Android Capstone Project

Symptoms Management

26/10/2014
Coursera: https://class.coursera.org/androidcapstone-001

Table of contents

Requirements

First of all, these are the requirements to be met with the system. This list is taken from the project statement.

Functional Requirements

1. The *Patient* is the primary user of the mobile app.
2. The Patient will receive a *Reminder* in the form of an alarm or notification at some adjustable frequency, at least four times per day.
3. Once the Patient acknowledges a Reminder, the app will open for a *Check-In*.
4. During a Check-In, the Patient is asked, “How bad is your mouth pain/sore throat?” and can respond with “well-controlled,” “moderate,” or “severe.”
5. During a Check-In, the Patient is asked, “Did you take your pain medication?”
6. If a Patient is taking more than one type of pain medication, each medication will require a separate question during the Check-In (e.g., “Did you take your Lortab?” followed by “Did you take your OxyContin?”). The Patient can respond to these questions with “no” or “yes.”
7. During a Check-In, if a Patient indicates he or she has taken a pain medication, the Patient will be prompted to enter the time and date he or she took the specified medicine.
8. During the Check-In, a Patient is asked, “Does your pain stop you from eating/drinking?” To this, the Patient can respond, “no,” “some,” or “I can’t eat.”
9. A Patient’s *Doctor* is able to monitor a Patient’s Check-Ins with data displayed graphically.
10. A Doctor can see all his/her Patients and search for a given Patient’s Check-In data by the patient’s name.
11. A Doctor can update a list of pain medications associated with a Patient’s account. This data updates the tailored questions regarding pain medications listed above in (6).
12. A Doctor is alerted if a Patient experiences 12 hours of “severe pain,” 16 hours of “moderate pain or severe pain” or 12 hours of “I can’t eat.”
13. A Patient’s data should only be accessed by his/her Doctor(s) over HTTPS.

Technical Requirements

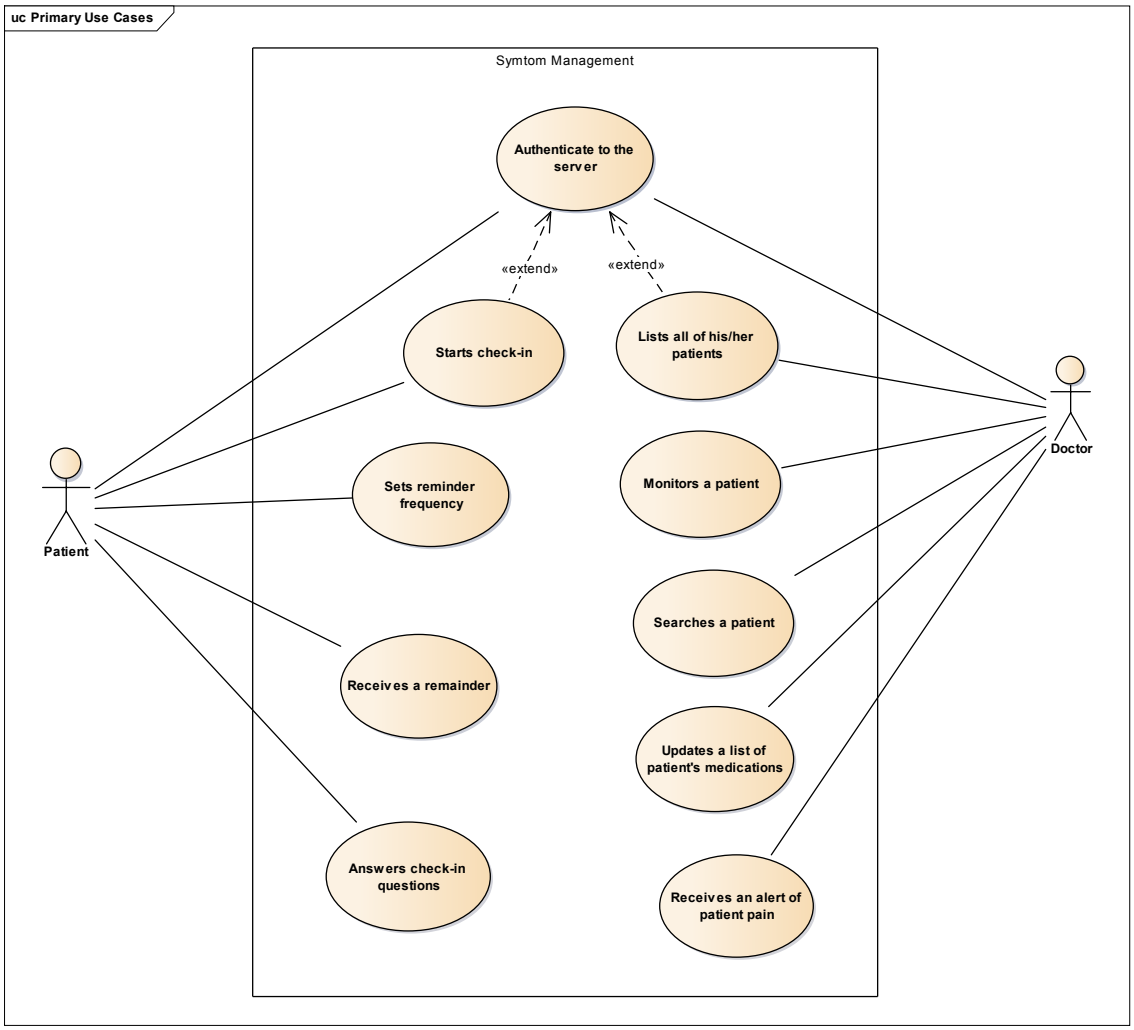
1. Apps must support multiple users via individual user accounts. At least one user facing operation must be available only to authenticated users.
2. App implementations must comprise at least one instance of at least two of the following four fundamental Android components: Activity, BroadcastReceiver, Service and ContentProvider.
3. Apps must interact with at least one remotely-hosted Java Spring-based service over the network via HTTP
4. At runtime apps must allow users to navigate between at least three different user interface screens. o e.g., a hypothetical email reader app might have multiple screens, such as (1) a ListView showing all emails, (2) a detail View showing a single email, (3) a

compose view for creating new emails, and (4) a Settings view for providing information about the user's email account.

5. Apps must use at least one advanced capability or API from the following list covered in the MoCCA Specialization: multimedia capture, multimedia playback, touch gestures, sensors, or animation. Experienced students are welcome to use other advanced capabilities not covered in the specialization, such as BlueTooth or Wifi-Direct networking, push notifications, or search. Moreover, projects that are specified by commercial organizations may require the use of additional organization-specific APIs or features not covered in the MoCCA Specialization. In these cases, relevant instructional material will be provided by the specifying organization.
6. Apps must support at least one operation that is performed off the UI Thread in one or more background Threads or a Thread pool.
7. There may also be additional project-specific requirements (e.g., required use of a particular project-specific API or service).

Use Cases

This section describes the Use Cases diagram that shows, at high level, the functionality of the system for Patients and Doctors. The Backend functionality to manage Patients or Doctors is not in the requirements of this System.



Use Cases description

For each Use Case, I am going to write a table of contents to describe in more detail the functionality of the System.

