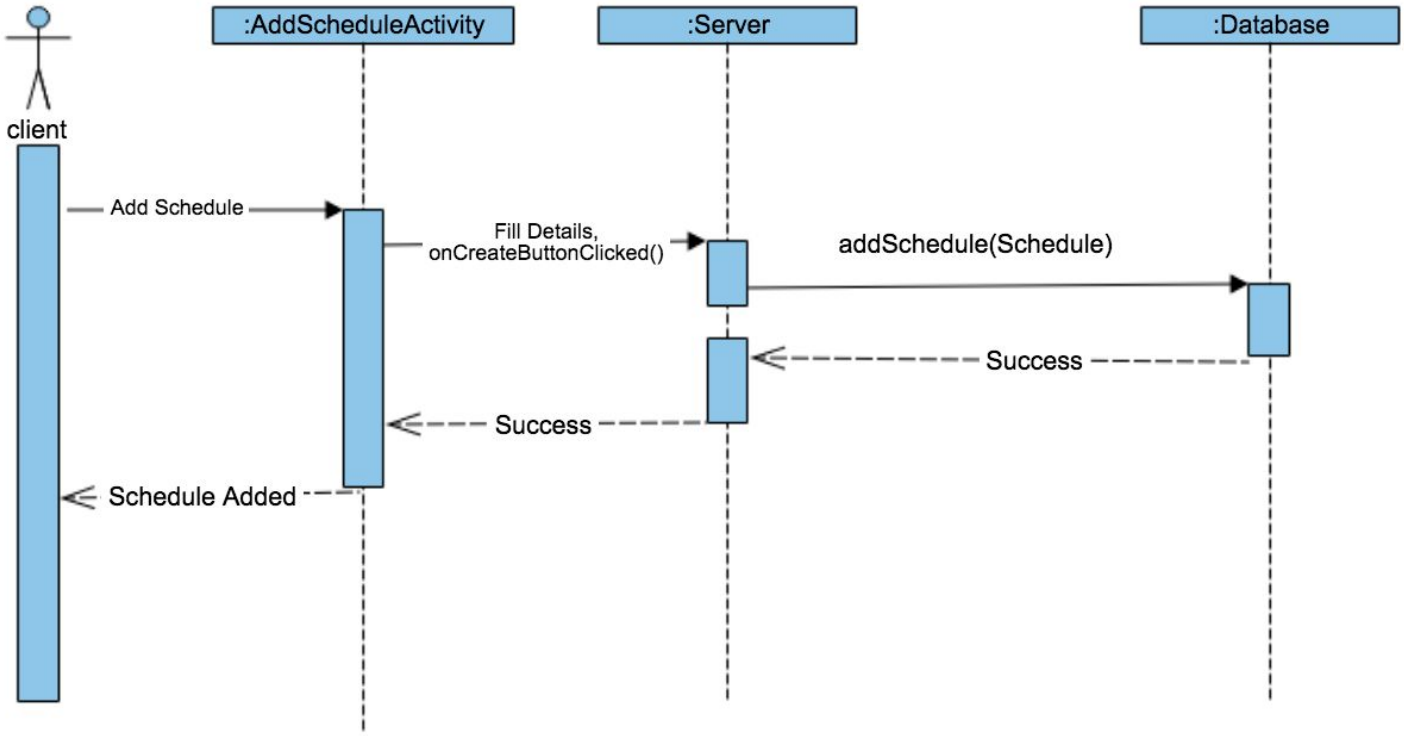
The progress of removing an existing Profile: the user clicks on the profile to be removed and is directed to the `EditProfileActivity`, where all the information about the profile can be edited. The user scrolls down and clicks the deleteButton to remove the current Profile. The information is then sent to the Server and the server updates this information in the database. The user can now see all the Profiles except the one he/she just removed.



