Dentanoid

Group 20

[Joel Mattsson, James Klouda, Cornelia Olofsson Larsson, Mohamad Khalil, Jonatan Boman, Lucas Holter]

The user stories covered by the system:

NOTE: The links below are linked to the Wiki pages and the associated issues. All of the user stories don't have issues linked to them because the group agreed that an issue shall only be linked to the user story that most accurately describes the implementation.

Requirement 1

1. As a user I want to be able to authenticate myself so that i can access my data - [Related issue(s): Basic Auth Server, Implement authentication in the API, Create sign in/sign up page, Add authentication handling]

Requirement 2

2. As a user I want to be able to register an account so that I can later authenticate my self - [Related issue(s): Create CRUD functionality for Patient entity, Create CRUD functionality for Dentist entity, Implement patient endpoints for patient api, Create sign in/sign up page, Create CRUD functionality for Patient entity]

Requirement 3

3. As a user I want to use a map so that I easily can see the clinics around me - [Related issue(s): Map navigator, Create dentist endpoints for Patient API, Display statistics of clinics]

Requirement 4

4. As a user I want to be able to schedule an dentist appointment so that I can fix my teeth - Related issues: [Simple dentist UI, Create appointment endpoints for Patient API, Enhance Patient Appointments response to Include Dentist Information, Page for viewing clinics and available times, Filter clinics by timeslots, CRUD for appointments]
5. As a user I want to find all dental clinics in my area so that I can more easily book an appointment [Create clinic endpoints for the Patient API , Map integration, Clinic Validation, Connect API to Clinic Service, Map endpoint, Finalize service]

Requirement 5

6. As a user I want to be able to select appointment time so that I can fit it into my schedule Related issues: [Create show available timeslots component, Add endpoint for getting available times within a timespan for certain clinics, Get all available times Implement live update for timeslots]

Requirement 6

7. As a dentist I want to be able to check my upcoming appointment so I can schedule my days - Related issues: [Simple dentist UI]
    a. As a patient I want to be able to check my upcoming appointments so that I can schedule my days (TODO REVIEW) - Related issues: [My booking view, Availabletime endpoint]

Requirement 7

8. As a user I want to book an appointment at an specific clinic so that I do not have to go far
9. As a dentist i want to know who my patient is so that I can accurately provide them care - Related issues: [Simple dentist UI]

Requirement 8

10. As a user I want to get notified when clinics of my choice submit a new time slot so that I can book an appointment before someone else takes it

Requirement 9

11. As a user I want to use the system on my phone so that i can book an appointment when im on the fly

Requirement 10

12. As a user I want to be able to cancel appointments so that I attend to important matters without having to pay

Requirement 11

13. As a user I want to know if my appointment is canceled so that i don't waste my time - Related issues: [Dynamic topics, Booking&cancelling support]
14. As a dentist i want patients to know when I cancel their time so that they don't give me a bad review - Related issues: [Simple dentist UI, CRUD for availabletimes]

Requirement 12

15. As a dentist I want to publish time slots so that patients can book available times - Related issues: [Simple dentist UI, Publish time slots]

Requirement 13

16. As a dentist I want to specify time slots myself so that they fit my schedule - Related issues: [Simple dentist UI, Setup structure, Establish connection with Dentist Client]

Requirement 14

17. As a dentist I want to be able to cancel appointments so that my patients don't show up when I'm not available - Related issues: [Simple dentist UI, Dentist cancel time slot]

Requirement 15

18. As a dentist I want to get notified when someone books a time slot so that I can know when to be at the clinic - Related issues: [CRUD for available times, Simple dentist UI, booking&cancelling support,
19. As a user I want the dentist to be notified when I book an appointment so that the dentist knows I'm coming - Related issues: [Setup notification service environment, Implement email functionality, Email template expansion and sending functionality, booking&cancelling support, Implement automated email retry system]