Name	Authenticate to the Server
Roles	Patient, Doctor
Description	
The user enters the login and password in a form. This information is required when the user starts the App. The user and password is sent to the server via https protocol. The server authenticates the user and sends back the OAuth token. The user will not need to enter the login and password again to communicate with the server, due to the fact that the OAuth token is used next time when is needed to communicate with the server. The user can press a check below the username and password fields to indicate that he/she wants to be remembered next time that he/she starts the app again. In this case, the OAuth token is	

managed by Android Authenticator manager for futures uses. Once the user is authenticated for the first time in their lives, a background process communicates with the server and starts to download the initial information. If the user is a Patient the server sends back a list of his/her medications and role information in order to present Patient's screen. If the user is a Doctor, the server sends back only role information in order to present Doctors' screen.
Considerations
Data for Patients and Doctors and the relationships between them have to be defined at the database server with another computer system. The mobile device has to be connected to WIFI or cellular network The patient's medication list will be saved internally in the mobile device's database

Functional requirements	Basic Project Requirements

Name	Starts check-in
Roles	Patient
Description	
Once the user is authenticated successfully, another screen will be shown, with the information of the next time reminder and below, a button to start manually a new check-in. Once the user press the button, the Use Case "Answer check-in questions" is presented. Starts check-in will be the first screen that patients will see when they start the app and they have already been authenticated in the server. If no medication has been associated with him/her, a message will be shown to alert patient that he/she can't do a check-in.	
Considerations	
The user is authenticated in the server	
Functional requirements	Basic Project Requirements

Name	Sets reminder frequency
Roles	Patient
Description	
The user will be able to change the frequency of the Reminders in the preference screen. The options will be: 4 times per day, 6 times per day or 8 times per day. In this Use case, the user will be able to change other preferences like the interval of time to connect to Server for looking for his/her medications. This parameter maybe won't be necessary if the server uses Push notifications. This functionality will be accessed from the app menu with the label "set preferences"	
Considerations	
Functional requirements	Number 2 (of the list of App Requirements for Symptom Check)

Name	Receives a reminder
Roles	Patient
Description	
At the interval period of time defined by the user, the App will send a reminder in the form of Notification. The user will be able to see this reminder in the notification area of the device. The notification will show an icon and the text message similar to "Please, do a check-in". The user can press the notification at the notification area to open a screen with the use case "Answer check-in questions".	

Considerations	
The patient has a list of medication defined by his/her doctor saved at the mobile device database.	
Functional requirements	Number 2 and 3 (of the list)

Name	Answers check-in questions
Roles	Patient
Description	
The App shows a wizard with several screens with questions about his/her pain. Every screen has only one question with different options in a form of buttons. The questions are described at the requirements numbers 4 to 8. Once the user has finished the wizard, the data that the user has answered are saved in the local device database with the state "needs to be sent" and a background service is started to send the data to the server. When the data is successfully sent it, the service updated the state of the data to "send it". While the data is not successfully send to the server, the service in background will try to send it another time. This allow user to turn off the mobile once the questions have been answered. During the wizard some questions appear about the list of medications that the patient is taken. This list of medication is updated from time to time on the server when the doctors change it and a process in background will download the new list from server and save it in the local database.	
Considerations	
The mobile device has to be connected to WIFI or cellular network to send the check-in data. Maybe to avoid pull request to download the medication list from time to time, a push notification system like GCM will be used.	
Functional requirements	Numbers: 4, 5, 6, 7, 8

Name	Lists all of his/her patients
Roles	Doctor
Description	
Doctors will be able to list all of his/her patients. This list will be ordered by name and lastname and will be the first screen that doctors will see when they start the app and they have already been authenticated into the server. From this list, pressing a patient, the doctor will access to another screen with the list of the last 10 days of check-in data filled by his/her patient. Pressing any of these check-ins, the doctor will be able to see all of the check-in data.	
Considerations	
In order to protect medical data information, the list of patients will be requested to the server and will not saved locally. Check-in information is saved in the local database but referenced by the identifier of each patient and not by his or her name. The mobile device has to be connected to WIFI or cellular network.	
Functional requirements	Number 10

Name	Searches a patient
Roles	Doctor
Description	
Doctors will be able to search a patient by his/her exact name. This functionality will be accessible from an option menu that will appear in the patients list screen. If no patient is located by his/her name an error will be shown. If patient is located, the app	

will show the screen with the list of last ten days of check-in data filled by the patient. Pressing on each row, the doctor will be able to see all check-in data.	
Considerations	
The mobile device has to be connected to WIFI or cellular network.	
Functional requirements	Number 10

Name	Monitors a patient
Roles	Doctor
Description	
Doctors will be able to see the patients' pain data in a form of graphic. This will allow doctors to see the patients' pain evolution to take measures for the treatment. This functionality will be accessible from an option menu that will appear in the patient check-ins list screen.	
Considerations	
The medical information will be queried in the mobile device's local database.	
Functional requirements	Number 9

Name	Updates a list of patient's medication
Roles	Doctor
Description	
Doctors will be able to update a list of pain medication associated with a Patient. To do this, a screen will be presented with a list of pain medication. Doctors will be able to add new medication writing the name in an input text and pressing the button Add. Doctors will be able to delete a medication from the list pressing the row medication for a long time and selecting option delete. This functionality will be accessible from an option menu that will appear in the patient check-ins list screen.	
Considerations	
All of the changes will be sent to the server and saved in the mobile device. The mobile device has to be connected to WIFI or cellular network.	
Functional requirements	Number 11

Name	Receives an alert of patient pain
Roles	Doctor
Description	
Doctors will receive a notification in the notification area of the mobile device when a Patient experiences 12 hours of “severe pain,” 16 hours of “moderate pain or severe pain” or 12 hours of “I can’t eat”. Doctors will be able to press the notification and the app will start showing the patient check-ins list screen. When the notification arrives, a sound will be played in order to alert the Doctor that patient needs a better medication.	
Considerations	
A background service in the Doctors' app will request from time to time for new patient's check-ins. When a new check-in comes, it will be analyzed by the app to find out if an alert needs to be notified in the notification area. Maybe to avoid pull request to download the check-in from time to time a push notification system like GCM will be used.	
Functional requirements	Number 12

