Software Architecture Document

Version 1.4

for

UberSanté

Prepared by

Name	ID
Kevin Luu *(Leader)*	40037514
Tamar Merdkhanian	40030718
Michael Tang	40028150
Tristan Vu	40028927
Wu Wen Tang	40028075
Mathew Jackson	27709315
Zhen Yee	40028478

Document history

Date	Version	Description
20/02/2019	1.0.0	Add template
12/03/2019	1.1.0	Add text skeleton
13/03/2019	1.2.0	Add views diagram
13/03/2019	1.3.0	Finalize documents
11/04/2019	1.4.0	Update class diagram

Table of Contents

1. Introduction **5**
1.1 Purpose 5
1.2 Scope 5
1.3 Definitions, Acronyms, and Abbreviations 5
1.4 References 6
1.5 Overview 6

2. Architectural Representation **7**
Figure 1 - Architectural representation 7
2.1 Logical View 8
2.2 Process View 8
2.3 Data View 8

3. Architectural requirements: Goals & Constraints **9**
3.1 Functional requirements 9
3.2 Non-functional requirements 10

4. Use case view (Scenarios) **11**
Figure 2 - Use case view representation 11

5. Logical view **12**
5.1 Layer 12
Figure 3 - System Architecture Represented by 3 layers 13
5.2 - Subsystems 13
5.2.1 - Controller 13
5.2.2 - Exceptions 13
5.2.3 - Models 14
5.2.4 - Services 14
5.2.5 - DTO 15
5.2.6 - Repositories 15
5.3 Communication Diagram 16
5.3.1 Book an appointment 16
5.3.2 Update an appointment 16
5.3.3.Cancel an appointment 17

6. Process view **19**

7. Data view **22**

8. Quality **23**
8.1 - Scalability 23

8.2 - Reliability and Availability 23
8.3 - Portability 23

9. Deployment View 25

1. Introduction

1.1 Purpose

The content contained within this documented is targeted to the developers of the the system and other stakeholders versed in software architecture. It's purpose is to provide a detailed view of how the system is setup and help visualize how all the pieces fit together.

1.2 Scope

The architecture used for this system is a layered architecture. This web application involves the implementation of a set of complex architectural views. Each of these views are described by class diagrams, OCL expressions and communication diagrams. This document influences the design and the architecture used in the implementation phase. Modifying use cases will change the domain model, the class diagram and the implementation of the system.

1.3 Definitions, Acronyms, and Abbreviations

Term	**Definition**
SAD	Software Architecture Document (this document)
SRS	Software Requirements Specification
UML	Unified Modeling Language
User	Any person using the system, be it admin or patron
Admin	User possessing additional privileges over that of a patron
Patron	User who will have access to the basic functions of the system
UC	Use Case
RUP	Rational Unified Process

1.4 References

[1] The “4+1” view model of software architecture, Philippe Kruchten, November 1995, http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/Pbk4p1.pdf

1.5 Overview

The SAD is broken down into seven sections. Section 2 presents the modelling of the 4 architectural representations used for our system. This includes the logical view, physical view, process view and the data view. Section 3 covers the goals and constraints of the architecture through specifications in formal logic. Sections 4 through 7 goes into further detail of the different views of the system, showing its class diagrams, the OCL expressions involved and its communication diagrams. Section 8 details the quality of the system by providing details on the communication between the frontend and backend.

2. Architectural Representation

The architecture we are using for our system is a layered architecture. There are three layers presentation, business and data access. The data access layer also known as persistence layer of the system deals with connection to the database and . The presentation layer of the system will never be able to access the data layer without going through the business layer.

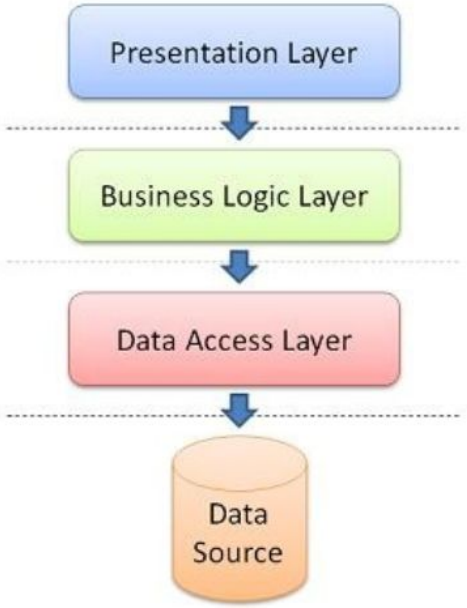


Figure 1 - Architectural representation

Usage of data mapper, services, repository and unit of work can be used to describe the layer more in-depth.

This document details the architecture using a modified interpretation of the views defined in the “4+1” model[1]. The views used to document <Uber Sante> are:

2.1 Logical View

- Audience: Designers

- Area: related to functionality that system provides to end-users
- Related Artifacts: Class diagram, and interaction diagrams (communication diagrams, or sequence diagrams).

2.2 Process View

- Audience: Integrators
- Area: Related to the interactions
- Related Artifacts: Activity Diagram

2.3 Data View

- Audience: Data specialists, Database administrators
- Area: Persistence: describes the architecturally significant persistent elements in the data model.
- Related Artifacts: Relational Data Model

3. Architectural requirements: Goals & Constraints

3.1 Functional requirements

The use cases and requirement references below are ones that are architecturally relevant to the application and their important helped come up with the architecture of the project.

Source	ID	Name	Architectural relevance	Addressed in
SRS	UC1 (FR8)	**Patient Book Appointment**	Patient booking allowed the development of the booking system architecture, i.e. what type of information is associated with an Appointment (patient, doctor ID, start time, end time, room, type of appointment) and how it's connected to other components such as Patient and Doctor.	Section 3.2
SRS	UC4 (FR13, FR14, FR15, FR16)	**Doctor Update Availability**	Doctor availabilities allowed the development of the availability system architecture, i.e. what type of information is associated with an Availability (doctor ID, start time, end time, availability type), how it's connected to other components such as Doctor and how part of this information is available to patients.	Section 3.2
SRS	FR1	**Patient Registration**	Patient registration made it obvious that a Patient type had to implemented for the architecture with the various patient attributes associated to each one (name, last name, health card, etc).	Section 3.1

SRS	FR2	**User Login (Patient, Doctor, Nurse)**	Similar to Patient registration, this made it obvious that a Patient, Doctor and Nurse type had to be implemented with their respective attributes.	Section 3.1
SRS	FR6	**Medical Rooms for Clinic**	Medical rooms for clinics revealed that Clinic should be a Singleton in the architecture since there are attributes associated to it such as medical rooms.	Section 3.1
SRS	FR7	**Number of Doctors for Clinic**	Similar to medical rooms, number of doctors associated to clinics revealed that Clinic should be a Singleton in the architecture.	Section 3.1
SRS	FR13	**Cart System**	In order to implement a cart system, a Cart had to be implemented in the architecture as well associated with each patient.	Section 3.1

3.2 Non-functional requirements

The non-functional requirement references below are ones that are architecturally relevant to the application and their important helped come up with the architecture of the project.