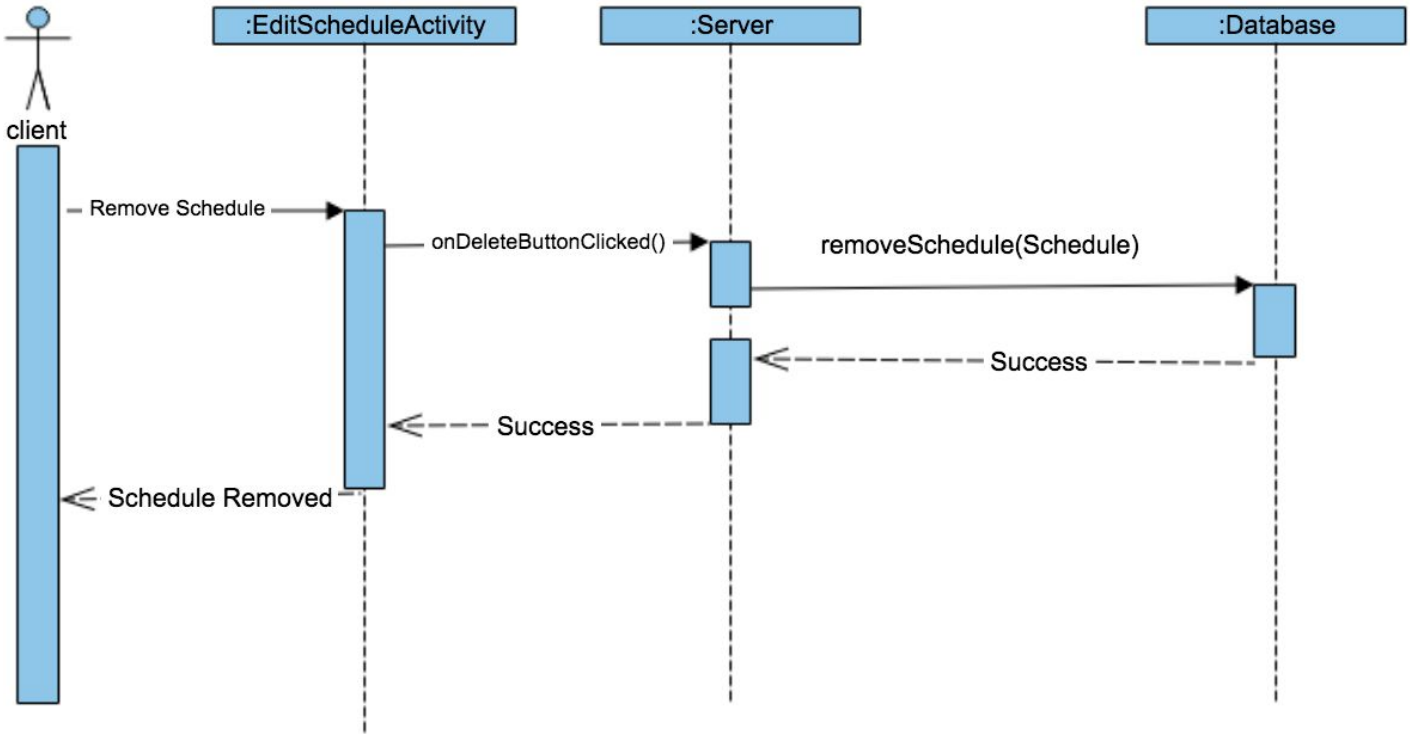
The progress of manually activating a profile for a period of time: the user goes to the `ProfilesActivity` and pushes the toggle in front of the specific profile to an active state. The server records this action and updates the list of blocked apps in the database according to this profile. This profile is now active. Now the apps from this profile will also be blocked on the phone.