Requirement 16

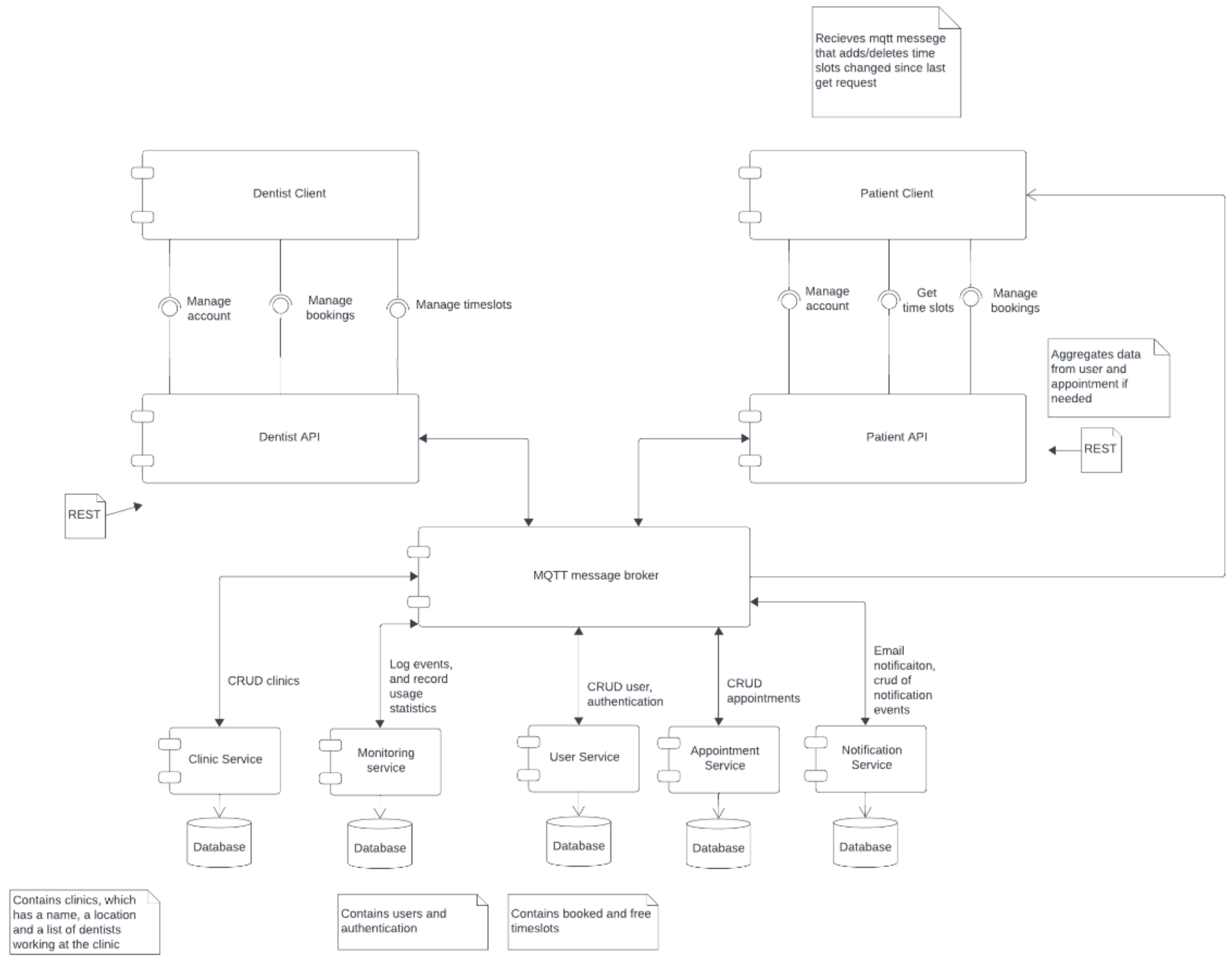
20. As a dentist I want to be notified when my patients cancel so that I can treat someone else - Related issues: [Simple dentist UI, Add get patient email endpoint for booking cancellation notifications]
21. As a dentist I want users to be notified when I publish new appointments so that they book a time at my clinic - Related issues: [Implement emailing lists, Create CRUD endpoints for notification service, Create subscribe to emails page, Enhancement: update subscriber capability]