Source	ID	Name	Architectural relevance	Addressed in
SRS	NFR2	**Payment method**	Since credit card is the only acceptable payment method, an adapter design pattern should be implemented.	Section 3.3
SRS	NFR3	**Annual checkup limit**	Each patient can only book one annual check up in a year and this information must be associated with each patient.	Section 3.3

4. Use case view (Scenarios)

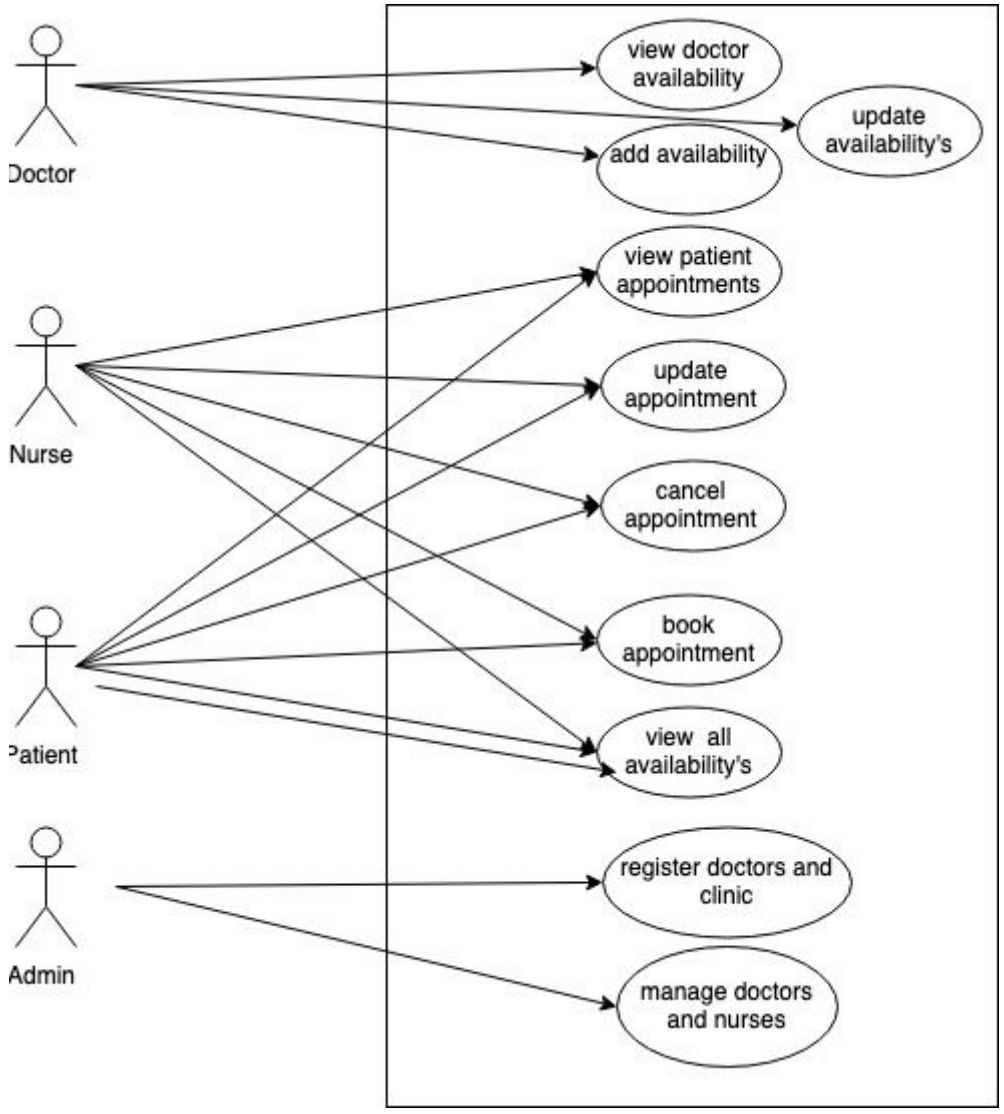


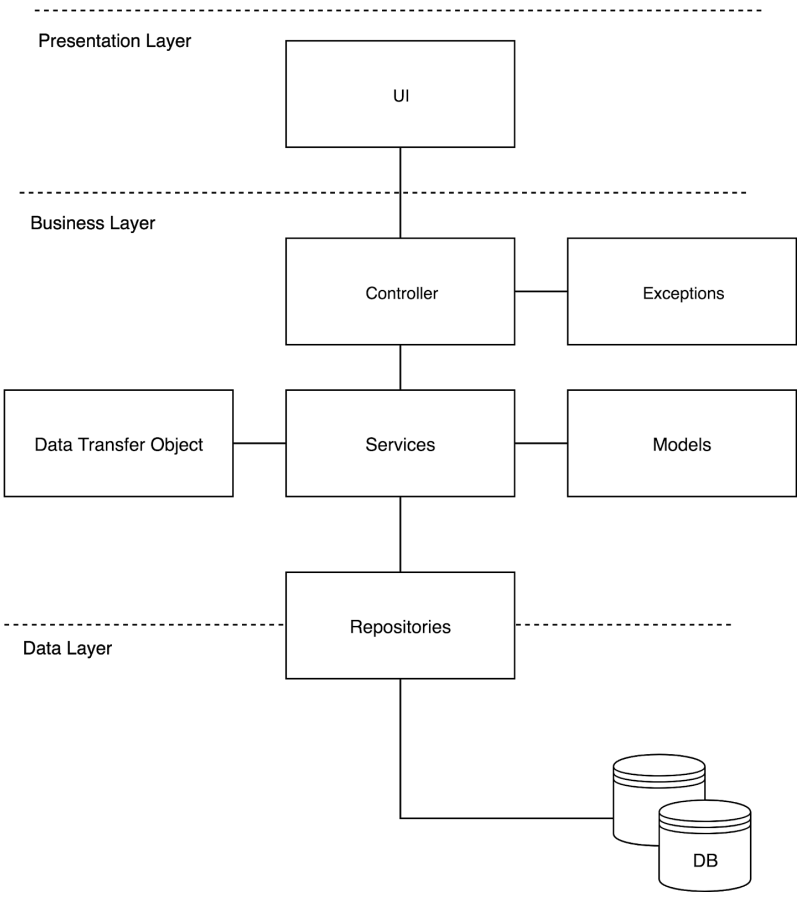
Figure 2 - Use case view representation

5. Logical view

5.1 Layer

The architecture implemented in UberSanté as mentioned previously is the layer architecture. It uses and is separated into 3 layers: the presentation layer, the business layer and the data layer.

In the presentation layer, the frontend is composed of everything that the user should see. The presentation layer is just a representation of the UI that is written in typescript using Angular 6 as the framework. The business layer contains all the logic that the application uses. From the business layer, the controller handles every request sent from the UI. Controller then makes an appropriate call to its service. Each service contains the necessary logic to build objects from the models, transform an object to a data transfer object (DTO) and calling a repository to make a query to the database. The data layer is the where the datas reside in. A