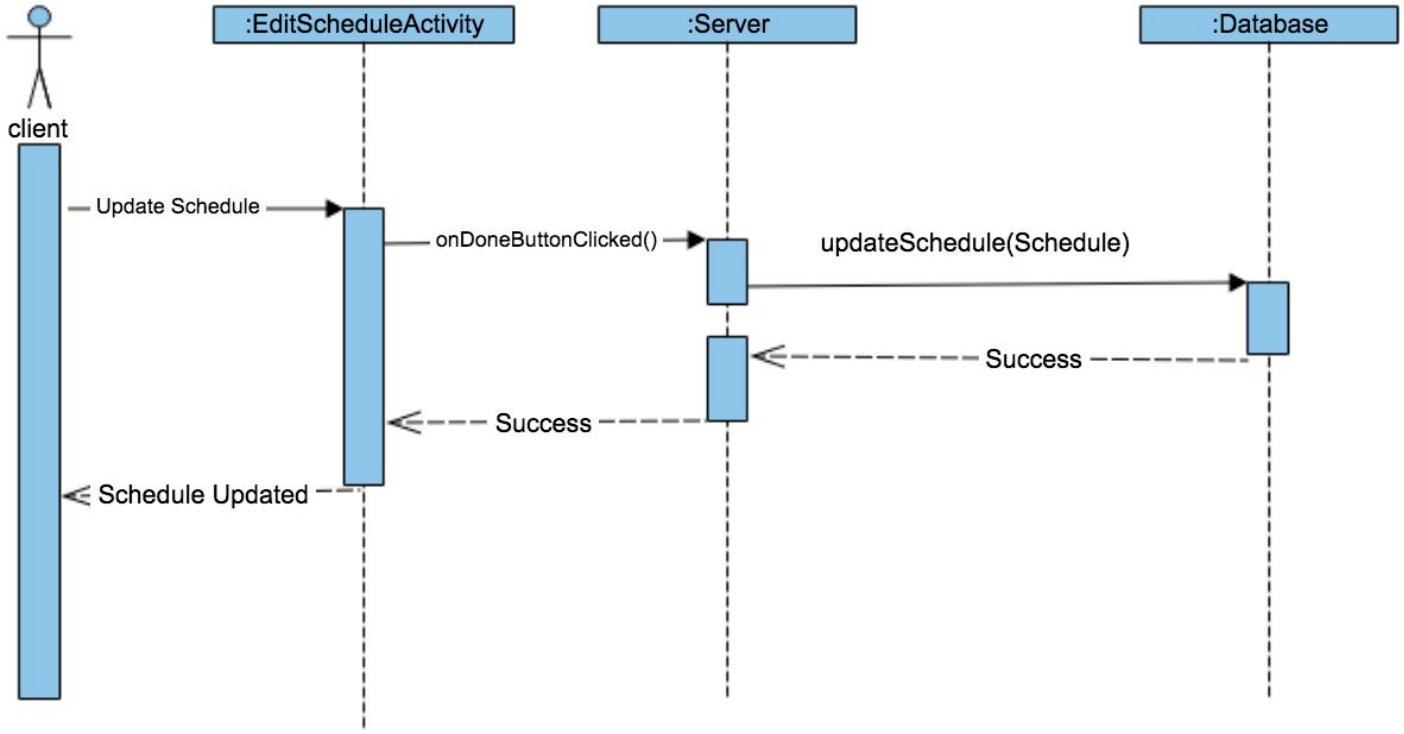
The progress of manually deactivating a profile: the user goes to the `ProfilesActivity` and pushes the toggle in front of the specific profile to an inactive state.The server records this action and updates the list of blocked apps in the database according to this profile.This profile is now inactive.Now the apps from this profile will be unblocked on the phone.The server will now push the blocked notifications to the user.