The key and feature tasks that the system performs:

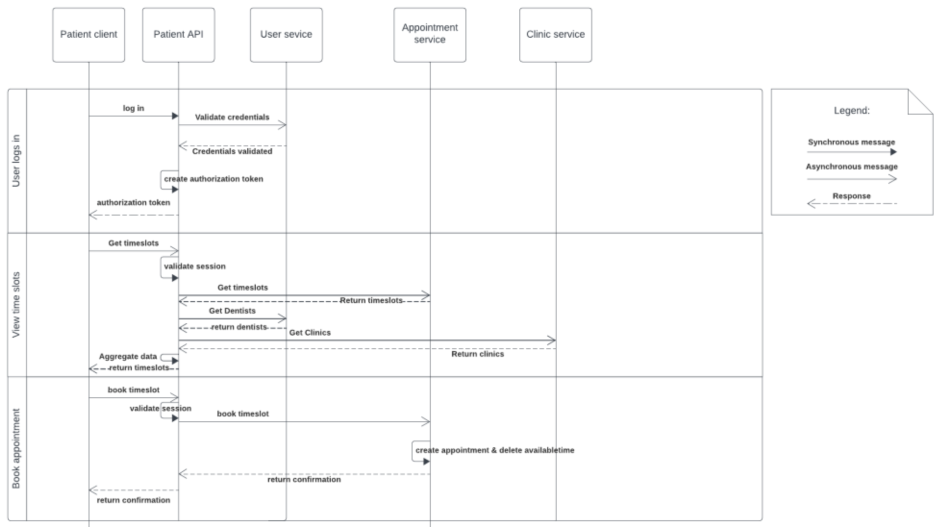
- Authentication
- Map displays clinics worldwide
- Notification is issued when booking an appointment
- Monitor availability of system
- Dentist’s bookings on dentist client
- Available times on patient client
- Register/delete clinic
- Add/remove dentist from clinic
- Load balancer
- Stress tests
- Filtering clinics on the availability of time slots in a certain time span
- Booking appointments
- Cancelling appointments
- Fetching and displaying appointments

Overview of system architecture:

The architecture did not receive any updates during the last sprint. The following figure is the final version of the system and is implemented as such.


Recieves mqtt messege that adds/deletes time slots changed since last get request
Dentist Client
Patient Client
Manage account
Manage bookings
Manage timeslots
Manage account
Get time slots
Manage bookings
Aggregates data from user and appointment if needed
Dentist API
Patient API
REST
REST
MQTT message broker
CRUD clinics
Log events, and record usage statistics
CRUD user, authentication
CRUD appointments
Email notificaiton, crud of notification events
Clinic Service
Monitoring service
User Service
Appointment Service
Notification Service
Database
Database
Database
Database
Database
Contains clinics, which has a name, a location and a list of dentists working at the clinic
Contains users and authentication
Contains booked and free timeslots

A patient uses the system to book an appointment



Software architecture

In brief terms, the overall architecture is a mixture of many architectural styles that communicates using MQTT and REST. The two main architectures incorporated in the system are:

- **Microservices:** When intuitively glancing at the diagram above and seeing the 5 independent services, it appears clearly that this is one of the dominating styles in this system.
- **Publish/Subscribe:** By virtue of the MQTT broker's central role in the system, this style's impact and significance is fundamental.

