System Architecture

Addendum

medsync

This document serves to rectify the previously presented architecture for this system. Here we will explain:

1. How this issue was detected (and why it is an issue)
2. How this issue was solved
3. The solution we found
4. Why we will implement our found solution
5. The consequences of these changes for the rest of the project

Detection of the issue (and why it is an issue)

The architecture we developed in iteration 1.1 did not present any major holes, but there was never a conviction that a document-based DB would be the best solution to treat a patient's vital signs. Such DB is the best solution for situations where entities of the same category have similar, flexible attributes, which we'd want to tackle cleanly (for example, a patient having an attribute that a different patient wouldn't have). Considering our system, we initially opted to keep the patient's data in the document-based DB instead of the main relational DB, since we considered the patient entities weren't related between each other, and besides we'd have a big amount of data coming in continuously and we'd need to keep the history of such data. At the time, we discarded a time-series DB due to a misconception we had about its usage, and we opted for a document-based DB, which would turn out to be a big mistake. Not only do the patients not have the characteristics previously mentioned (attribute flexibility), a document-based DB is extremely inefficient facing a big influx of data, since whenever such data comes, the entire document pertaining a patient must be read and updated. With that said, and having in account this conclusion we reached, it became clear that the project's progress had come to an halt and this architecture issue had to be solved before anything else could be done.

How to solve this issue

After realizing that a document-based DB wouldn't be the most suitable for this project, we were forced to rethink our options. Considering we still have a big data influx coming from several sources, we needed to find a model that could handle this, besides being able to interpret data history that would come each second. For that, a timeseries model would indeed be the best, but a timestamp:value format would mean that we'd need several databases if we wanted only a value associated to a DB, which would obviously be a terrible solution. The challenge was indeed to adjust a timeseries model towards something we could use.

The solution we found

We arrived at the conclusion that we could indeed use a timeseries model, if we adjusted the way how we handled the data. Instead of immediately sending received data to the DB, we'd wait for all data pertaining a particular bed to arrive from all (4) sensors, and then simultaneously send all that data to the DB. We'd then have a structure like:

```
name: vitals
-----------------
time                    bed     heart_rate      temperature     oxygen  systolic  diastolic
2015-08-29T07:06:00Z    1       65              36              90      102       72
```

(This output comes from using InfluxDB, which is the tool we'll be using in our system)

By referencing the bed from which the data came instead of identifying the pacient, we can keep the same data structure even if a patient is not in a bed, or even if the patient changes within a bed, or even when a certain value is not read (it will show null in the DB). In a similar manner, every data relative to the patient would now need to be passed over to the relational DB, which now will include every data within the system instead vital sign data (and its history)

Note: In the example above, the "bed" attribute is called a tag, which in InfluxDB represents an indexed attribute (almost like a primary key) which guarantees that queries that filter by timestamp + bed number are still pretty efficient, resulting in our system still taking advantage of a timeseriesDB.

