Android app with token authentication

Kastriot Kadriu

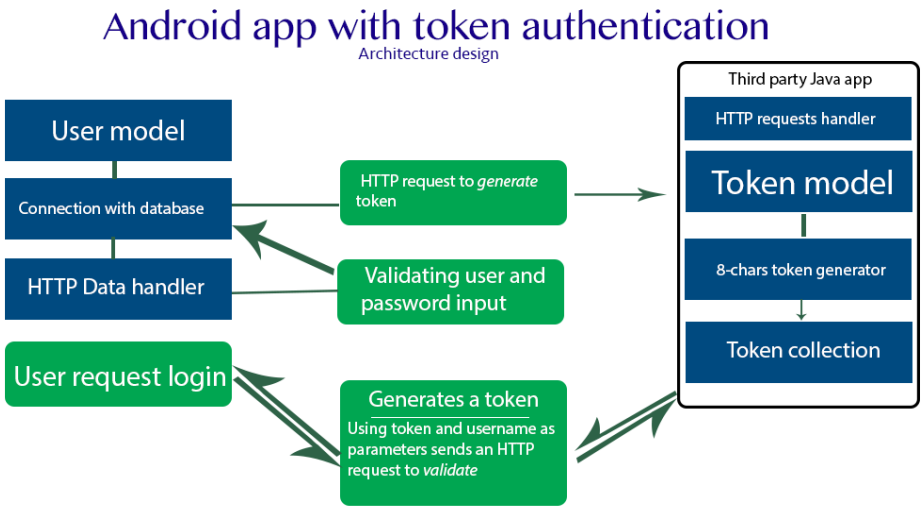
Objective:

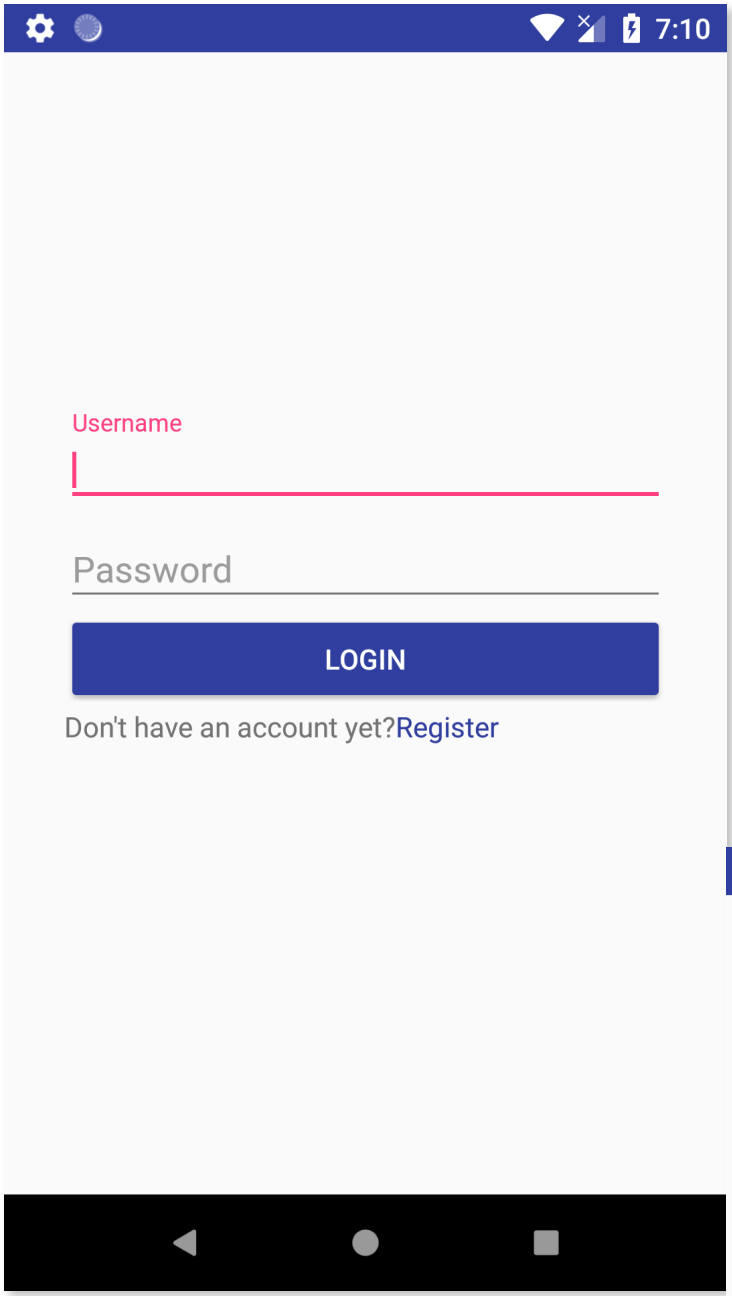
The objective of this project is to create an Android OS application that uses token authentication to handle login requests. The user can either register or login. When the user logins, a connection to check if the entered information is correct with the database is established. In the other hand, a Java application is created in order to handle the generation and validation of tokens. Android app communicates with Java app with Http requests on the http server that is established in the Java app. The emphasis stands at instances that create token model, manage token collection and connect with database. The following paper will present the architecture, user interface wireframe and the actual coding logic behind the respective application.