However, upon a closer observation, two other architectural styles are also present:

- **Layered Style:** In the sense of requests and responses and separation of concerns into distinct tiers, this is a valid style. To be more specific, the following mapping of tiers and high-level layers can be done:
    - Patient Client & Dentist Client = Client
    - Patient API & Dentist API = Server
    - Microservices = Databases
- **Pipe and filter:** Due to the fact that the dataflow resembles a pipe with self-contained filters. The clients constitute the data source and the microservices are similar to filters that inspect the input (request) and produces an output (response)

System Analysis

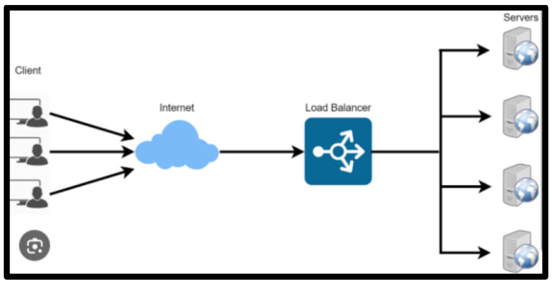
Scalability

- **RequestID:** Patient API and Dentist API generates unique strings using UUID and sends that as an attribute in the payload messages to the specific microservice. The requestId is used as a key that maps to its response from a microservice. Once the microservice returns its response to one of the APIs, the requestID in the payload is examined. If its value matches an existing key in the map it is deleted and the interaction between the two components is successful. In other words, each API capitalizes upon a key-value structure to keep track of the current requests from the APIs. Moreover, a mqtt-timeout that lasts for 10 seconds is set, so that the request automatically is deleted from the map if the time exceeded.
- **User Service & Clinic Service:** The Clinic Service was implemented as an action to make the system more scalable by going beyond a specific city. If we were to go with our previous idea and merge the clinic-functionality in User Service, a multitude of setbacks in developing the system would constrain the maintenance:
    - If we aim to show clinics in not only Gothenburg, the system must listen to the request of clinics and then display accordingly, rather than just showing hardcoded clinics in the nearby area (on the assumption that the system is widely used across the world and that millions of clinics exist). Now, imagine if we merged the functionality of User Service and Clinic Service so that each DB-instance contains clinic name, a list of dentists and a list of users. This would entail that the responses that Patient Client receives contain a vast amount of redundancy. Retrieving the users of the system when only wanting clinics is irrelevant. Not only does this make it harder for the developers when adding new payload attributes in the future, but the additional memory required decreases the performance. Upon observing the current DB-instances of clinics, a few of which has ratings, photoURL and address (more details about these attributes can be found in README:

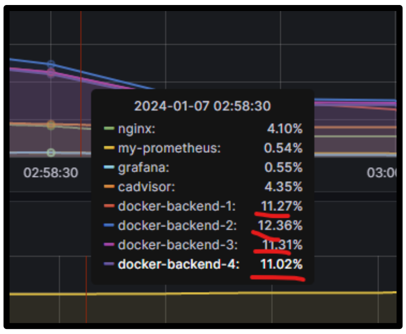
https://git.chalmers.se/courses/dit355/2023/student-teams/dit356-2023-20/group-20-distributed-systems-clinic-service/-/tree/main?ref_type=heads), and playing with the idea of mixing users and their respective passwords, it appears clearly that it's an unmaintainable fashion of architecture.

   - Furthermore, the absence of Clinic Service would have a negative impact on availability. If the fictitious service (Clinic Service + User Service) crashes, a user won't be authenticated, and clinics won't be displayed on the map. This contradicts one of the main benefits that comes with a microservice architecture: No single points of failure

- **Load Balancer:** We implemented a load balancer with the objective of increasing scalability. By distributing requests from the clients across multiple servers, the load on one server decreases. Another benefit of this implementation is that the fault-tolerance escalates in conjunction with the multiplicity of available servers when provided with the fact that adding or subtracting servers occur on demand, it's safe to say that the availability of the system is boosted.