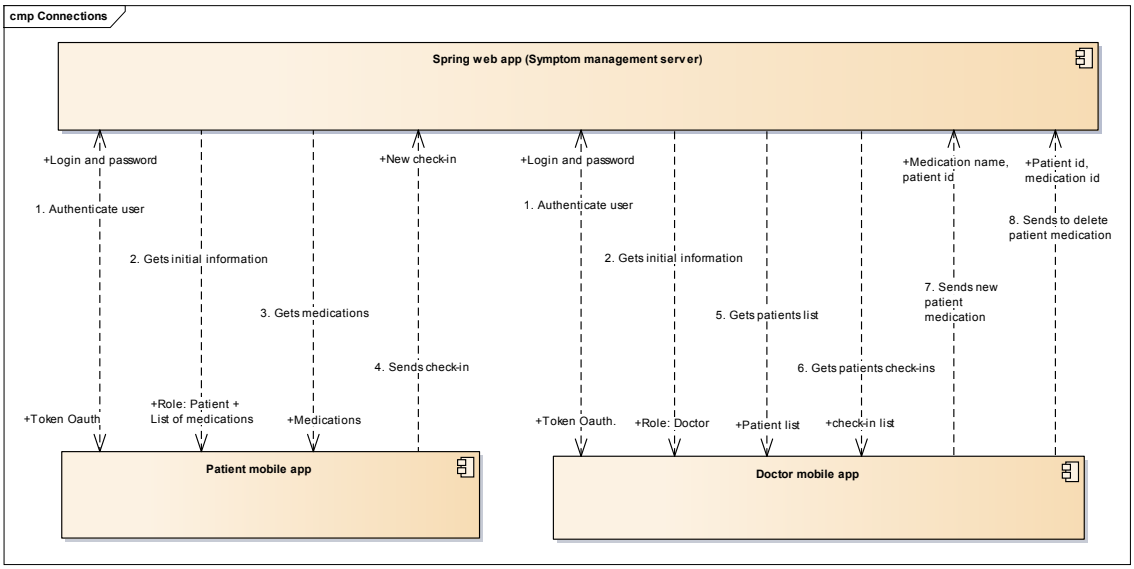
System design. General description

The system is composed by two main subsystems.

1. A Spring web App that exposes an API REST. This web app has a database to store all the information about doctors, patients and the relationships between them. In the chapter "Server design" you can read the technical information about this subsystem.

2. An Android mobile app for users (doctors and patients) to interact with the system. This App has two set of functionalities, one for doctors and another for patients. In the chapter "Mobile App design" you can read the technical information about this subsystem.

The next diagram shows how these two main subsystems communicate to each other. All of the communications are over HTTPS. Two independent boxes in the diagram represent the Android mobile App, despite that both boxes are the same mobile App, to show the communication between the server and the functionality for the doctors and the patients.



The mobile app has been designed to use a local database with the same database schema than the server. The main reasons for this decision are:

- It allows users to use the app offline. A process in background synchronizes data to and from the server transparently for the users.
- It reduces the amount of traffic between the app and the server. This means cheaper costs for the users.
- It reduces the server overhead because there are fewer requests from clients.

In order to protect medical information, it's important to know that doctors' app will not save any information about the patients names.

Communications between subsystems

This section describes the communications between mobile app and the server.

Path	1. Authenticate user
From	Patient and doctor mobile App
Sent data	Login and password
Return data	Token OAuth
Description	
The user authenticates to the server with an OAuth grant password mechanism.	

Path	2. Get initial information
From	Patient and doctor mobile App
Sent data	

Return data	If authenticated user is a patient:
	The server returns in JSON format the list of patient medications, information that tells app the user is a patient, and finally patient and doctor information.
	
	{Role: “patient”,
	patient: {id: long, name: text, lastname: text, birthdate: date,
	medications:[{id: long, name:text, active: bool}]},
	doctor: {id: long, name: text},
	}
	
	if authenticated user is a doctor:
	The server returns in JSON format information that tells app the user is a doctor, and doctor information.
	
	{Role: “doctor”, doctor: {id: long, name: text, lastname: text }}

Description	
The app gets the patient and doctor initial information from the server	

Path	3. Sends medications
From	Patient mobile App
Sent data	
Return data	The list of patient medications in JSON format {medications:[{id:long, name:text, active:bool}] • Active: false indicates that his/her doctor has been deleted the medication.
Description	
The app gets the patient medications list from the server	

Path	4. Sends check-in
From	Patient mobile App
Sent data	A JSON format with all the fields that patient filled in during check-in {howbad: text, painstop: text, takemedication: bool, checkin_date: date, checkin_medications:[{medication_id: long, takeit: bool, takeit_date:date}]}

	• howbad values: “well-controlled,” “moderate,” or “severe” • painstop values: “no,” “some,” or “I can’t eat.” • date_checkin: date and time in ISO 8601 format
Return data	
Description	
The app sends a new check-in to the server	

Path	5. Gets patients list
From	Doctor mobile App
Sent data	
Return data	A patients check-in list in JSON format {[id: long, name: text, lastname: text, birthdate: date]}
Description	
The app gets his/her patient list from the server	

Path	6. Gets patients check-ins
From	Doctor mobile App
Sent data	Patient name
Return data	A patients check-in list in JSON format [{patient_id: long, show_alert:Boolean, checkinlist: [{howbad: text, painstop: text, takemedication: bool, checkin_date: date, checkin_medications: [{medication_id: long, takeit: bool, takeit_date: date}]}] //end_checkinlist }] • show_alert: true or false. To indicate apps that a notification needs to be alert the doctor. • howbad values: “well-controlled,” “moderate,” or “severe” • painstop values: “no,” “some,” or “I can’t eat.” • date_checkin: date and time in ISO 8601 format
Description	
The app gets a new check-ins from the server	

Path	7. Sends a new patient medication
From	Doctor mobile App
Sent data	The patient medication name in the request body
Return data	The medication object saved in the server database name:text
Description	
The app sends a new medication for a patient to the server	

Path	8. Sends to delete a patient medication
From	Doctor mobile App
Sent data	The patient id and the medication id in the request URL
Return data	
Description	
The app sends to delete a patient medication to the server	