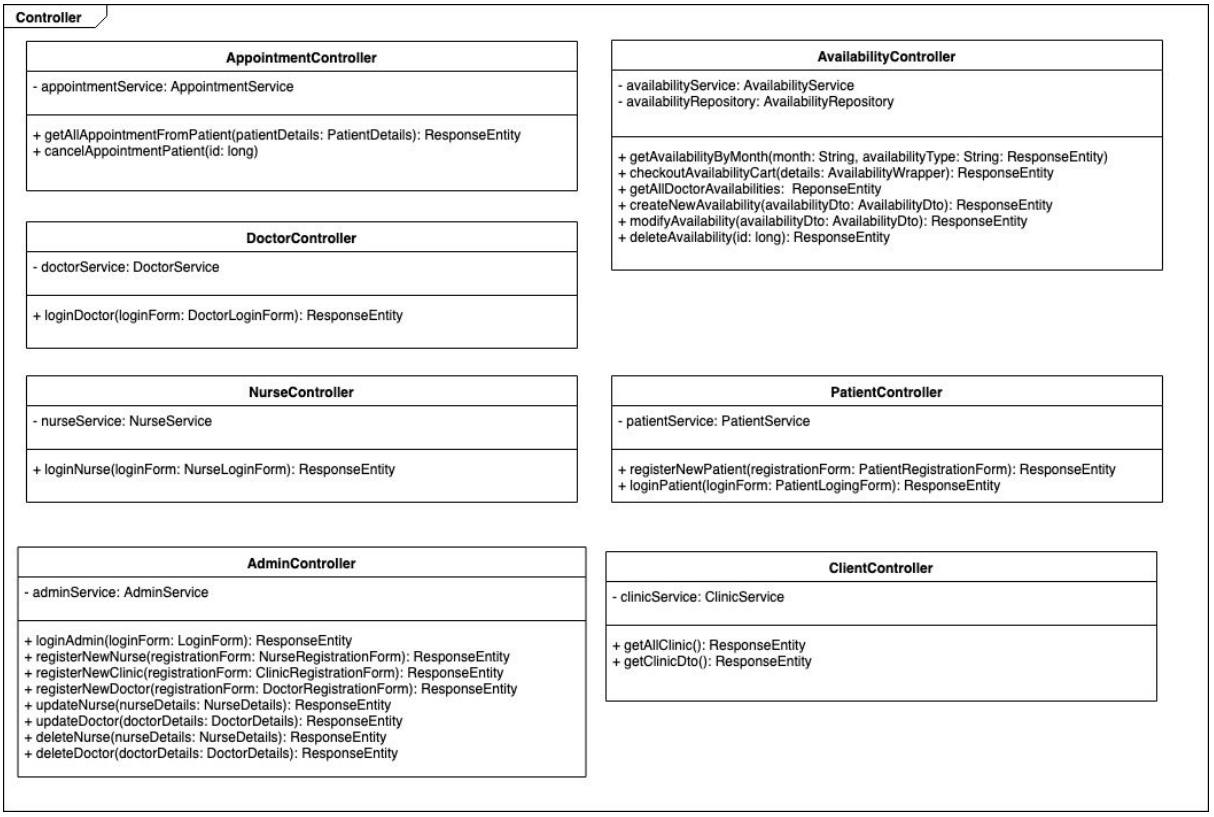


repository makes a create, read, update and delete (CRUD) function to the database.

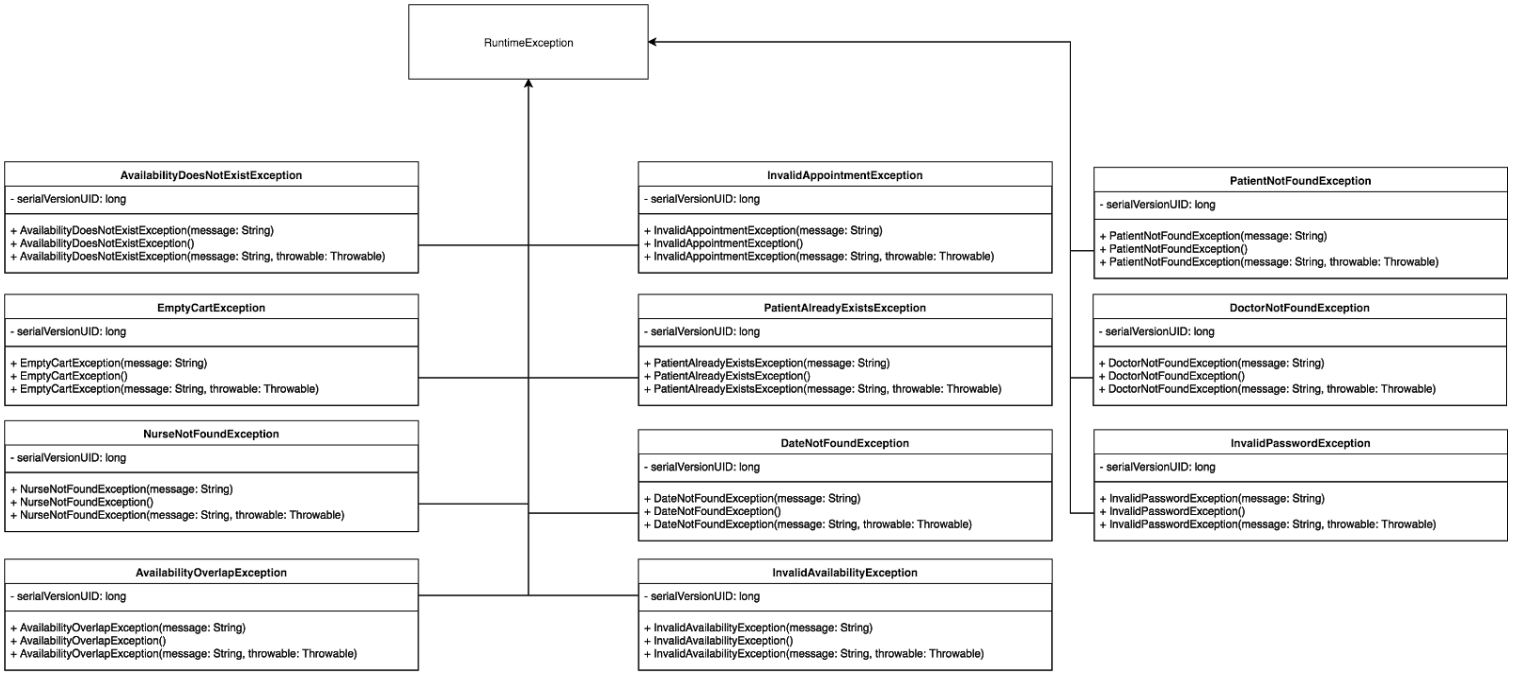
Figure 3 - System Architecture Represented by 3 layers