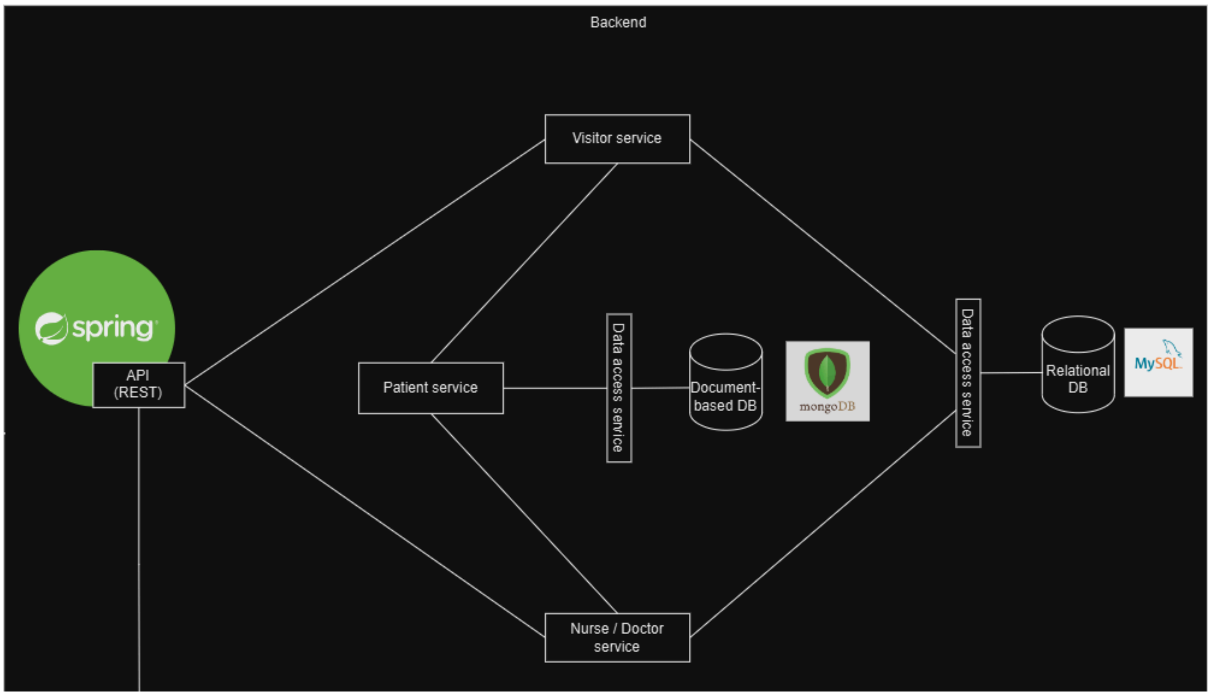
Why this solution

The timeseries model was always the most adequate to capture data that are sent repeatedly and periodically. This model not only allows us to store that data efficiently, it also enables us to make an analysis of historical data, which many other models do not allow. Like we explained above, a document-based model would probably be the worst answer one could come up with. A relational DB would also not be adequate since the table containing vital signs would quickly become huge with a very high number of data coming every second. A key-value DB appears to be a viable solution, since it is easily scalable and it is good for consecutive and periodic data writing, but a timeseries model also allows one to make an analysis of historical data (for example, within a time frame), and furthermore key-value databases are not optimized for timestamp related queries. Given this, it is easy to understand, given all the pros and cons, that a timeseries model is the most adequate towards storing vital data of all patients.