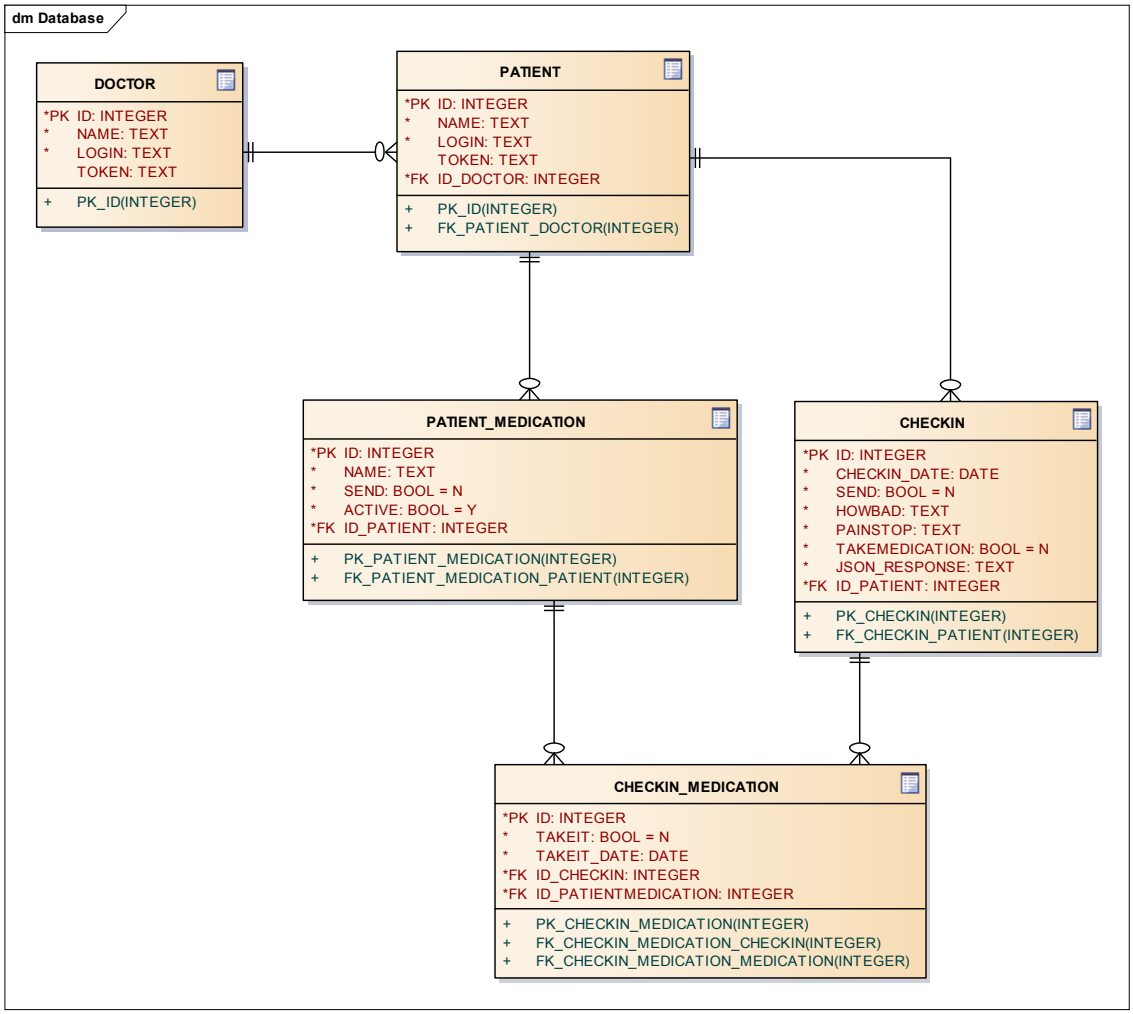
Server design

The server will be implemented in Spring and will run in a Tomcat server with MySQL database to store all the information about the system. The server will allow authenticate users with login and password with OAuth password grant mechanism. Every user will have an independent account and all of their information will be related in the same database. Doctors will have DOCTOR_ROLE assigned and Patients will have PATIENT_ROLE, and of course doctors functionality will be protected with DOCTOR_ROLE and patient functionality will be protected with PATIENT_ROLE. This blocks any attempt to patients to access the doctors functionality and vice versa. In the case a doctor is a patient too, then he/she will have a different account as a patient.

The next three sections describe the server database schema, server class diagram and the REST API.

Database schema

The next diagram shows the system database schema.



This database schema is replicated at mobile app to allow users to work offline and to improve data access from the app. But, not all the information is replicated at mobile app. The chapter "Database" from "Mobile App design" describes what information is saved for the mobile app.

The next list describes the database tables.

- **DOCTOR**. Represents the doctors of the system. This table is maintained by an external system. There is a one-to-many relationship with patient table.
- **PATIENT**. Represents the patients of the system. A patient has a check-ins list and a medications list. Each patient has only one doctor (I assume this decision in my design)
- **PATIENT_MEDICATION**. Represents the patient medications. Each patient_medication row belongs to one patient. This table is maintained by doctors app.
- **CHECKIN**. Represents the patients check-in. Each checkin row belongs to one patient. This table is maintained by patients app.
- **CHECKIN_MEDICATION**. Represents the answer for each patient medication during each check-in. It's a many-to-many relationship between CHECKIN and PATINET_MEDICATION tables. This table is maintained by patients app.

The Next fields are only used by the mobile app and it doesn't make sense at server database:

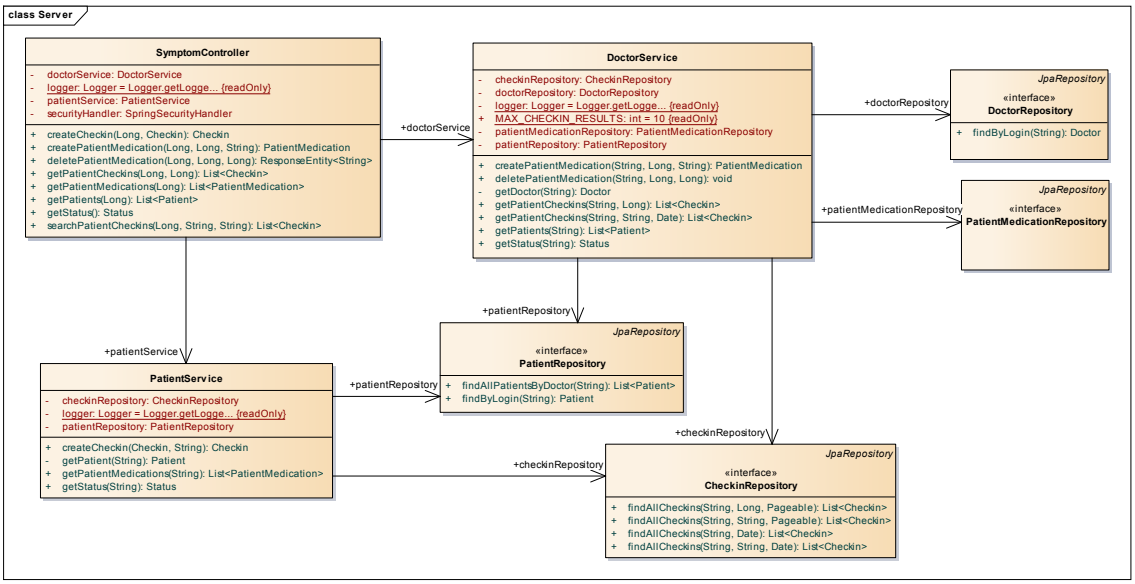
- **SEND**: BOOL. This field is used to indicate if the information has been sent to server successfully. The default value is 'N'. When the service running at mobile device sends this information successfully, changes the field to 'Y'.
- **TOKEN**: TEXT. This field is used to indicate if user is using Android authenticator manager to mange the authentication process.

This section describes some fields to clarify the model. The rest of the fields are clear enough.

- **ACTIVE**: BOOL. This field is used to indicate that patient medication has been deleted by doctor. The default value is 'Y' to indicate that patient medication is active and the patient has to answer if they are taking it. If the doctors delete a patient medication the value changes to 'N'.
- **CHECKIN_DATE**: DATE. Represents the date and time of the check-in.
- **HOWBAD**: TEXT. Represents the answer for: How bad is your mouth pain/sore throat?". The values can be: "well-controlled", "moderate", "severe".
- **PAINSTOP**: TEXT. Represents the answer for: Does your pain stop you from eating/drinking?". The values can be: "no," "some," or "I can't eat."
- **TAKEMEDICATION**: BOOL. Represents the answer for: "Did you take your pain medication?".
- **TAKEIT**: BOOL. Represents the answer for: "Did you take your ....?" for each patient medication.
- **TAKEIT_DATE**: DATE. Represents the date and time when patient took his/her medication

Class diagram

The next diagram shows the server class diagram.