5.2 - Subsystems

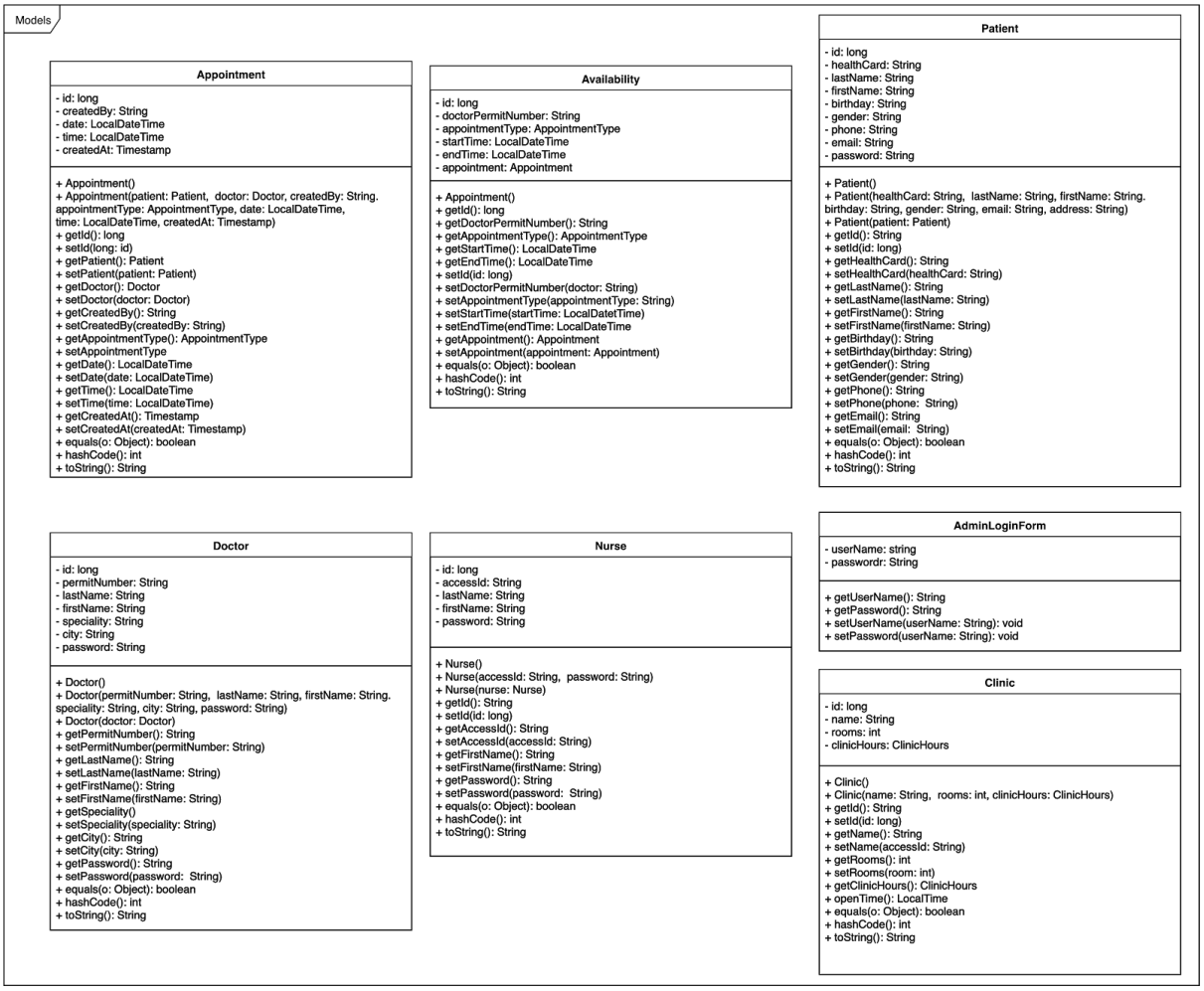
5.2.1 - Controller



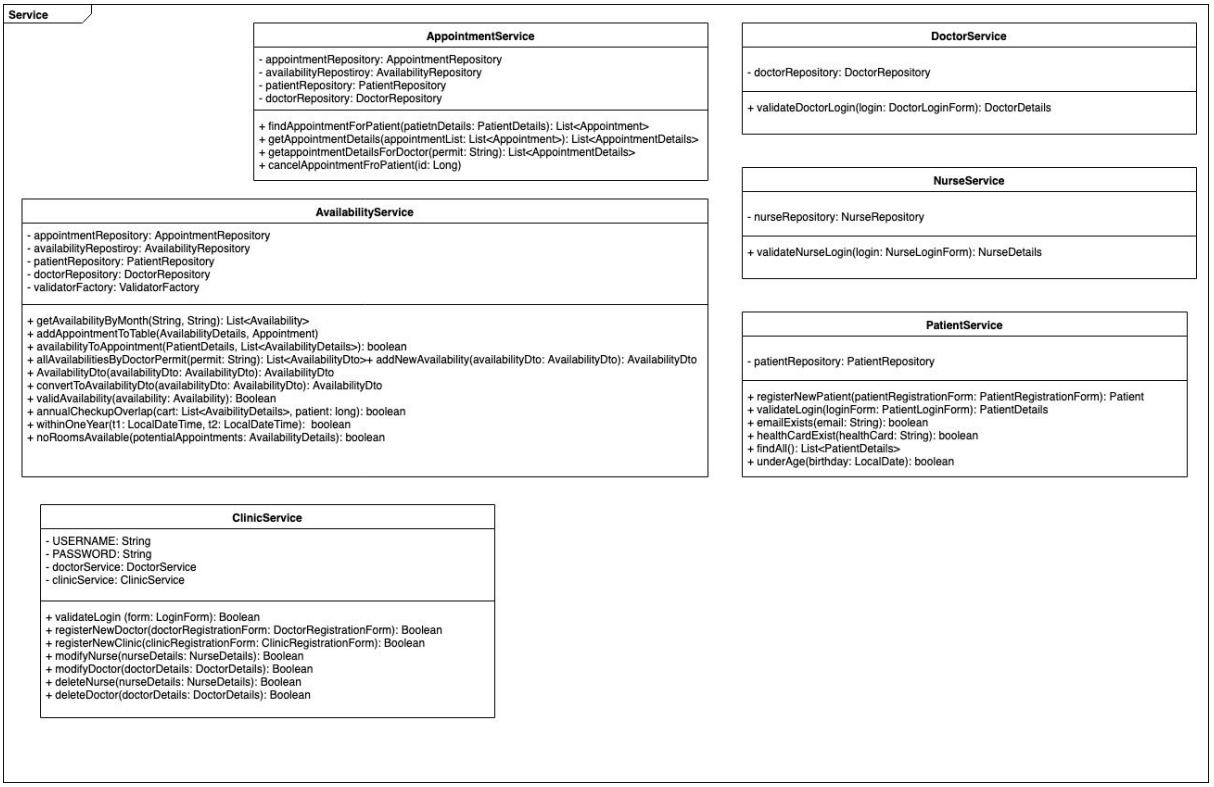
5.2.2 - Exceptions



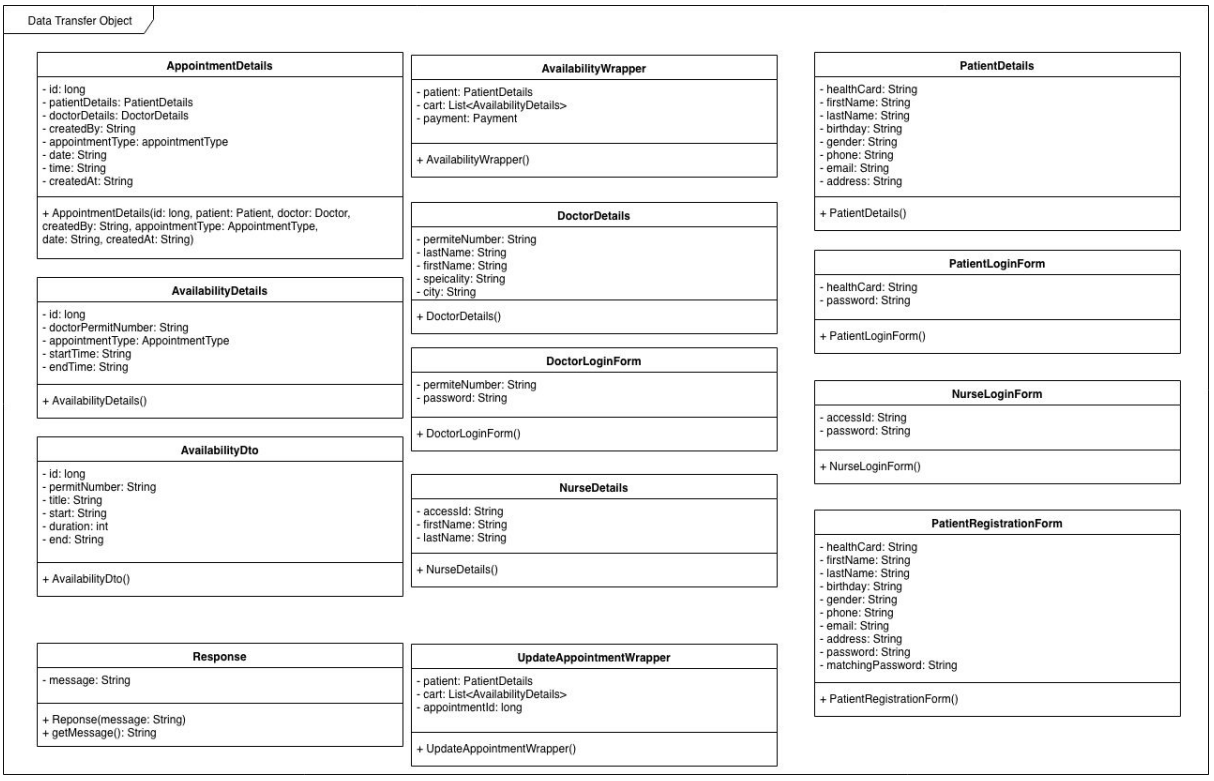
5.2.3 - Models


Models
Appointment
- id: long
- createdBy: String
- date: LocalDateTime
- time: LocalDateTime
- createdAt: Timestamp
+ Appointment()
+ Appointment(patient: Patient, doctor: Doctor, createdBy: String. appointmentType: AppointmentType, date: LocalDateTime, time: LocalDateTime, createdAt: Timestamp)
+ getId(): long
+ setId(long: id)
+ getPatient(): Patient
+ setPatient(patient: Patient)
+ getDoctor(): Doctor
+ setDoctor(doctor: Doctor)
+ getCreatedBy(): String
+ setCreatedBy(createdBy: String)
+ getAppointmentType(): AppointmentType
+ setAppointmentType
+ getDate(): LocalDateTime
+ setDate(date: LocalDateTime)
+ getTime(): LocalDateTime
+ setTime(time: LocalDateTime)
+ getCreatedAt(): Timestamp
+ setCreatedAt(createdAt: Timestamp)
+ equals(o: Object): boolean
+ hashCode(): int
+ toString(): String
Availability
- id: long
- doctorPermitNumber: String
- appointmentType: AppointmentType
- startTime: LocalDateTime
- endTime: LocalDateTime
- appointment: Appointment
+ Appointment()
+ getId(): long
+ getDoctorPermitNumber(): String
+ getAppointmentType(): AppointmentType
+ getStartTime(): LocalDateTime
+ getEndTime(): LocalDateTime
+ setId(id: long)
+ setDoctorPermitNumber(doctor: String)
+ setAppointmentType(appointmentType: String)
+ setStartTime(startTime: LocalDatetTime)
+ setEndTime(endTime: LocalDateTime
+ getAppointment(): Appointment
+ setAppointment(appointment: Appointment)
+ equals(o: Object): boolean
+ hashCode(): int
+ toString(): String
Patient
- id: long
- healthCard: String
- lastName: String
- firstName: String
- birthday: String
- gender: String
- phone: String
- email: String
- password: String
+ Patient()
+ Patient(healthCard: String, lastName: String, firstName: String. birthday: String, gender: String, email: String, address: String)
+ Patient(patient: Patient)
+ getId(): String
+ setId(id: long)
+ getHealthCard(): String
+ setHealthCard(healthCard: String)