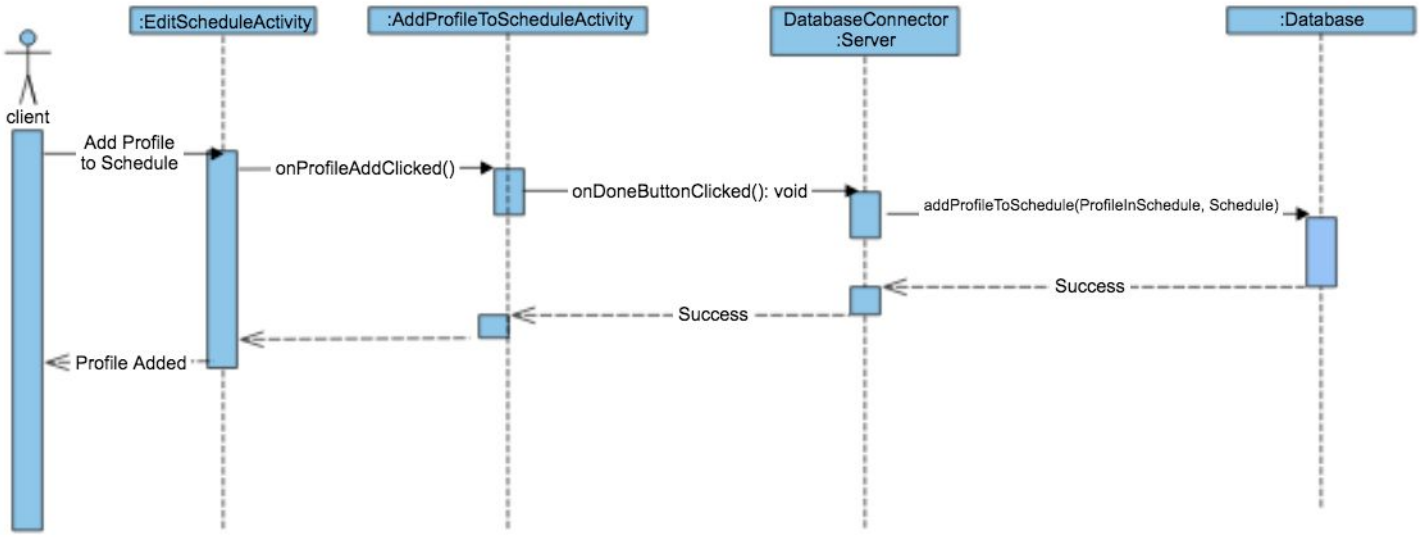
The progress of creating a new schedule: the user clicks on the create new schedule button and is directed to the `AddScheduleActivity`. Here, after the user fills in the required details and clicks the createButton, the information is sent to the server which makes a `Schedule` object and adds it to the database. A successful entry would return the user a new Schedule which the user can see and add profile to it.



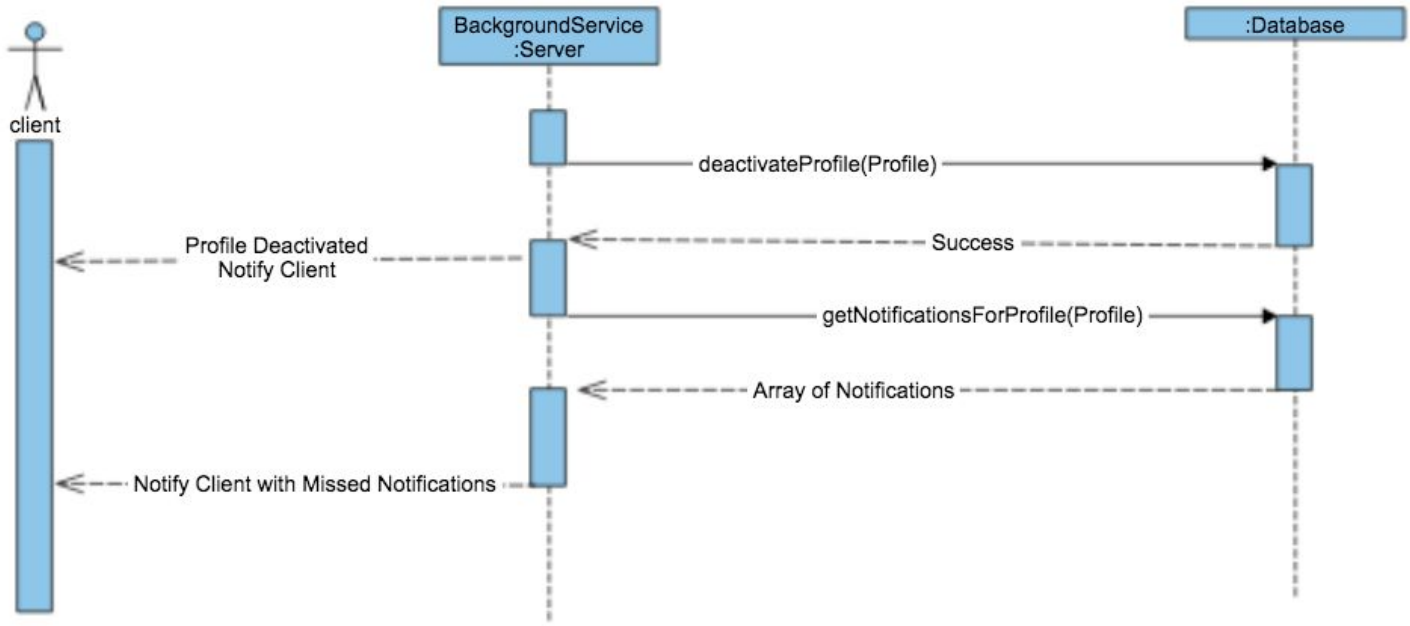
The progress of removing an existing Schedule: the user clicks on the schedule to be removed and is directed to the `EditScheduleActivity`, where all the information about the schedule can be edited. The user scrolls down and clicks the deleteButton to remove the current Schedule. The information is then sent to the Server and the server updates this information in the database. The user can now see all the Schedules except the one he just removed.