Designing the architecture

While designing the architecture, several things had to be kept in mind. Tokens had to be generated third party. Android app had to communicate with Java app to handle token-related requests. User and token models had to be created altogether with interfaces that implement those models. User model was referenced directly when the data was input, whereas token model was used in token collection.

Token collection contained methods for token generation and token search. Tokens were stored in local storage in the third party app. The database chosen for this project was MongoDB. While the MongoDB driver doesn't have a (proper) working implementation in Android Studio, a connection interface with a MongoDB API had to be implemented. To establish this connection, an HTTP data handler had to be created. This interface would implement GET, POST, PUT & DELETE methods.



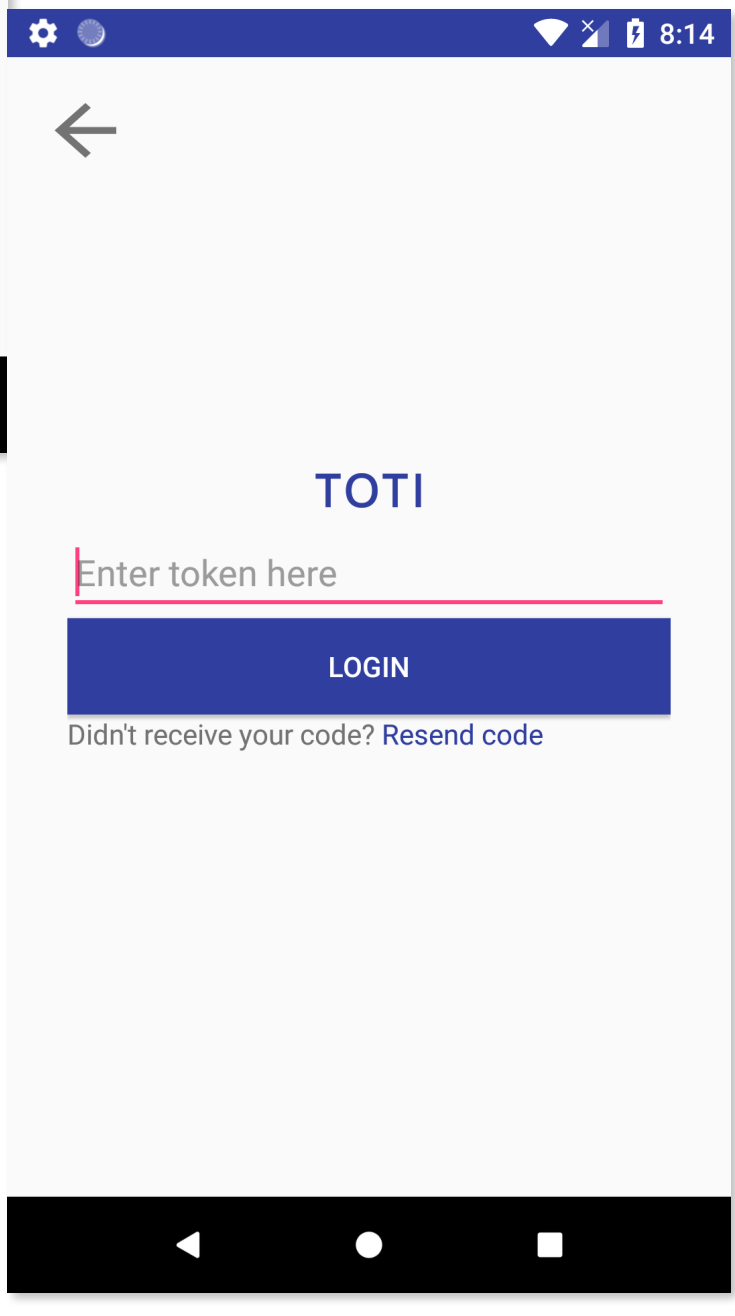


This is how the Login Screen looks. Once the Login button is clicked it validates the data entered in the Username and Password fields.

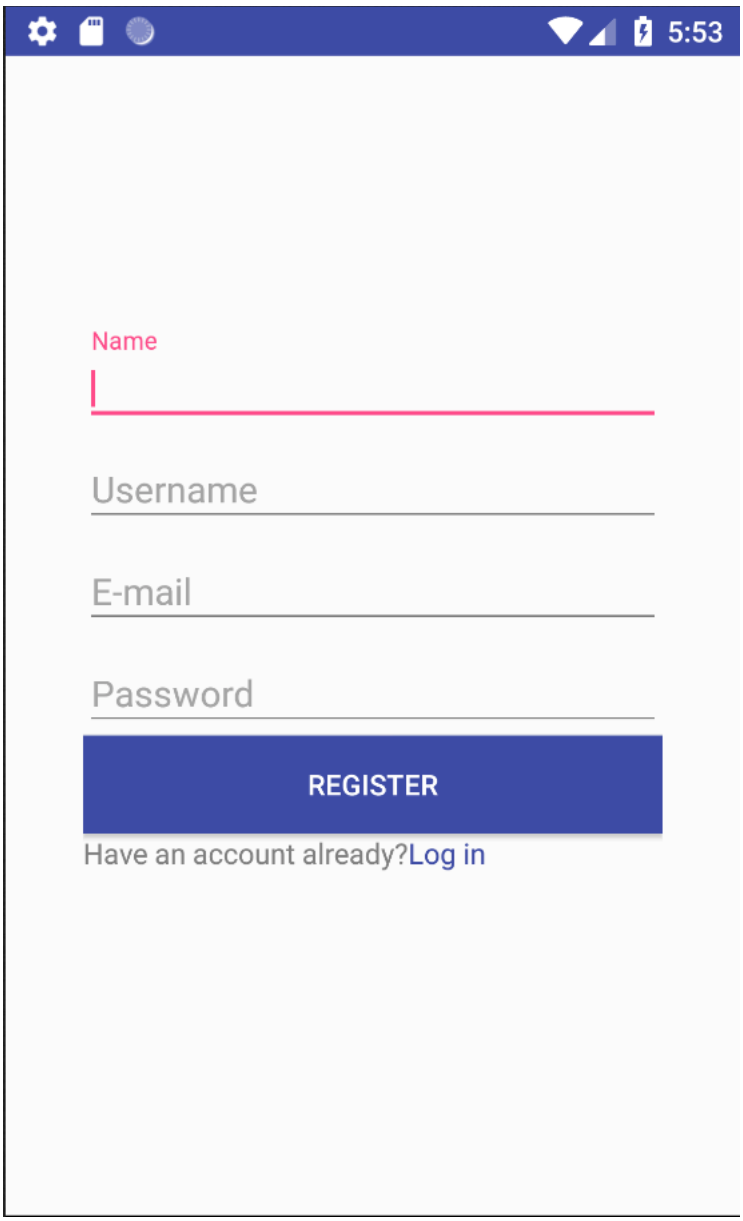
Our application contains three main user interfaces which in Android Studio world translate to Activities so three activities were created. In these three activities, methods to GET or POST data were called.

Designing the user interface

Our app contains three main user interfaces, one for login, one for registration and the other one for token input. Layouts contains EditText forms for data input, TextView for data output and button to call events.