+ getLastName(): String
+ setLastName(lastName: String)
+ getFirstName(): String
+ setFirstName(firstName: String)
+ getBirthday(): String
+ setBirthday(birthday: String)
+ getGender(): String
+ setGender(gender: String)
+ getPhone(): String
+ setPhone(phone: String)
+ getEmail(): String
+ setEmail(email: String)
+ equals(o: Object): boolean
+ hashCode(): int
+ toString(): String
Doctor
- id: long
- permitNumber: String
- lastName: String
- firstName: String
- speciality: String
- city: String
- password: String
+ Doctor()
+ Doctor(permitNumber: String, lastName: String, firstName: String. speciality: String, city: String, password: String)
+ Doctor(doctor: Doctor)
+ getPermitNumber(): String
+ setPermitNumber(permitNumber: String)
+ getLastName(): String
+ setLastName(lastName: String)
+ getFirstName(): String
+ setFirstName(firstName: String)
+ getSpeciality()
+ setSpeciality(speciality: String)
+ getCity(): String
+ setCity(city: String)
+ getPassword(): String
+ setPassword(password: String)
+ equals(o: Object): boolean
+ hashCode(): int
+ toString(): String
Nurse
- id: long
- accessId: String
- lastName: String
- firstName: String
- password: String
+ Nurse()
+ Nurse(accessId: String, password: String)
+ Nurse(nurse: Nurse)
+ getId(): String
+ setId(id: long)
+ getAccessId(): String
+ setAccessId(accessId: String)
+ getFirstName(): String
+ setFirstName(firstName: String)
+ getPassword(): String
+ setPassword(password: String)
+ equals(o: Object): boolean
+ hashCode(): int
+ toString(): String
AdminLoginForm
- userName: string
- passwordr: String
+ getUserName(): String
+ getPassword(): String
+ setUserName(userName: String): void
+ setPassword(userName: String): void
Clinic
- id: long
- name: String
- rooms: int
- clinicHours: ClinicHours
+ Clinic()
+ Clinic(name: String, rooms: int, clinicHours: ClinicHours)
+ getId(): String
+ setId(id: long)
+ getName(): String
+ setName(accessId: String)
+ getRooms(): int
+ setRooms(room: int)
+ getClinicHours(): ClinicHours
+ openTime(): LocalTime
+ equals(o: Object): boolean
+ hashCode(): int
+ toString(): String