- **Stress tests:** By continuously monitoring the load and its distribution across the different instances of the server, an awareness of a system's capabilities and weaknesses is brought to light. While these types of tests don't have a direct impact on the scalability, they do provide the developers with the necessary information in the realm of optimal future allocations of work forces. In many cases, these tests indicate a strong need for improvements in supporting larger quantities of clients. It is analogous with the purpose of communicating in a team. While the ultimate goal with a project is to produce tangible results, the indirect goal is that the developers communicate so that they get the information on how to coordinate their efforts to ultimately bring the final product into physical existence. As of right now, the stress test discloses a manageable and evenly distributed load, which serves as a green light for robustness and prospective expansions of user bases without experiencing a drop in performance.



```
2024-01-07 02:58:30
nginx:               4.10%
my-prometheus:       0.54%
grafana:             0.55%
cadvisor:            4.35%
docker-backend-1:   11.27%
docker-backend-2:   12.36%
docker-backend-3:   11.31%
docker-backend-4:   11.02%
```

Fault-tolerance

- **Status codes:** In distributed systems where multiple services are coordinating tasks, the importance of each autonomous component's awareness of the states of its peers is of highest priority. Therefore, an explicitly stated agreement was made. Each payload response from the microservices must contain a 'status' attribute associated with an Integer. The APIs check the status codes to determine whether the request was successfully carried out or not and delete the corresponding request from the map.
- **Try & Catch:** When the requests to a server that is down aren't running for whatever reason, the other services are still able to run. Considering our choice of architecture, it is of the highest importance to make every microservice autonomous. Although dependencies of communications between entities is inevitable in a system, the services must still be able to execute independently. In turn, if one service is facing downtime, the other ones should not be negatively affected. Ideally, the user experience is unaffected if a service is deviating from specified behavior. However, that's not possible when the services coordinate and collaborate on tasks. To counteract this problem, the group used try & catch in multiple instances:
    - **Payload Input:** Format – Example: Expected position: "lat,lng" --> The associated service doesn't crash if "test" is inputted
    - **Alert Window:** If an exception in Patient Client is thrown, an alert window pops up that informs the user that something went wrong.

- If the user clicks on the 'OK' button so that the window disappears, what remains of the UI is the components that are loaded with their respective response from the requested microservice. A concrete example of this is the map: If the Clinic Service isn't running, the map-area turns white without interrupting the runtime flow

- **Isolating related data:** Grouping data that is related --> Microservice architecture
    - **Appointment Service & Notification Service:** In the beginning of the project, the group's conceptual idea was to combine both of these services into a bigger one that would be responsible for both aspects. Dealing with appointments and notifications are two different categories that can be separated. Through the lens of not only fault-tolerance but scalability and maintainability, self-contained entities with concrete areas of responsibility are imperative. In this particular example, merging these two services into one would bring the benefit of a quicker development of functionality. It would be significantly easier to develop the system if we got rid of this service, since we wouldn't have to think about connecting two of them to the other components. Nevertheless, the negative side effects would appear in the future when it's time to add new features.
    - **Patient API & Dentist API:** What we could have done is to have a direct connection between the clients and the MQTT broker by neglecting the existence of these APIs. However, the scalability of the system would face a negative downturn. Due to the fact that the APIs serve as an intermediary between the clients and the broker, they support extensions of different types of clients. The separation of concerns where the code in the clients is the UI representation of the system, whereas the APIs have the responsibility of receiving requests and sending back responses to the client-side, enables the developers to only focus on the distinct features that the to-be implemented client will have. In the future, we may, for instance, want to add more components from respective APIs in the following way:
        - Patient API --> Guest Client
        - Patient API --> Admin Client

