Got It! Project report

Índice de contenido

Assessment quick guide:

Basic Project Requirements:

- *App supports multiple users via individual user accounts:*
  Done. The server enforces this though oatuh2 and the ClientAndUserDetailsService provided in previous assignments coupled with a customized User object, with the required modifications to suit this project.
  The client enforces this by using a log-in fragment (LogIn.java) which requires a user and password input for it to log-in into the Spring service, and a sign up page so new users can be created at will (SignUp.java).

- *App contains at least one user facing operation available only to authenticated users:*
  The application forces all users to be authenticated before performing any action other than signing up or loging in. Login is excuded by default, but I created a custom exception in the access rules so signing up is also (obviously) available to non-authorized users, as seen in GotItServer – edu.hh.auth – Oauth2SecurityConfiguration.java, lines 74 to 79.

- *App comprises at least 1 instance of each of at least two of the following 4 fundamental Android components: Activity, BroadcastReceiver, Service, ContentProvider.*
  My application uses the following: 1 Activity (MainActivity.java) plus an extra Activity only for debug purposes which may or may not work (ServiceTest.java) and is there only for your convenience, and mine. A BroadcastReceiver is registered in AlarmReceiver.java, which is tasked with handling the alarm notifications so the user is notified at some interval (30 seconds in test mode) about doing a quiz. A Service is of course also implemented. OpsService.java handles all communications with the Spring server, and due to convenience, also handles user preferences.
  The only element not used of these 4 are ContentProviders, for which I found no use in this application other than, maybe, caching.

- *App interacts with at least one remotely-hosted Java Spring-based service:*
  It does. As you can see in OpsService.java (lines 95-118), the method login() creates a Rest interface which connects with our Spring service using the GotItApi.java provided.

- *App interacts over network via HTTP/HTTPS:*
  It does. The client uses the UnsafeHttpsClient.java provided previously to handle HTTPS communications between client and server (OpsService.java, line 108). The server enforces this in Oauth2SecurityConfiguration.java, lines 275 to 296, through a Bean to customize the container preferences.

- *App allows users to navigate between 3 or more user interface screens at runtime.*
  Our users have about 8 screens which they can navigate about, all implemented through fragments which are encrusted into our MainActivity.java Activity instance. These screens are: ConfigScreen.java (which handles settings like alarm frequency and user quiz notifications to followers), FillQuizFragment.java, LogIn.java, NotificationList.java, Self/OtherProfile.java (yeah, yeah, this should be only one, I know, too late to fix), SignUp.java, and UserDataListFragment.java, which shows the list of registered users to the teen/follower can follow, unfollow or check basic data on them.

- *App uses at least one advanced capability or API from the following list (covered in the MoCCA Specialization): multimedia capture, multimedia playback, touch gestures, sensors, animation.*
  I use the multimedia capture capability so the users can set their own profile pictures using their devices built-in cameras. This can be seen in SelfProfile.java in the client, which creates an intent on an interface button click to open the camera (lines 57-67), and later recovers the result through an @Override of onActivityResult() in lines 75 to 91. It then relays that image to the OperationsManager object held in the Activity, which will through retrofit send the image to the server, so it can either create the image anew or replace the previous one, which can be seen in RestController.java, method storeImage (lines 216 to 225).
  I also use in my service (OpsService.java) a SharedPreferences object to store current alarm preferences, in methods getAlarmConfig() and setUpAlarm().

- *App supports at least one operation that is performed off the UI Thread in one or more background Threads of Thread pool:*
  It does. If you check any OpsService.java network operations, all are done through HaMER framework, using asynchronous calls to the methods of the GotItApi.

Functional Description and App Requirements:

- *The Teen is the primary user of the mobile app. A Teen is represented in the app by a unit of data containing the core set of identifying information about a diabetic adolescent, including (but not necessarily limited to) a first name, a last name, a date of birth, and a medical record number.*
  It is. All the users information is held in an UserData.java object, which among other things, stores which kind of user type is that (a teen, a follower) and personal data.

- *The Teen will receive a Reminder in the form of alarms or notifications at patient-adjustable times at least three times per day:*
  As can be seen in OpsService.java methods setUpAlarm() and getAlarmConfig(), an alarm is set to remind the teen about his quizzes once a while. Test frequency is 30 seconds, and can be otherwise configured to once every 6 or 8 hours. This is received by a broadcastReceiver, AlarmReceiver.java, which will force the app to be open on notification click. It can also be configured by the user in ConfigScreen.java, accessible through a Menu in the application.