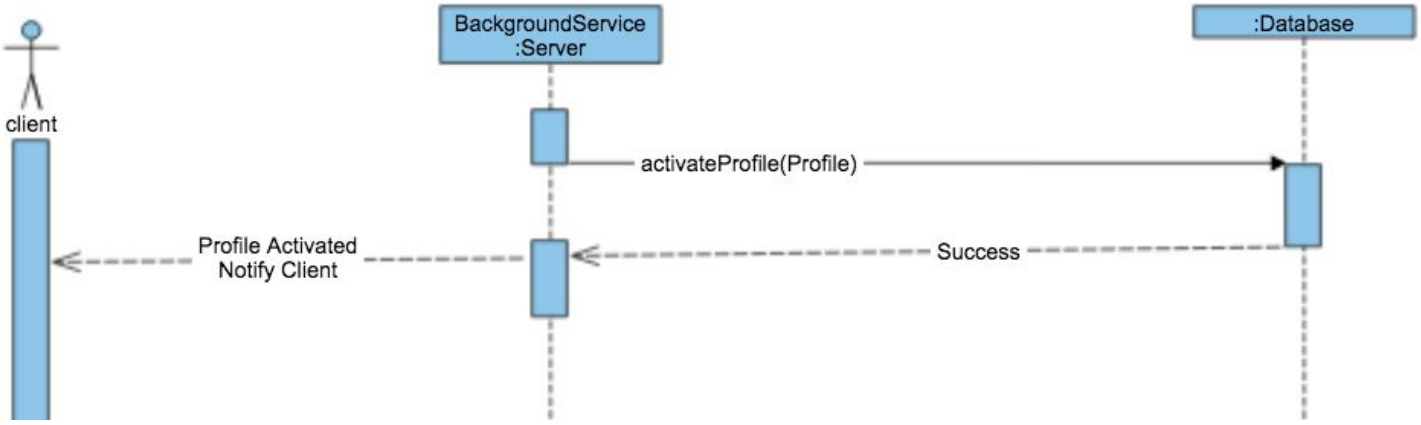
The progress of editing an existing Schedule: the user clicks on the schedule to be edited and is directed to the `EditScheduleActivity` where all the information about the schedule can be edited. The user changes the schedule name and clicks the doneButton to save these changes. The information is then sent to the Server and the server updates this information in the database. The user can now see the new name for the Schedule.