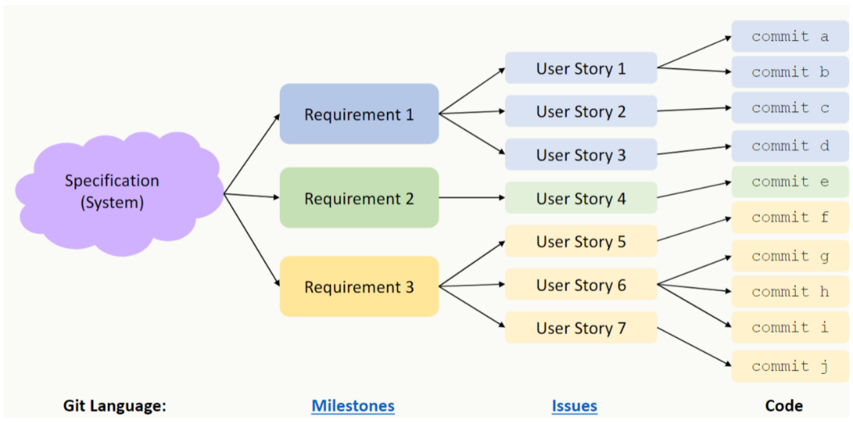
- Dentist API --> Dental Clinic Owner

As a result of wanting to expand the roles of accessibility for clients, we could create client components from the APIs to leverage on their benefits of scalability. The next question that arises is whether or not it's redundant to create entirely new components to support these features. Is it worth the hassle or do we just declare more variables on the risk of the code becoming bloated and unmaintainable? This is where architectural rationale comes into play. Questions such as "How will this change affect the system and our position in adding new features in the future?", "To what extent will this change constrain us or the stakeholders?"

Brief description of development process

Traceability

The system is defined by requirements that in turn are the building blocks of user stories in which the code is linked to.



Beyond the traceability, the group focused on three main things each week:

- Feature branching: An issue that contains commits must be linked to a branch created as a result of an implementation of a feature
- Meetings: Review and retrospective meetings occurred on a weekly basis
- Sprints: Each milestone was divided into two sprints (except the milestone over Christmas – it had one sprint), making a sprint equivalent to one week. The conceptual idea of implementing sprints in the development process is based on the idea of decomposing bigger tasks into sub-tasks with independent goals into a smaller time frame as a way of preserving the objective of incremental delivery: Continuous delivery of tangible progress that allows the reviewer to give indicative feedback

For more information about the development process, check the wiki page: https://git.chalmers.se/courses/dit355/2023/student-teams/dit356-2023-20/group-20-distributed-systems/-/wikis/Home/Development-process

Outcomes of the retrospective

As mentioned in the section above, the group consistently performed retrospective meetings that focused on what could have been done better and what has worked well that can be brought into the next sprint. Overall, the general learning lessons are the following:

From the start of the project, put more emphasis on...

- <u>Establishing a clear payload structure:</u> Discuss it from an architectural point of view and identify necessary attributes and states of components that need to be communicated between the entities
- <u>The architectural aspect of the system:</u> It plays a vital role in the capabilities of the system from every aspect. Scalability, maintainability, performance and availability are four qualities of any software that arguably take precedence over any other quality in regard to user experience, and all of these share a direct dependency on the laid-out architecture. Neglecting an architectural rationale can mark the difference between a successful and failed project. If not done properly, the structural communication between the components will constrain the development of the system. Conclusively, asking the question "How will this architectural change prevent the developers from adding more features in the future?" is a good approach when deciding on communications between components.
- Emphasizing the effort required to integrate multiple 'independent' moving parts – both in terms of coordinating with team members and the actual workload it can entail.
- The importance of constant and clear communication between team members. Making sure all team members are heading in the same direction is of utmost importance in a project of this scope, since it can easily be the case that everyone agrees on something verbally or over text but in reality, we carry our own biases and end up with different ideas of what has been agreed upon, what something means etc.

For more information about the group's meetings and learning lessons, check this wiki page: https://git.chalmers.se/courses/dit355/2023/student-teams/dit356-2023-20/group-20-distributed-systems/-/wikis/Home/Sprints