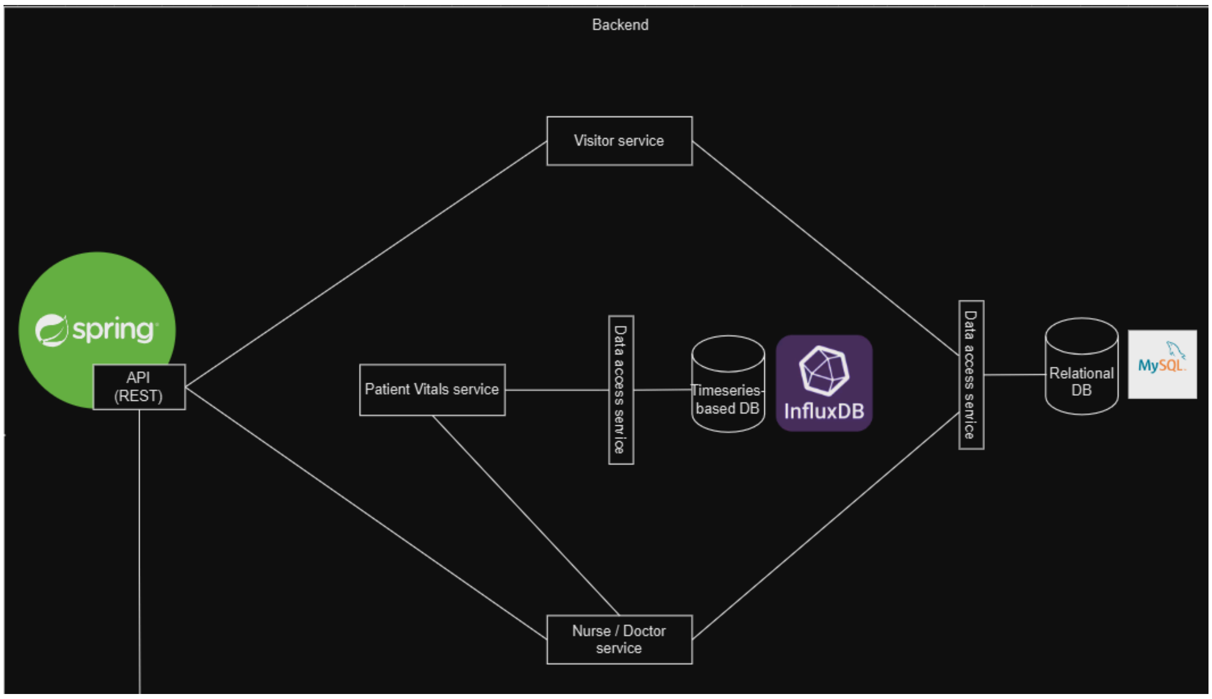
Consequences of these changes

Given that we still hadn't made much progress in terms of code until now, the impact on that field is minimal. However, the interaction flux changes significantly. Furthermore, the DB schema also ends up being adjusted, mainly by the fact that patient data is now present in this DB. Besides that, this also ends up altering some services and creating some services that weren't previously created.

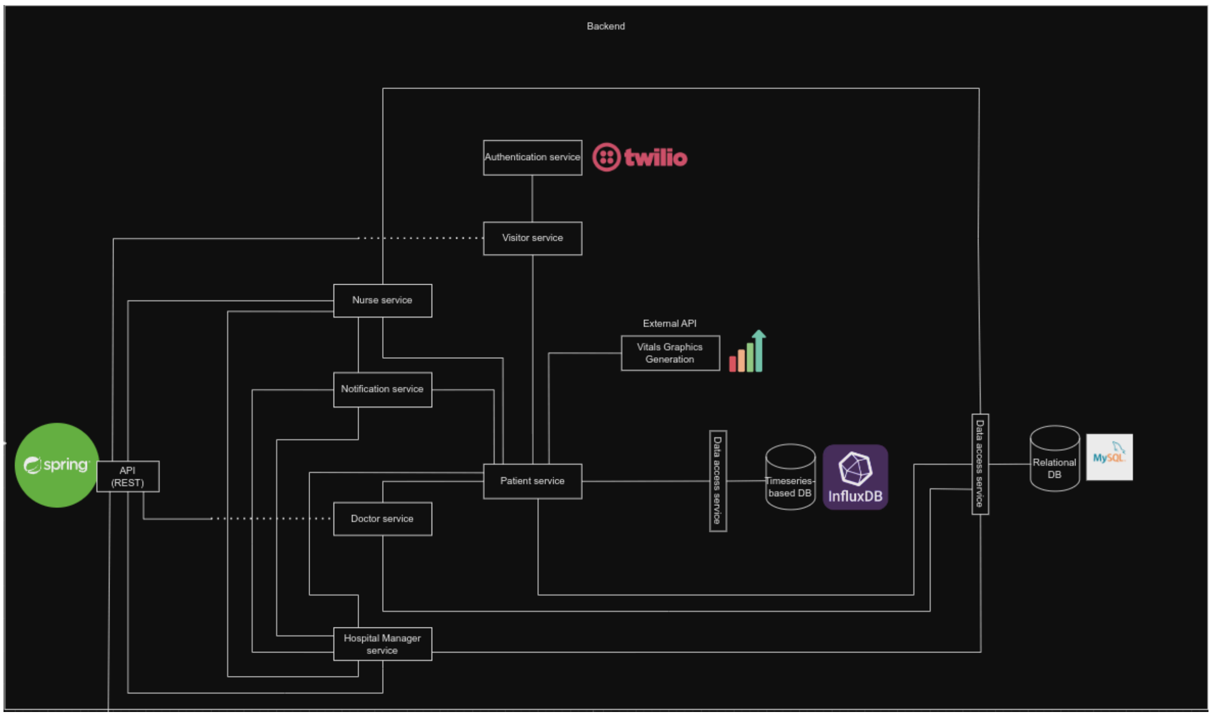
Before:



After:

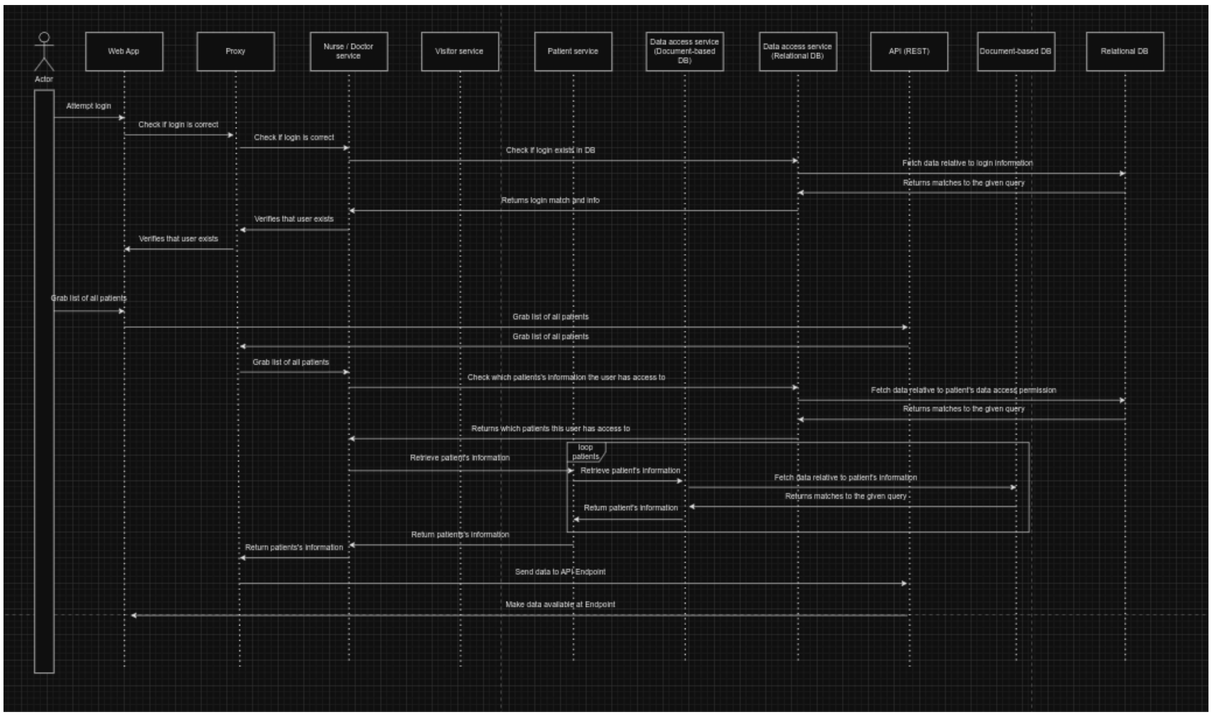


Additionally, in order to really close this chapter, we took this chance to refine the services we had, as well as adding a new notifications service, and due to the introduction of a timeseries DB we are now also able to bring in an external API from QuickChart. Besides that, we also added an authentication services that uses the Twillio tool.



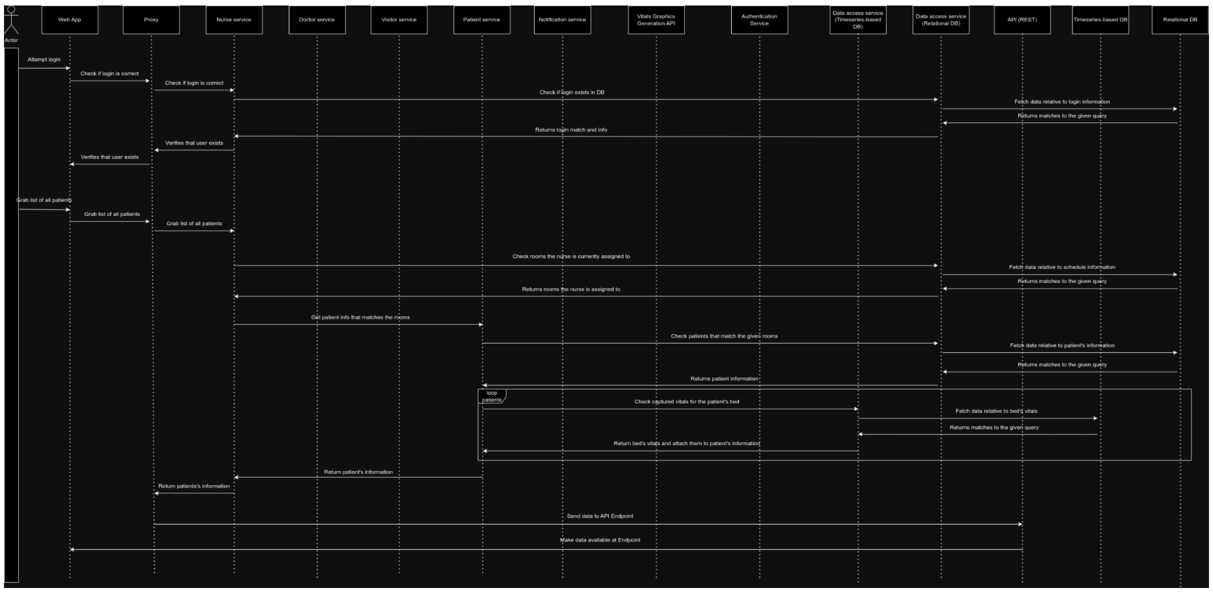
With these changes also comes a different interaction flux. Below you can see the comparison between the flux we previously had vs our new flux when a Nurse logs in and checks the main page where all the patient information is available.