The server is designed by three layers:

- Controller layer: This layer publishes an REST API for mobile apps. Only authorized users will access REST API methods. This layer contains a **SymptomController** class. This class accepts requests from mobile apps and calls service layer in order to access to the business logic. Other function in this layer is transform JSON objects to Java and Java objects to JSON in order to communicate with mobile apps. All JSON to Java transformations are made by Jackson library. The next section describes the REST API in more detail.
- Service layer: This layer represents the business logic. This layer accepts calls from Controller layer and invokes to the next layer classes in order to access to the database. This layer contains **DoctorService** class to implement doctor business logic and **PatientService** class to implement patient business logic. In this layer, all business methods verify that every user can only access to their own information. For example, if a malicious user invokes an URL and changes the ids to access information that don't belongs to that user, a Security exception will be thrown.
- Data access layer: This layer accesses to the database to insert, update, delete and query patient and doctors' data. This layer contains **DoctorRespository** class to access DOCTOR table, **PatientRespository** class to access PATIENT table, **CheckinRespository** class to access CHECKIN table, **PatientMedicationRespository** class to access PATIENT_MEDICATION table, **CheckinMedicationRespository** classes to access CHECKIN_MEDICATION table.

REST API

The server publishes an API that is used by the mobile App. This API will be accessible at https://server/symptom.

All /doctors/xxx paths are protected by DOCTOR_ROLE. All /patients/xxxx paths are protected by PATIENT_ROLE.

These are the operations that API implements:

- /oauth
  Operation type: Authenticated
  Description: The user sends username and password and receives a token if he/she is a valid user or an error if not.

- /status
  Operation type: GET
  Roles allowed: DOCTOR_ROLE, PATIENT_ROLE
  Description: If the user is a patient then receives a list of his/her medications and his/her personal information stored in the system. If the user is a doctor then receives only his/her personal information.
  JSON_FORMAT: see Path: 2. Get initial information at "Communications between subsystems section".

- /patients/{patient_id}/medications
  Operation type: GET
  Roles allowed: PATIENT_ROLE
  Description: The user with {patient_id} identifier receives a list of his/her medications in JSON format.
  JSON_FORMAT: see Path: 3. Gets medication at "Communications between subsystems section".

- /patients/{patient_id}/checkins
  Operation type: POST
  Roles allowed: PATIENT_ROLE
  Description: The user with {patient_id} identifier sends a check-in to the server in order to be saved in the central database.
  JSON_FORMAT: see Path: 4. Sends check-in at "Communications between subsystems section".

- /doctors/{doctor_id}/patients
  Operation type: GET
  Roles allowed: DOCTOR_ROLE
  Description: The user with {doctor_id} identifier receives a list of his/patients in JSON format.
  JSON_FORMAT: see Path: 5. Gets patient list at "Communications between subsystems section".

- /doctors/{doctor_id}/patients/{patient_id}/checkins
  Operation type: GET
  Roles allowed: DOCTOR_ROLE
  Description: The user with {doctor_id} identifier receives a check-in list of his/her patient with {patient_id} identifier in a JSON format. The server only will send the last 10 days of check-ins because it would be a lot of information.
  JSON_FORMAT: see Path: 6. Gets patients check-ins at "Communications between subsystems section".

- /doctors/{doctor_id}/patients/checkins/search?patientName="patientName"& dateFrom="date"
  Operation type: GET
  Roles allowed: DOCTOR_ROLE
  Description: There are several possibilities:
  1. Parameter patientName is not null and dateFrom is not null: The user with {doctor_id} identifier receives a check-in list of his/her patient whose name is "patientName". The check-ins returned were filled after date "date".
  2. Parameter patientName is not null and dateFrom is null: The user with {doctor_id} identifier receives a check-in list of his/her patient with name "patientName". The server only will send the last 10 days of check-ins.
  3. Parameter patientName is null and dateFrom is not null: The user with {doctor_id} identifier receives a check-in list of **all** his/her patients that were filled after date "date".
  4. Parameter patientName is null and dateFrom is null: An error will be returned.

  JSON_FORMAT: see Path: 6. Gets patients check-ins at "Communications between subsystems section".

- /doctors/{doctor_id}/patients/{patient_id}/medications
  Operation type: POST
  Roles allowed: DOCTOR_ROLE
  Description: The user with {doctor_id} identifier sends a new medication for his/her patient with {patient_id} identifier in a JSON format.
  JSON format: see Path: 7. Sends new patient medication at "Communications between subsystems section".

- /doctors/{doctor_id}/patients/{patient_id}/medications/{medication_id}
  Operation type: DELETE
  Roles allowed: DOCTOR_ROLE
  Description: The user with {doctor_id} identifier deletes a medication with {medication_id} for his/her patient with {patient_id} identifier.
  JSON format: see Path: 8. Sends to delete patient medication at "Communications between subsystems section".

Mobile App design

The mobile app will be the same app for all users. Users will have to authenticate with their username and password in order to allow them to work with the system. The mobile app will use a local database that will be accessed via a **ContentProvider** class. The communications with the server will use Retrofit framework. Finally, the mobile App will have two **services**, one to download medications and check-ins and another to send check-in information.

The next three sections describe the mobile app database schema, the class diagram and the user interfaces.

Database schema

The Mobile App uses the same database schema than the server because the app has been designed to reduce data traffic. Because of this, both patient and doctors app need to save and query to this local database, and a process in background synchronize the local data with the server. Of course, each user will only have the data that belongs to each user.

From the patients app point of view, the information that is saved locally is:

- **DOCTOR table**: A row that represents his/her doctor. I assume in my design that a patient has only one doctor.
- **PATIENT table**: A row that represents his/her information.
- **PATIENT_MEDICATION table**: Only his/her medications. Maybe less than ten rows.
- **CHECKIN table**: Only his/her check-ins. Maybe less than a five hundred rows.
- **CHECKIN_MEDICATION table**: Only his/her answers for each medications in every check-in. Maybe less than a thousand rows.

From the doctors app point of view the information that is saved locally is:

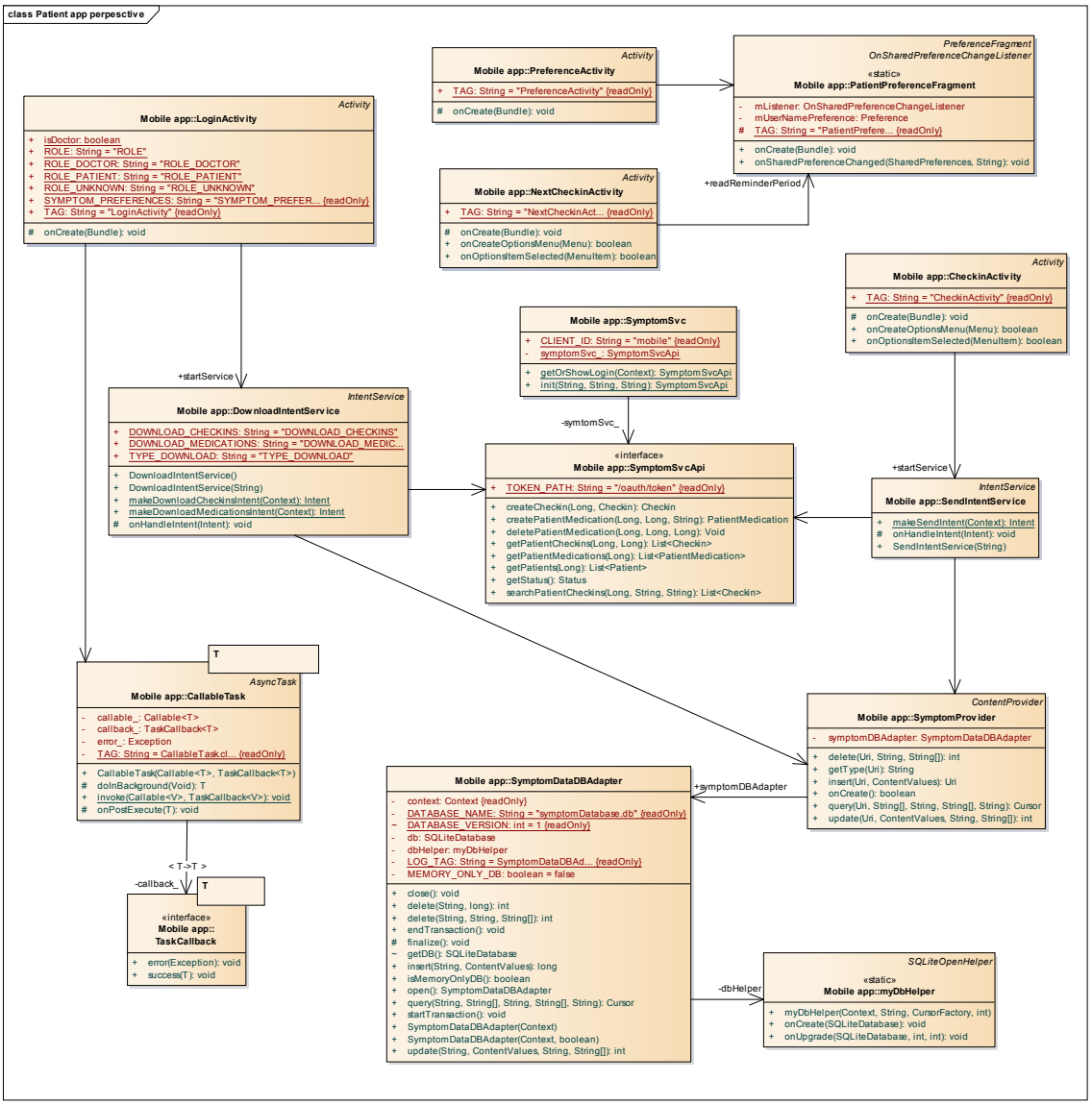
- **DOCTOR table**: Only one row that represents his/her information.
- **PATIENT table**: No information is saved for security reasons.
- **PATIENT_MEDICATION table**: The list of his/her patients' medications. Maybe less than a thousand rows.
- **CHECKIN table**: The list of his/her patients' check-ins. Maybe less than ten thousand rows.
- **CHECKIN_MEDICATION table**: The list of his/her patient check-in's answers for each medication. Maybe less than ten thousand rows.

To avoid uncontrolled growth of some tables of the doctor mobile app, it would be necessary a functionality that deletes all historical information about patients that leave the treatment. This is not a requirement of this system and for academic reason it won't be implemented.

Class diagram

This section shows the app class diagram in two different diagrams for a better explanation.

The following diagram shows the mobile app class diagram from the patient point of view.



Patient classes description

- **LoginActivity:** This activity class has these responsibilities:
    - It shows a Login screen.
    - Once the user inserts his/her username and password and press Login button, the activity authenticates the user calling **init**() method of **SymptomSvc** class. This method makes a request to the server using */oauth* REST API method. Once the user is authenticated and only if it's the first time the user uses the app, the activity will call **getStatus**() from **SymptomSvcApi** class to retrieve who is this user, his/her role in the system and the medication list if the user is a patient. This method calls REST API method: */status* at sever. Once the information is returned it will be saved in the database using

SymptomProvider. All of this process is made via **CallableTask** class for doing it in background.

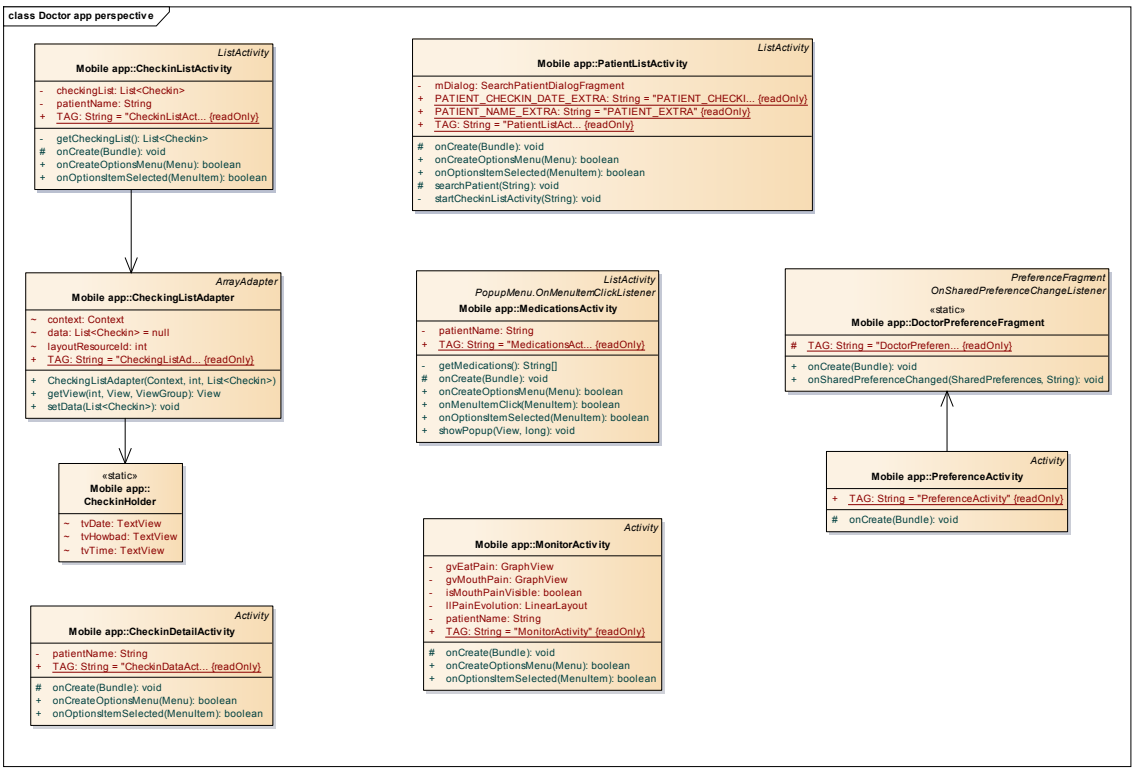
- If the user is a patient and the reminder alarm is not started, the activity will start an Alarm for reminders check-in. When the alarm is fired, a notification is sent to the notification area and a new alarm will be program for next reminder at the interval frequency defined by the user. The activity will need to read the SharedPreference object in order to know that value.
- If no download alarm is started, the activity will start an Alarm for download new medications or new checkins depending if the user is a patient or doctor. When the alarm is fired, the app starts **DownloadIntentService** service to start a process in background to download medications or checkins. This is done sending a different intent object to the service. Once the alarm is fired, a new alarm will be program for next download process at the interval frequency defined by the user. The activity will need to read the SharedPreference object to know that value.
- Finally the activity starts the **NextCheckinActivity** class if the user is a patient or **PatienListActivity** if the user is a doctor.