- *Once the Teen acknowledges a Reminder, the app will open for a Check-In. A Check-In is a unit of data associated with that Teen, a date, a time, and can include the user's responses to the following set of Questions at that date and time (...):*
  Done. You can see in FillQuizFragment.java that I use a template layout to add all the questions which are sent by the server and boxes to allow the user to input data depending on the type of reply expected (float, string) to that question. He can later click to submit button so his replies will be sent to the server and processed.

- *Feedback is the mechanism by which Check-In data is summarized and provided to the user in a meaningful way. A Teen is able to monitor their Feedback data that is updated at some appropriate interval (e.g., when a Check-In is completed, daily, weekly, or when requested by Followers). The Feedback data can be viewed graphically on the mobile device.*
  I'm afraid I had no time to implement this feature. Nonetheless, the server can send back all the quizzes performed by a user, which would be used to create graphs or summaries in the client. This process would be done client-side so client-only updates can be performed to increase the ways the information can be presented or analized without shutting down servers.

- *A Follower is a different type of user (e.g., a parent, clinician, friend, etc.) who does not the ability to perform Check-Ins, but who can receive Check-In data shared from one or more Teens. A Teen can be a Follower for other Teens.*
  There are two privileges implemented in the server: PRIVILEGES_FOLLOW, and PRIVILEGES_POST_QUIZ, as seen in a utility class Utils.java which handles authority creation, and in PersistentDataManager.java, which actually stores stuff like the new users through it's createNewUser() method. Teen users (as defined by GotItApi) by default will have both of these privileges, while Follower users (again as defined by GotItApi) will only have PRIVILEGES_FOLLOW assigned to them. These checks are done in the RestController.java Spring controller. I attempted to use @PreAuthorize but some Spring woes disallow it's use in controllers, making them useless, and I couldn't make it work in a @Service annottated class either by direct instancing or by interfacing, thus I perform a check on the user authorities by hand. If he does not have the proper one, he's not doing a quiz! (example in RestController.java, method “requestNewQuiz” and “postNewQuiz”, beginning in line 230).
  Also, the client will not show the Post Quiz menu item to users which are not granted the PRIVILEGES_POST_QUIZ privilege.

- *The Teen can choose what part(s) of their data to share with one or more Followers:*
  Teen have the ability, both at sign-up and when changing their settings in ConfigScreen.java fragment, to disallow their next quizzes from triggering notifications to their followers and breaking those relationships. Nonetheless, any notification created prior to this change will still be accessible to followers. You can check for this behaviour in the server RestController.java, line 271.

- *Teen data should only be disseminated to authorized/authenticated Followers and accessed over HTTPS to enhance privacy and security of the data:* There is a database which holds which followers have access to which quizzes, through a JPA implementation and FollowElement.java objects which hold references to each teen and follower. And as only HTTPS communications are allowed between client and server, that requirement is also fulfilled (as shown in the previous basic application requirements).

Project description

Server:

The server is written using the Spring framework. It uses both oauth2 and HTTPS to guarantee secure connections between clients and server, so private information (like names, quizzes, or record numbers) is not disseminated to unauthorized users or to anyone performing packet sniffing.

In edu.hh.Application.java you can see how the container is customized so only HTTPS connections are accepted through the Tomcat server we configure, by returning a customized servletContainer (lines 110 to 135) with such configuration. In edu.hh.auth.Oauth2SecurityConfiguration.java, a new @Configuration scheme is created which among other things, prepares the ClientAndUserDetailsService to be used.

I have kept all the network operations within a single @Controller annottated class, edu.hh.RestController.java. This makes everything somewhat easier to find, but for some odd reason, disables @preAuth and @postAuth annotations from it's mapped methods. This has forced me, as shown in the privilege-dependent methods requestNewQuiz() and postNewQuiz() to actually check privileges by hand before granting the user the chance of posting or requesting quizzes to be made.

As an additional note, in RestController.postNewQuiz(), followers are only notified of new quizzes done by the people they follow if those have enabled the flag “allowFollowers”, as shown in RestController.postNewQuiz(), line 272.