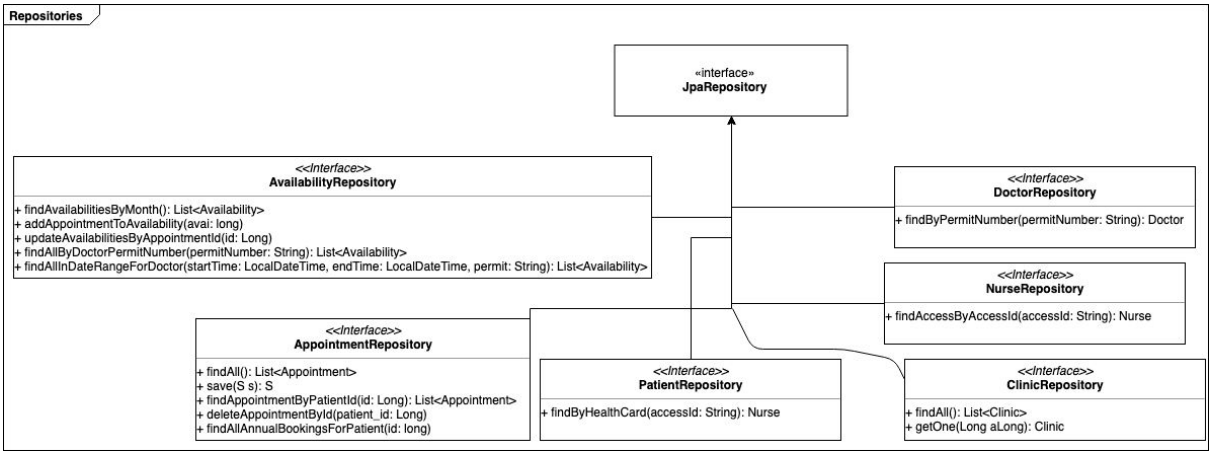
5.2.4 - Services



5.2.5 - DTO

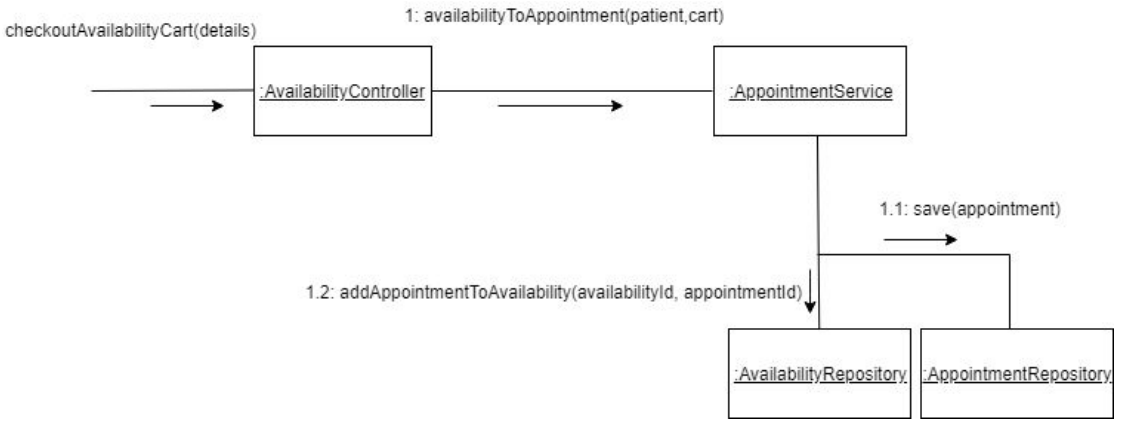


5.2.6 - Repositories



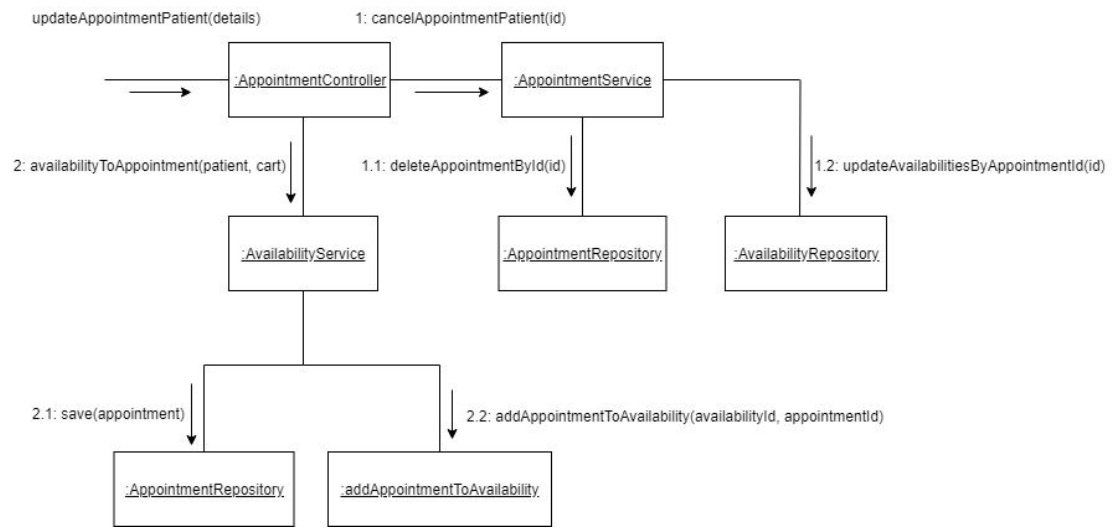
5.3 Communication Diagram

5.3.1 Book an appointment

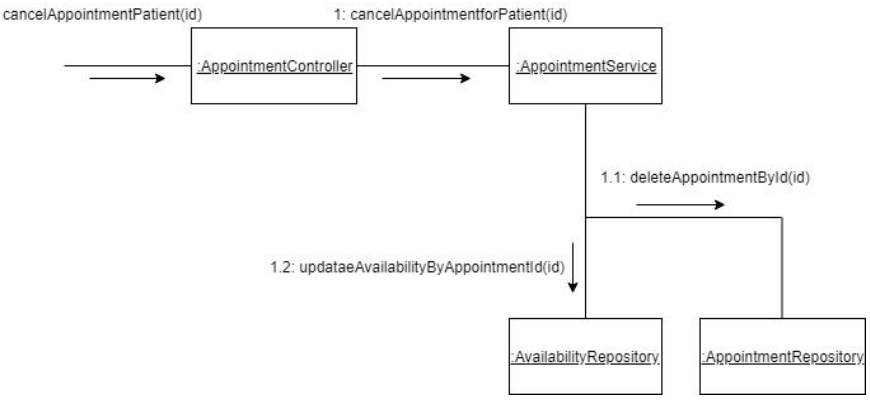
ID	**CD1**
Contract C1	checkoutAvalabilityCart
Related SRS artifacts	UC1, SSD1
Diagram	