- **PreferenceActivity** and **PatientPreferenceFragment**: This activity allows user to change the reminders frequency and the download service frequency.
- **NextCheckinActivity:** This activity shows the next check-in screen with the date and time of the next reminder. The activity reads the SharedPreference reminder frequency value defined in **PatientPreferenceFragment** class. Additionally allows user to press a button on the screen to start a new check-in manually. If the user presses this button the activity starts **CheckinActivity** class.
- **CheckinActivity:** This activity shows the wizard screen that allows user to answer all questions defined in the check-in. The first action is to read the patient medication list from local database using **ContentProvider** class. This is done in background using **CallableTask**. If no medication is present in the local database, an alert is shown to the user and the process can't continue. In other case, the activity will present a wizard screen with all the questions. Once the user has finished, the activity will save the check-in data into the local database using **ContentProvider** class and will start the **SendIntentService** class to send the check-ins not sent it yet.
- **DownloadIntentService:** This service downloads medications or checkins depending of the intent object sent it for the LoginActivity class. The service will call **getPatientMedications**() or **getPatientCheckins**() from **SvcSymptomApi** class. These methods calls REST API methods: */patients/{id_patient}/medications* and /doctors/{doctor_id}/patients/checkins/search?dateFrom=today respectively at server.

  Once the information is returned, it will be saved into the database with **SymtomContentProvider** class and a new alarm will be programed again at the frequency defined by the user in the preference screen. If the user is a doctor he/she will be alerted if a patient experiences 12 hours of "several pain", "16 or more hours of moderate or severe pain" or 12 hours of "I can't eat". The service will find out in the returned check-in list information and will send a notification with a sound to the notification area.

- **SendIntentService:** This service class will send to the server all check-ins that haven't been sent it yet. For this, the service first query the local database, using **ContentProvider** class, for checki-ns with status 'not send it', then the service uses **createCheckin**() from **SvcSymptomApi** class, that calls REST API method: */patients/id/checkin* in order to sent it to the server, and finally, the service updates check-in status with value 'sent it' in local database.
- **CallableTask** and **TaskCallback:** The **CallableTask** is an AsyncTask for doing work in the background. This class accepts two parameters. The first one is a **Callable** object with a method named call, that it will be executed in the **doInBackground**() method of the **CallableTask**. The second one is an **TaskCallback** object with two methods: **success**() that it will be executed if the **CallableTask** finishes ok and **error**() if not. This class allows other classes to execute work in background.
- **SymptomProvider, SymptomDataDBAdapter** and **myDBHelper:** Theses three classes allow other classes to access to the local database for reading and writing information.
- **SymptomSvc** and **SymptomSvcApi:** These two classes allow the other classes to access to the server for requesting information.

The following diagram shows the mobile app class diagram from the doctor point of view. Despite that these classes use the LoginActivity, DownloadIntentService, CallableTask, SymptomProvider and SymptomSvcApi classes, all of them are not shown because they are explained in the patient class diagram.



Doctor classes description

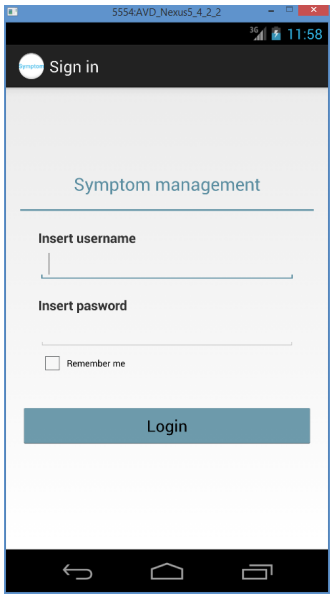
- **PreferenceActivity** and **DoctorPreferenceFragment**: This activity allows user to change download service frequency.

- **PatientListActivity:** This activity shows the doctor patient list screen. The activity calls **getPatients**() method from **SymptomSvcApi** class that calls REST API method: */doctors/{id}/patients* in order to download this information. Once the information is returned is showed in the screen with a String array adapter. All of this process is done in background using **CallableTask** class. If the user presses one row of the patient list the **CheckinListActivity** will be started in order to show patient check-in list. From the menu of this activity the user can search a patient by his/her name. A dialog will be showed to enter the patient name. Once the user enters the patient name and presses the search button, it will call **searchPatientCheckins**() from **SymptomSvcApi** class that calls REST API method: */doctors/{doctor_id}/patients/checkins/search?patientName=.* If no information is returned from the server an error will be shown. If the information is returned the **CheckinListActivity** will be started.
- **CheckinListActivity**: This activity shows the last 10 days of patient's check-ins. The activity class will read from the local database using **SymptomProvider** class, to check if exist this information. If not, it calls **getPatientCheckins**() from **SvcSymptom** class that calls REST API method: */doctors/{id_doctor}/patients/{id_patient}/checkins* in order to download this information. Once the information is downloaded, it is saved into the local database and a **CheckinListAdapter** class is used to be showed in the screen. All of this process is done in background using **CallableTask** class. If the user presses on one row of the check-in list, the **CheckinDetailActivity** will be started.
- **CheckinDetailActivity:** This activity shows the check-in data information screen. The activity reads this information using **ContentProvider** class. This process is done in background using **CallableTask** class.
- **MonitorActivity:** This activity shows a graph with the patient pain evolution. The doctor can access this activity from the menu of CheckinListActivity. The activity reads this information using **ContentProvider** class. This process is done in background using **CallableTask** class. The user can switch the pain evolution graph pressing a button on the screen.
- **MedicationsActivity:** This activity shows the patient medication list. The doctor can access this activity from the menu of **CheckinListActivity** class. The activity reads this information using **ContentProvider** class. This process is done in background using **CallableTask** class. The doctor can add new medication writing the medication name into the field and pressing Add button. The activity will call **createPatientMedication**() method from **SymptomSvcApi** class that calls REST API POST method: */doctors/{doctor_id}/patients/{patient_id}/medications.* Once the server returns the success of the operation, the new medication is saved into the local database and is shown into the medication list screen. This process is done in background using **CallableTask** class. Another possibility is to delete a patient medication pressing a long time on a medication row. A contextual menu will be shown with a delete option. If the doctor presses this option, the activity will call **deletePatientMedication**() method from **SymptomSvcApi** class that calls REST API DELETE method: */doctors/{doctor_id}/patients/{patient_id}/medications/{medication_id}.* Once the server returns the success of the operation, the medication is deleted from the local database and removed from the medication list screen. This process is done in background using **CallableTask** class.

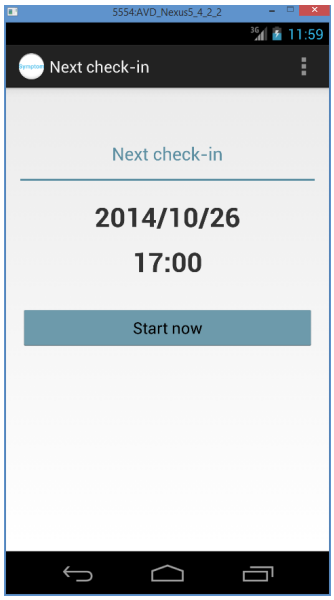
User Interfaces

These are the app user interfaces. Maybe they will change a little bit when I start to code, but right now they offer a good vision of the final app.