Before:



Actor
Web App
Proxy
Nurse / Doctor service
Visitor service
Patient service
Data access service (Document-based DB)
Data access service (Relational DB)
API (REST)
Document-based DB
Relational DB
Attempt login
Check if login is correct
Check if login is correct
Check if login exists in DB
Fetch data relative to login information
Returns matches to the given query
Returns login match and info
Verifies that user exists
Verifies that user exists
Grab list of all patients
Grab list of all patients
Grab list of all patients
Grab list of all patients
Check which patients's information the user has access to
Fetch data relative to patient's data access permission
Returns matches to the given query
Returns which patients this user has access to
loop patients
Retrieve patient's information
Retrieve patient's information
Fetch data relative to patient's information
Returns matches to the given query
Return patient's information
Return patients's information
Return patients's information
Send data to API Endpoint
Make data available at Endpoint

After:


Actor
Web App
Proxy
Nurse service
Doctor service
Visitor service
Patient service
Notification service
Vitals Graphics Generation API
Authentication Service
Data access service (Timeseries-based DB)
Data access service (Relational DB)
API (REST)
Timeseries-based DB
Relational DB
Attempt login
Check if login is correct
Check if login is correct
Check if login exists in DB
Fetch data relative to login information
Returns matches to the given query
Returns login match and info
Verifies that user exists
Verifies that user exists
Grab list of all patients
Grab list of all patients
Grab list of all patients
Check rooms the nurse is currently assigned to
Fetch data relative to schedule information
Returns matches to the given query
Returns rooms the nurse is assigned to
Get patient info that matches the rooms
Check patients that match the given rooms
Fetch data relative to patient's information
Returns matches to the given query
Returns patient information
loop patients
Check captured vitals for the patient's bed
Fetch data relative to bed's vitals
Returns matches to the given query
Return bed's vitals and attach them to patient's information
Return patient's information
Return patients's information
Send data to API Endpoint
Make data available at Endpoint

With these changes, almost every data is present in the relational DB, Since the timeseries DB uses the beds as a reference (tag), to access this data, the Nurse service needs to check which rooms the Nurse is responsible for, and then the Patient service handles returning the patient's info for the patients that are present in such rooms. What will be returned by the API is a combination of the data found in the relational DB and the data found in the timeseries DB.

Other remarks regarding new services

Authentication service - we still didn't have a service that would adequately handle visitors. By using the twillio tool, we can assure security in hospital visits. The visitor service will call the authentication service so that the visitor can insert a confirmation code (sent to their phone) to confirm their identity at an early stage, allowing the visitor to receive the location of the bedroom that they are to visit.

Notification service - in our system we'll have different types of notifications; for example, the nurses will have 2 different types of notifications, regarding critical states of patient, but also regarding notifications such as needing to go clean a bed that had a discharged patient in it. Due to this high variety in notification types, we will have a service that handles this by interpreting the incoming data and acting accordingly.

Nurse / Doctor service - these services were previously packed together (as well as the Hospital Manager service), but in reality these services serve very different purposes and behave differently. For this reason, and despite these services having the same connections, they were separated in the architecture

Vital Graphics Generation - another external API which specializes in data originated from timeseries DB queries, used to create graphics with historical data. This API is necessary for a good data usage, in order to visualize graphs regarding historical vital data for each patient. The use of this kind of API was not possible previously, but it is now due to our usage of a timeseries DB.