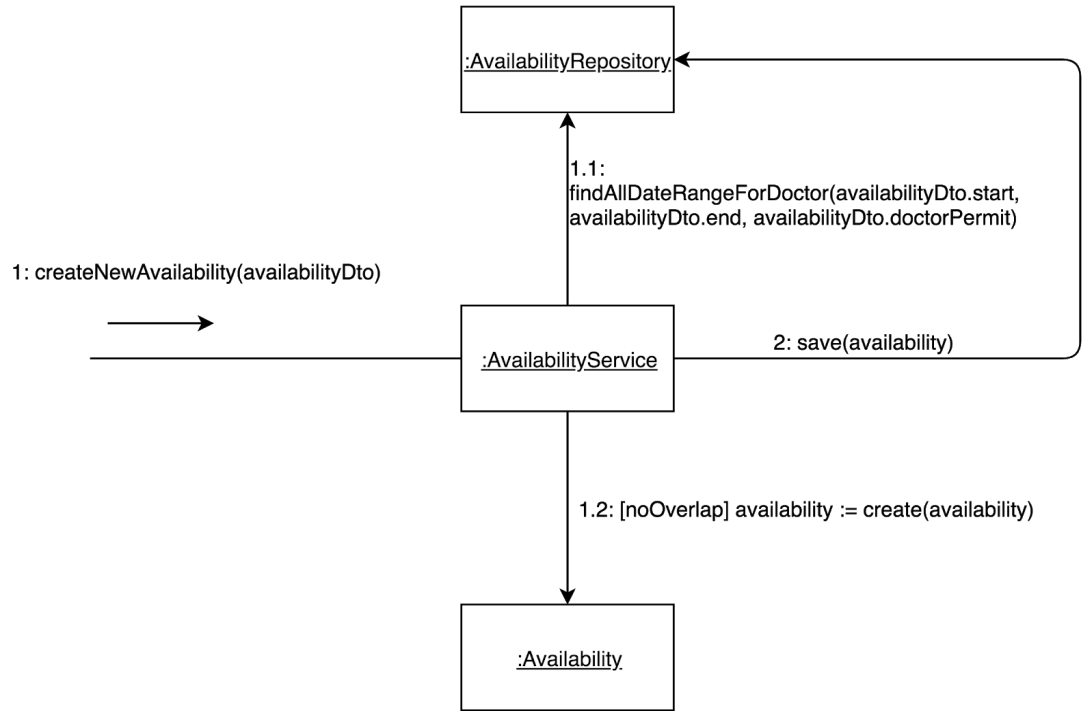
5.3.2 Update an appointment

ID	**CD2**
Contract C1	updateAppointmentPatient
Related SRS artifacts	UC2, SSD2

Diagram	

5.3.3.Cancel an appointment

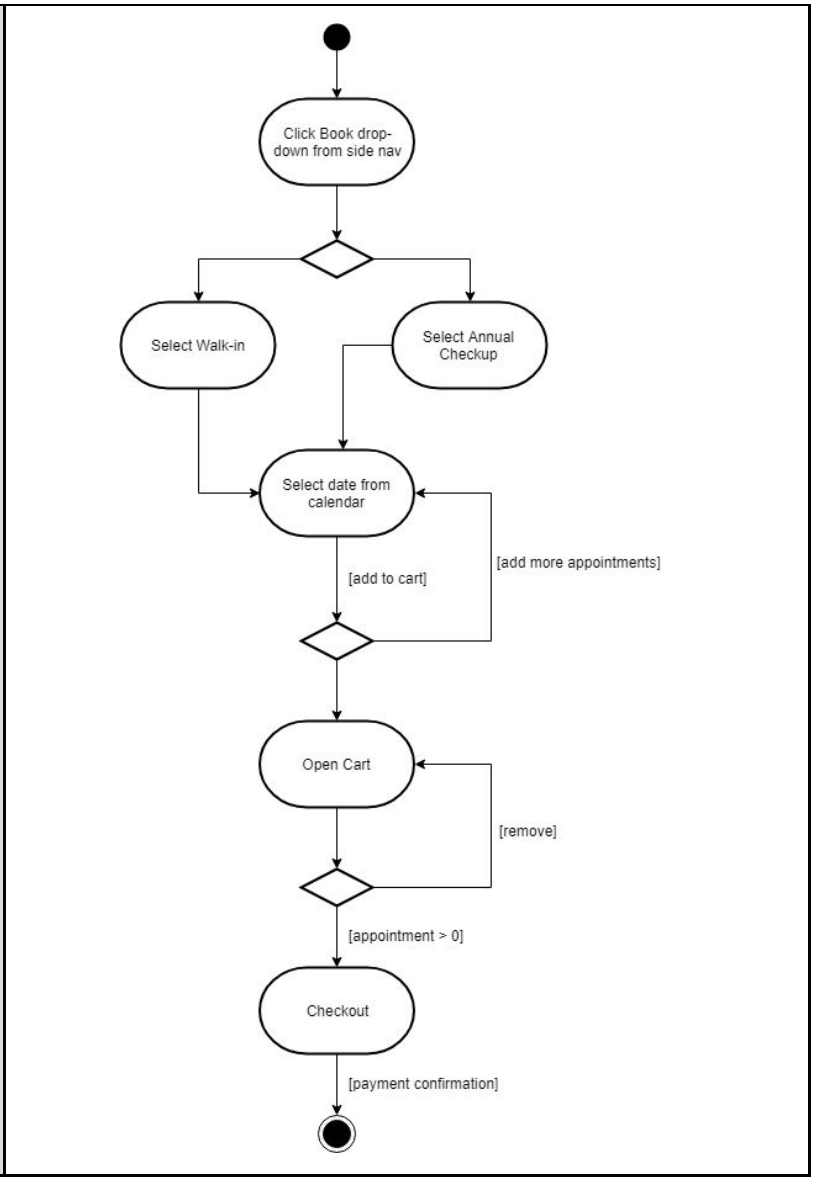
ID	**CD3**
Contract C1	cancelAppointmentPatient
Related SRS artifacts	UC3, SSD3
Diagram	

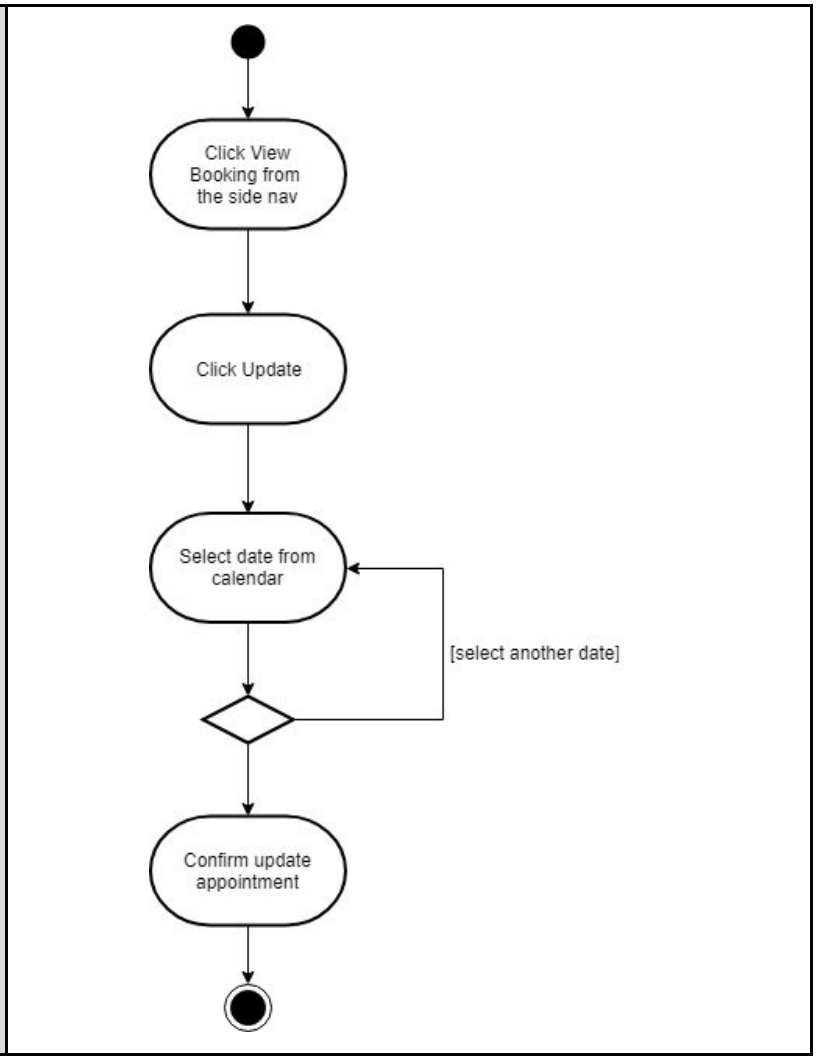
ID	**CD4**
Contract C6	createNewAvailabilitiy
Related SRS artifacts	UC4, SSD4
Diagram	