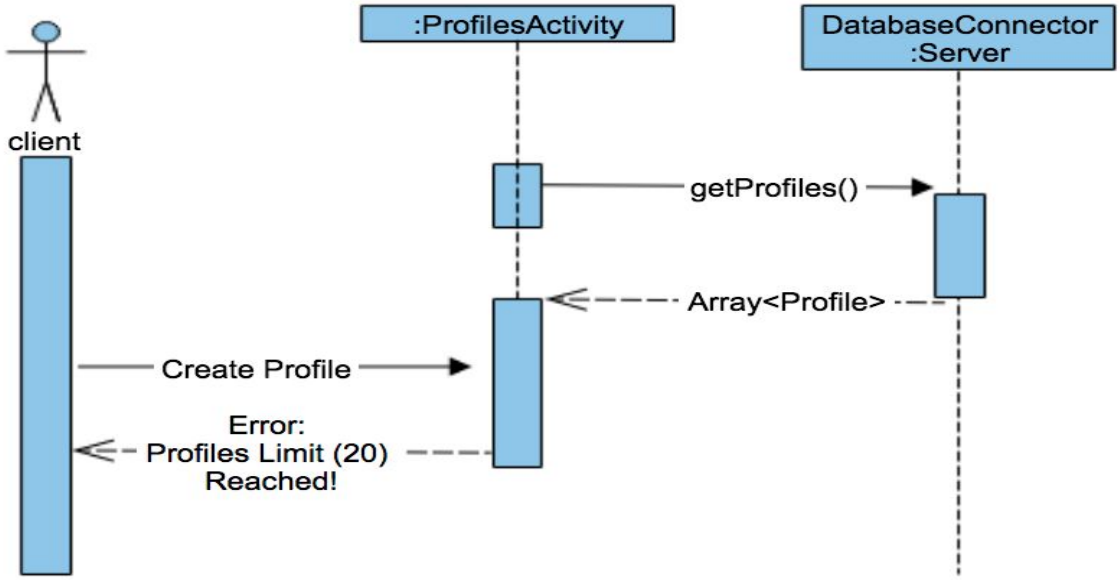
The progress of adding a profile to the user's desired schedule: the user selects the schedule where he would like to add a profile and taps the addProfile button which will lead him to `AddProfileToScheduleActivity` where the user has to select the profile to be added along with the time associated with the profile for the current Schedule. After selecting these details and submitting them, the information is then passed on to the server which adds the ProfileInSchedule Object to the Schedule object in the database. After this, the user can see the new profile in the weekly view under the particular schedule.



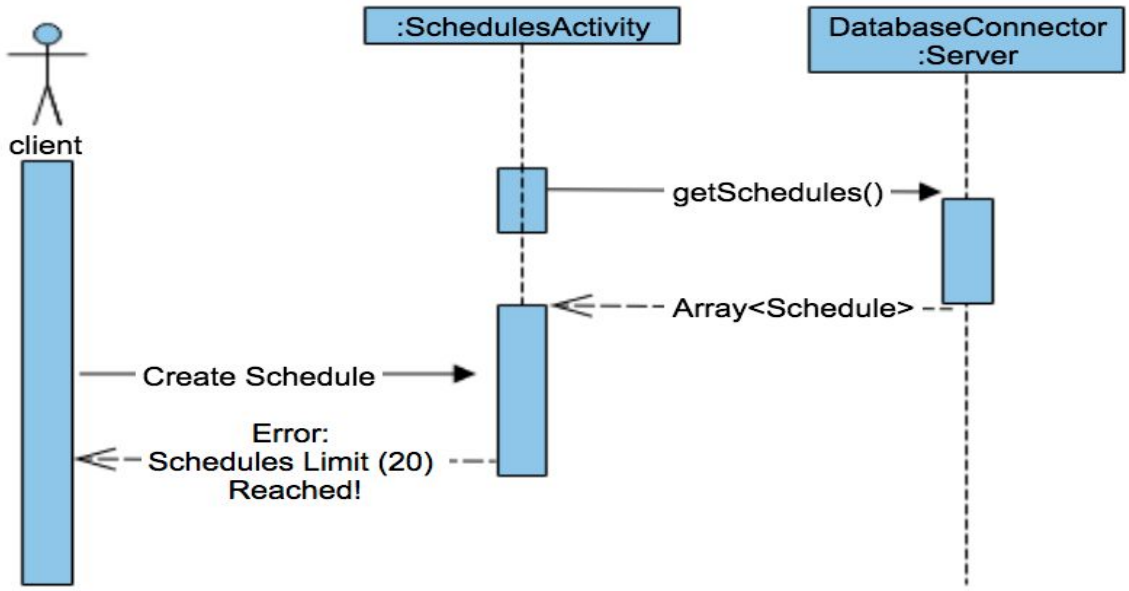
The progress of deactivating a profile according to a schedule: the server will keep track of time and as soon as a profile ends, the server would call `deactivateProfile()` and update the list of blocked app in the database and deactivate the profile. The server will also call `getNotificationsForProfile()` to retrieve the array of blocked notifications from the database.The server would now notify the user by displaying this array of notifications and also indicate that a profile has been deactivated.