Profile photos can be easily stored for any teen or follower who is registered. Everyone will be able to see them, and only the owner will be able to post or update his own photo. This is done by forcing that the 'Principal' who accesses the RestController.storeImage() method will be the user retrieved and who will have his data changed. Through edu.hh.repositories.ImageSave.java, the images are written to disk and retrieved if required by any other user who wants to fetch them to see his own, or other, user profile.

User (authentication entity) and edu.hh.datamodel.UserData (user data object) are different objects due to security concerns. I found it would be too easy to accidentally send passwords through HTTPS when sending UserData batches to anyone asking for the user list or a single profile, or when performing code modifications without enough care. Decoupling both elements allows us to forget about that issue altogether.

Quizzes are programmatically created. Due to how they are, though, it'd be trivial to include the list of questions in a database or external class along with the expected reply data type (Integer, Float, String..) to dinamically create them. Authors and quiz dates are stored in the same quiz.

Follow relationships are stored using FollowElement objects, which store both the follower and followee. JPA repositories stores these relationships (along with Notifications, Quizzes, Users

(authentication entity) and UserData objects (user non-critical details objects) allowing for persistence and easy fetching of them, using interfaces. As a matter of fact, I have tried to keep relational integrity among all tables but for the User(authentication) objects for the reasons stated above.

Finally, the server includes some default users (teen1, teen2, follower1, everyone's password is pass) for the sake of convenience. They are created in RestController.initDefaultUsers(), annotated with @PostConstruct so it doesn't trigger until everything else is ready, among it the JPA databases.

Client:

The client is composed of four main different features: a Fragment based UI, a Service to handle network and alarm configuration, the BroadcastReceiver to handle notifications to the user reminding him to log in and perform quizzes, and a manager class.

1. **User Interface:**
   The user interface is composed of a main activity (which holds references to the operations manager and manages the service binding) and 8 different fragments which allow the user to perform the different actions he's supposed to be able to. It also has a "god method" to handle operation results and view switches (processResult()).
   The ConfigScreen fragment allows the user to change both the frequency of alarms (default at 30 seconds for debug purposes), and if he allows or not followers to receive new notifications from his actions.
   LogIn fragment allows the user to log in, presenting him with the user and password text fields, and a button to do so.
   NotificationListFragment allows any user who has notifications to see them (including the date they were triggered at) and access their linked resource, a Quiz in this case.
   OtherProfile and SelfProfile are pretty much self-explanatory. These should really be condensed into a single one with action-depending triggers to hide or show certain fields like the button to upload profile picture, or the follow/not follow ones. They show some basic data of the user and it's profile picture, or the app logo if it's not set.
   QuizViewFragment allows the users who receive notifications to actually see the quiz as filled by the teens, using a dynamic layout created programmatically based on a scrollable layout. Same as QuizViewFragment, FillQuizFragment uses the same dynamic layout to allow the application to create the questions and textboxes with the correct keyboard type defined.
   Finally, UserDataListFragment shows all the users registered in the server, and clicking in any of them will open their profile view.
   All these windows are accessible through an obviously horrible menu. I'm really awful at UI design (let's ignore the obvious: If you have bothered checking my code, I also suck at it) and that requires a rework. And an artist. And a new career. Two years coding and I can't even make a decent UI. How sad! Oh, I mean, let's carry on...

2. **Service:**
   The service handles, through new threads spawned in each method and callbacks, all the network operations which this app requires in order to work. It is a bound service, which will create a RestBuilder to avoid storing the user and password anywhere, and use it whenever it requires to perform an action.
   For conveniency purposes and due to a failure in planning, I have set the alarm configuration here. The service stores the alarm config in a UserPreferences object, and can be called to change the current alarm configuration.

3. **Notifications handler:**
   AlarmReceiver.java serves as the BroadcastReceiver and notifications handler. When it receives a broadcast sent by the alarm created in the Service, it creates a notification which, on click, will request the OS to launch our application, so the user can log in and submit a new quiz.

4. **Manager:**
   OperationsManager.java is the class responsible of linking the MainActivity (and it's fragments) with the service, and perform any operations requested to the user. It is also registered as the service listener, so any reponses will be processed here before being sent to the UI to do what it has to. It also, out of conveniency, stores the current user UserData details.