Patient screens:



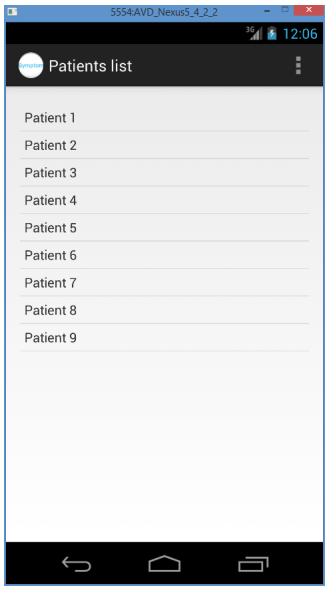
Login screen (same for doctors)



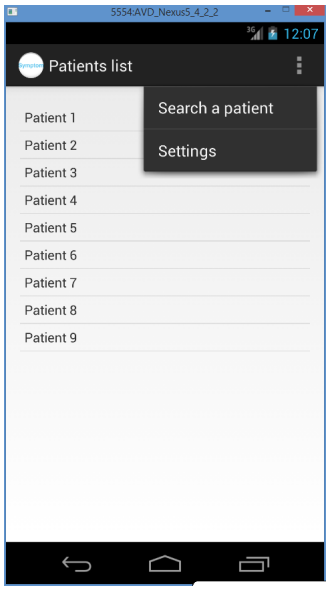
Next check-in

Check-in wizard

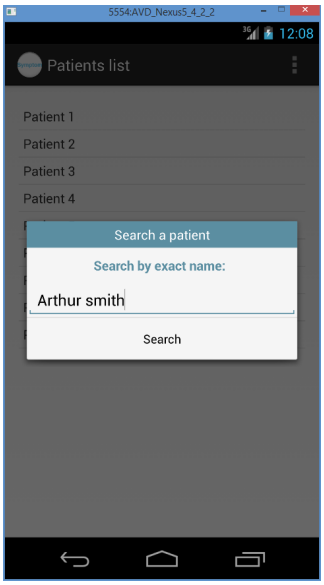
Doctor screens:



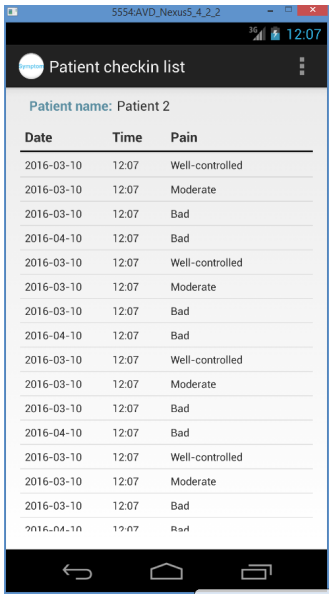
Patient list



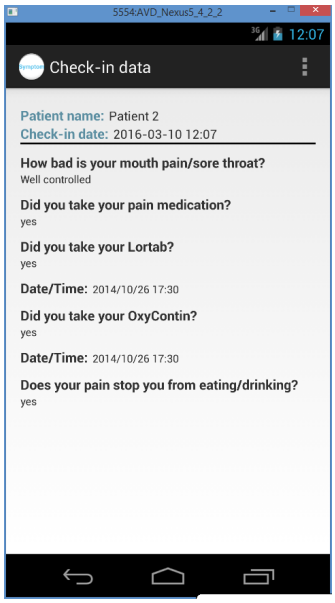
Menu search a patient



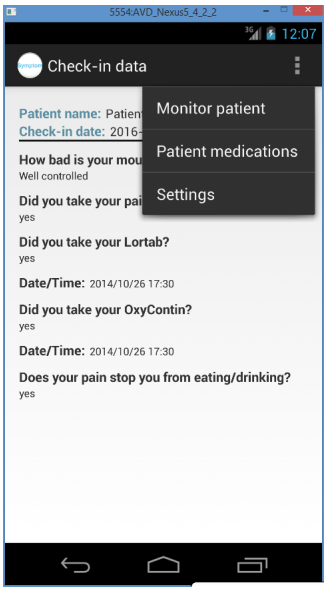
Search a patient



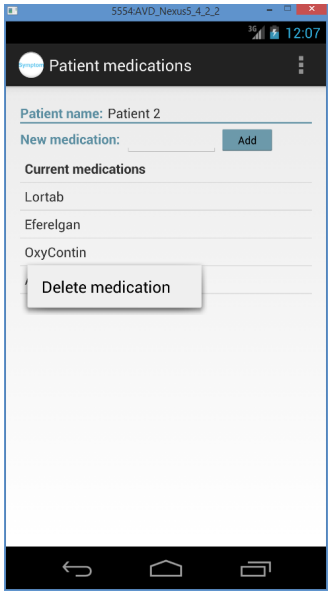
Patient list check-ins



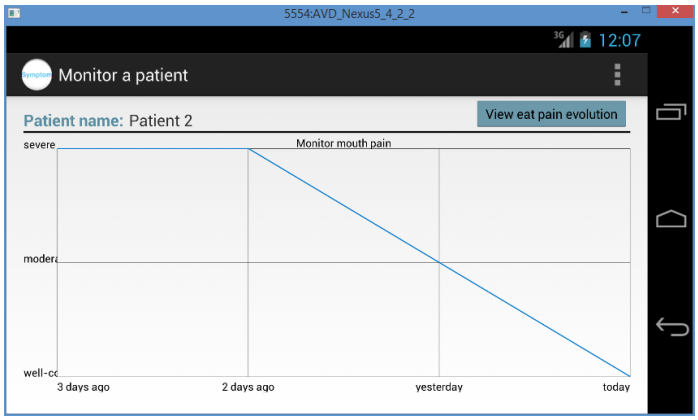
Check-in data



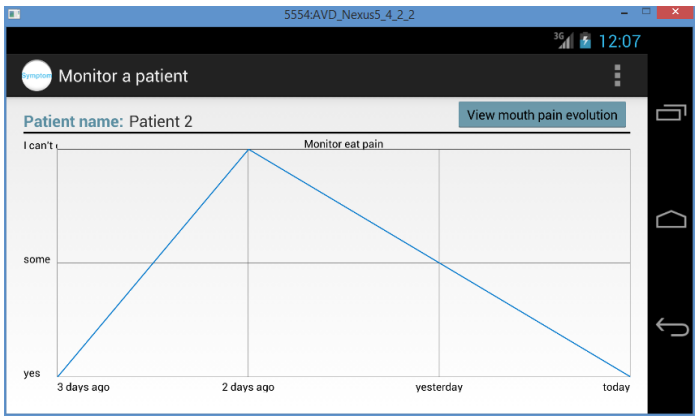
Doctor menu



Patient medications



Monitor a patient. Graph pain mouth evolution



Monitor a patient. Graph pain eat evolution