The progress of activating a profile according to a schedule: the server will keep track of time and as soon as a profile starts, the server would call `activateProfile()` and update the list of blocked app in the database and activate the profile.The server would now notify the user that a profile has been activated.



The progress of creating a profile after profiles limit (20) reached: `ProfilesActivity` will load all profiles, on launch, from the server by calling `getProfiles()`. If the total number of profiles is 20 (max) and the user attempts to create a new profile, the user will be alerted that the maximum number of profiles have been created and the user will need to delete an existing profile to add a new profile.



The progress of creating a schedule after schedules limit (20) reached: `SchedulesActivity` will load all schedules, on launch, from the server by calling `getSchedules()`. If the total number of schedules is 20 (max) and the user attempts to create a new schedule, the user will be alerted that the maximum number of schedules have been created and the user will need to delete an existing schedule to add a new schedule.

5. Requirement Changes

5.1. Change 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.

This would not change our architecture design. However, details of our class designs will have to be updated to incorporate this new requirement.

All the logic regarding time checking, blocking apps and notifications happens all in the BackgroundThread component of the Application Logic component of the architecture. This background thread (running as a service in Android) runs forever in background even when the app UI is not in foreground or is not open at all as it is a different thread from the main UI thread. We will need new helper functions in the background service to calculate the intervals which *Focus!* is running, calculate total hours every profile is running, total number of weeks a schedule runs and more. We will also need to require more memory -- we will construct an extra table in the database that records statistics to implementing this feature. The helper functions in BackgroundThread can make the necessary calls to add these statistics to database. To improve efficiency of the import/export process, the database connector should also be updated to add a couple of helper functions to facilitate batch import and export.

The graphical user interface components will add UI representations for users to back up and view the statistics data. The new activity we add will require user authentication to their Google account. The activity can then display all the statistics, allow backing up the data and also allow writing exported CSV file to storage and reading from an existing CSV storage.

Background service would also likely need an inner or helper class to handle its job of handling Google account authentication, as we will want to utilize the app storage sync feature available in Android. In addition, we can use Google's authentication service to allow *Focus!* to write to Google Drive, allowing the user to transfer their existing configuration easily.

5.2. Change 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.

This again would not change our architecture design. However, details of our class designs will have to be updated to incorporate this new requirement. In addition, some of our original requirements will be invalidated, e.g., storage will no longer be local to the devices.

There is not any architectural change involved in this change because courses and instructors formation will be handled by the UI. When the user makes these courses, the UI will the DatabaseConnector component to add these courses to the database. Hence the UI to DatabaseConnector and DatabaseConnector and Database connectors remain constant as our current architecture. More so, the BackgroundService still talks to DatabaseConnector component to check for any changes in the courses timings defined by the instructor and updates the UI accordingly. Hence the components and connectors in our three tier architecture do not change.

Courses have a parallel with Schedule.java -- both specifies some profiles/classes and some times when the profiles/classes are active. We can utilize such a parallel to ensure this app works as intended. The difference is that we invert the 'apps to be blocked' to 'apps to be allowed'. The backgroundThread will just have block every app in the phone except the ones associated with the course. Hence, this involves additions of functions but not any architectural changes.

More so, new UI (activities in Android) will be added to incorporate Google authentication and subsequent instructor profile (not to be confused with Profile.java) screen that shows their authentication information and buttons that allow actions, to allow users to browse the courses they administer and see the users associated with this course, invite students while on the other hand allowing students to view, accept, or deny invitations, etc. When the user accepts a request, the particular course object will be added to the Schedules Activity as a schedule.

Finally, the database will need new tables to hold extra information of this information such that we can make the new features specified possible. A new table should hold all instructors, a new

table to hold course data, a new table to associate users with courses with relevant access privilege information, and possibly more.