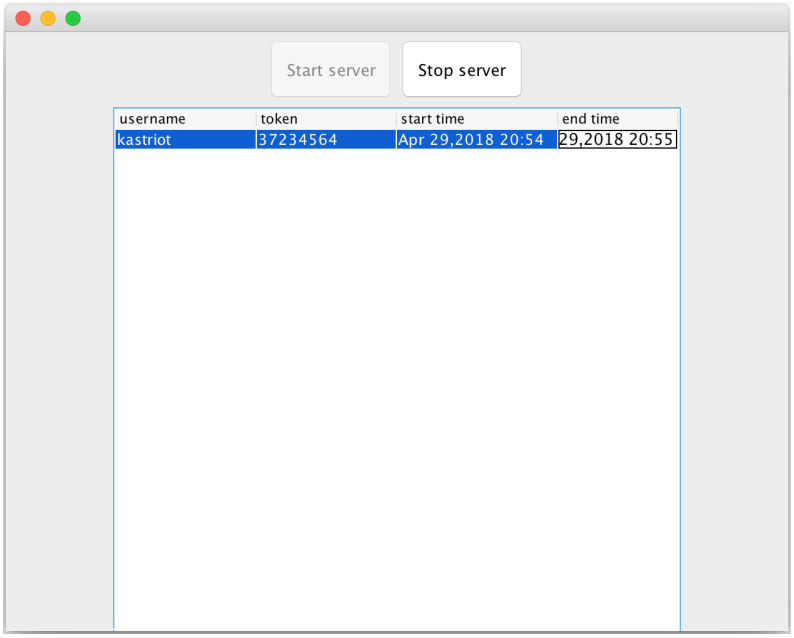
If the data entered in the Login Screen is corrected, the next screen is opened when the user is asked to enter the token to complete the log in process.



This is how the screen looks like when we want to put new data in the database.

Application logic

When the user clicks on Login button, our HTTP data handler instance establishes connection with the database to see if the record with the respective values exists. If it does, android App establishes a connection with Java app by sending a request to *generate* with username as a parameter. For security reasons we have added another parameter which can be used to validate the requests, something like an authorisation key. Java app gets the request and handles it. In Android App, another activity starts. This activity contains an input for token. When the user enters an 8 digit number, another connection with Java app is made with the request to *validate* with username, token and our authorisation key as parameters. Once again, Java app handles the requests and returns a response which then is taken by android app which grants entrance or denial to the account depending on the response.



This is how our Java app looks like. The GUI has been created using Swing.

Dealing with critical cases

The main critical case while working on this project was setting up the connection with Android App and Java app. One of the ideas was to establish the respective connection using sockets. However, socket programming within those two applications created little to no space for development and testing purposes. There were also problems with addressing on the socket server. To handle that we created an HTTP server that would accept requests, in this case, from our android app. To enhance security, we added a request validation using a parameter that would serve as our authorisation key.

One critical case is when two (or more) tokens are created, in some way or another, with short time difference between each other. This creates conflict because some tokens might be calling the timer class to delete the entry from collection, while the others might be calling a method to check if the entry exists. The conflict lies in the fact that when the collection is being modified by, in this case, the Timer class, it can't be used by other methods. To solve this issue, we use Concurrent collections, in this case, ConcurrentHashMap. Concurrent collections allow concurrent modifications from several threads without the need to block them.

To get specific data from our database, sometimes we needed to insert parameters in our API url. As specified above, the database chosen for this project is MongoDB. As we know, a MongoDB document schema is built with curly brackets "{ }". In the RFC 1738 standard it is stated that some certain characters are unsafe in URLs, those characters including the curly brackets. In our case, to build our URL to connect with DB API it was needed to encode the URL with the URLEnconder class.

In our application, we have created nested classes, in each activity, that extend AsyncTask class. AsyncTask class offers the *doInBackground*() method which, as one can guess by its name, offers to complete a task in the background, while the user is presented with the interface designed on *onPreExecute*() class, and finally, with the resulting interface in *onPostExecute().*

For full coding documentation check the projects on **github:**
https://github.com/tot98git/TokenAuth
https://github.com/tot98git/JavaThirdPartyApp