6. Process view

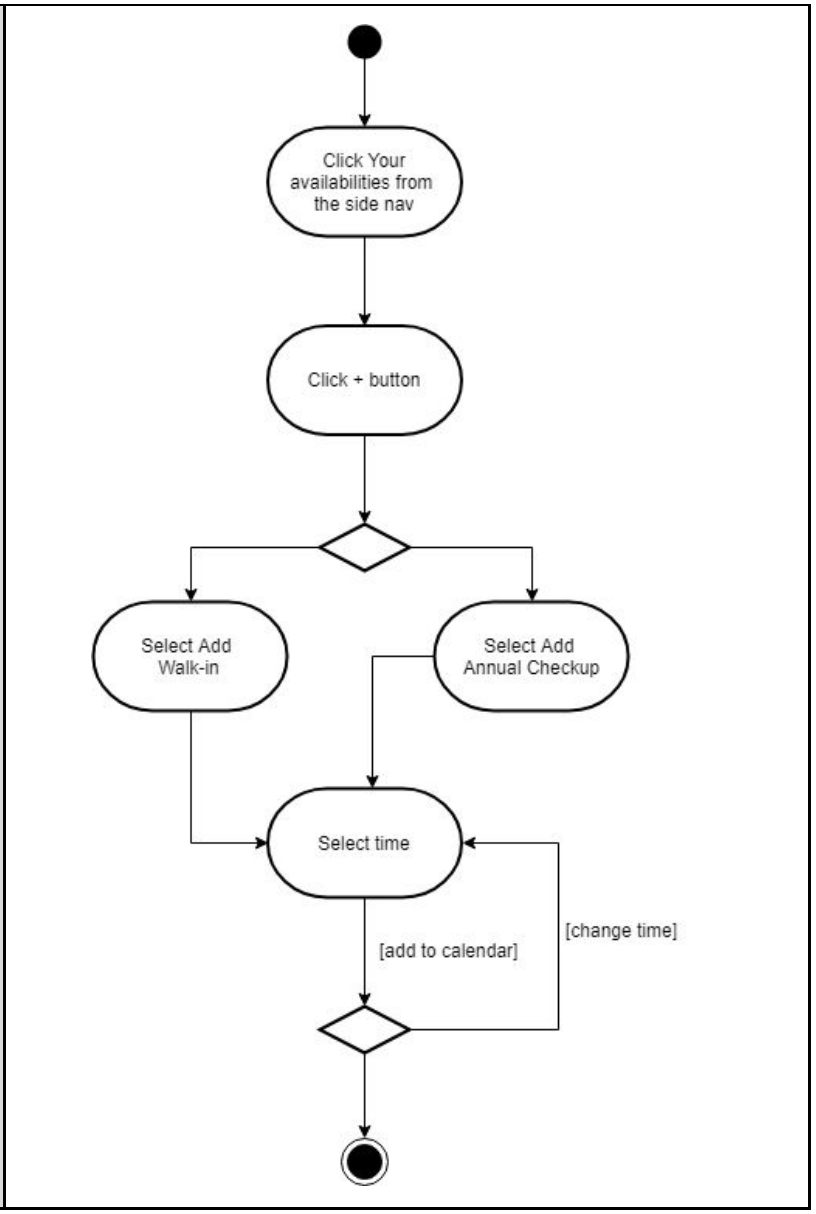
The process view deals with the system processes and how they communicate, focusing on the dynamic behavior of the system. This view illustrates parallelism and concurrency. Activity diagrams are used to capture this view.

UML Activity Diagrams

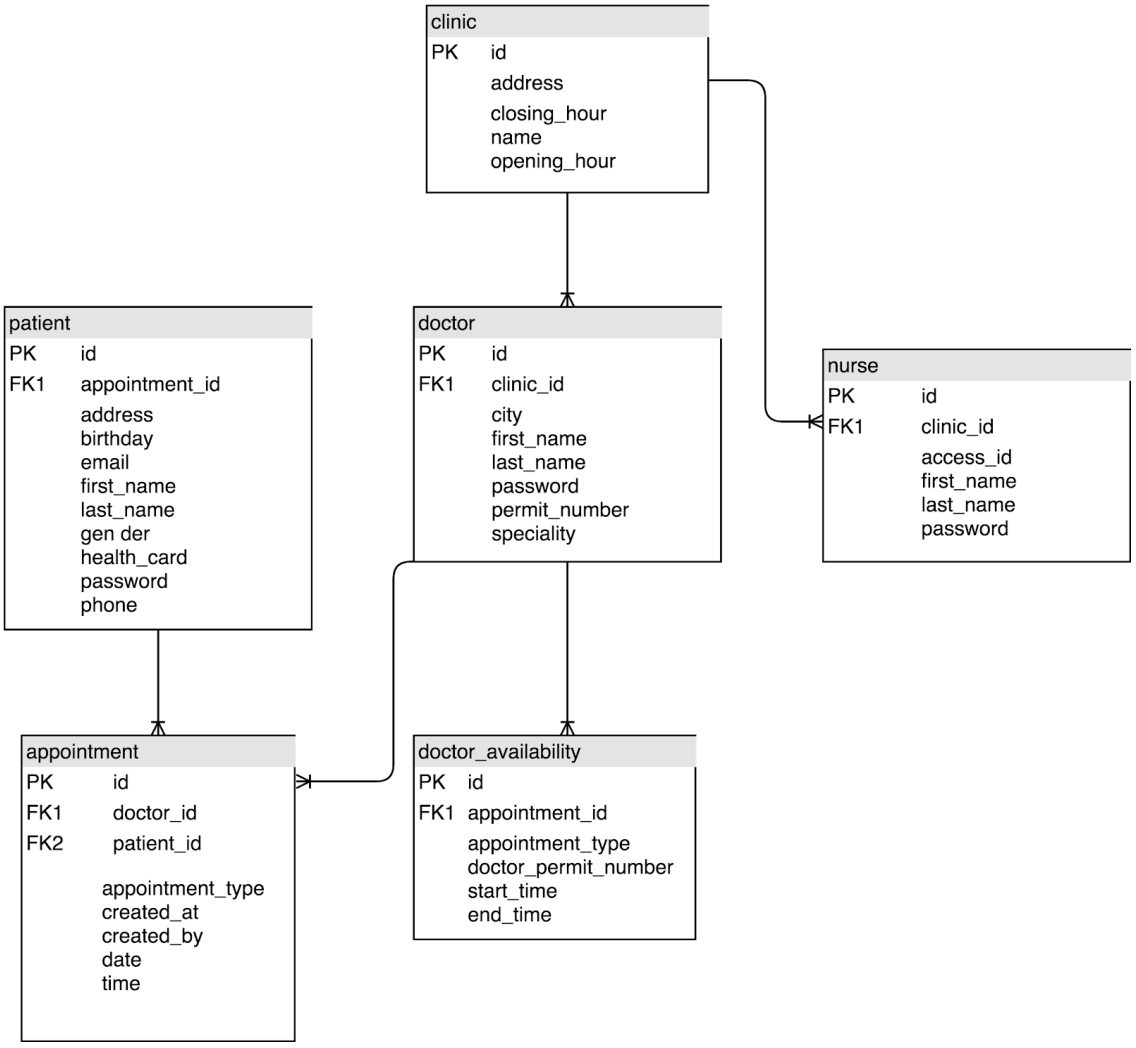
Booking	

Updating	

Click View Booking from the side nav
Click Update
Select date from calendar
[select another date]
Confirm update appointment

Doctor Adding Availability	

7. Data view


clinic
PK id
address
closing_hour
name
opening_hour
patient
PK id
FK1 appointment_id
address
birthday
email
first_name
last_name
gen der
health_card
password
phone
doctor
PK id
FK1 clinic_id
city
first_name
last_name
password
permit_number
speciality
nurse
PK id
FK1 clinic_id
access_id
first_name
last_name
password
appointment
PK id
FK1 doctor_id
FK2 patient_id
appointment_type
created_at
created_by
date
time
doctor_availability
PK id
FK1 appointment_id
appointment_type
doctor_permit_number
start_time
end_time

8. Quality

8.1 - Scalability

By using a three layer architecture and by using a data mapper with service architecture in the business layer, the scalability of the software will be optimal since we separate the domain logic from the database. Additionally, we handle multiple objects within their own class and repository which will not conflict if we were to add additional objects into the system.

8.2 - Reliability and Availability

We handle exception through try catch block if an exception is thrown, a display message is thrown to the user. Database is run on the cloud with amazon which makes it more secure and available at all time.

8.3 - Portability

UberSanté can be started in the Windows, Mac, or Linux environment when all technology used are installed properly. The frontend components are split into separate individual components each having their unique functions and usage. The components can be reused throughout different component such as nesting different components together.
Backend is split into its own class to be reused elsewhere.

A description of how the software architecture contributes to the quality attributes of the system as described in the ISO-9126 (I) standard. **For example**: The following quality goals have been identified:

Scalability:

- Description : System’s reaction when user demands increase
- Solution : J2EE application servers support several workload management techniques

Reliability, Availability:

- Description : Transparent failover mechanism, mean-time-between-failure
- Solution : : J2EE application server supports load balancing through clusters

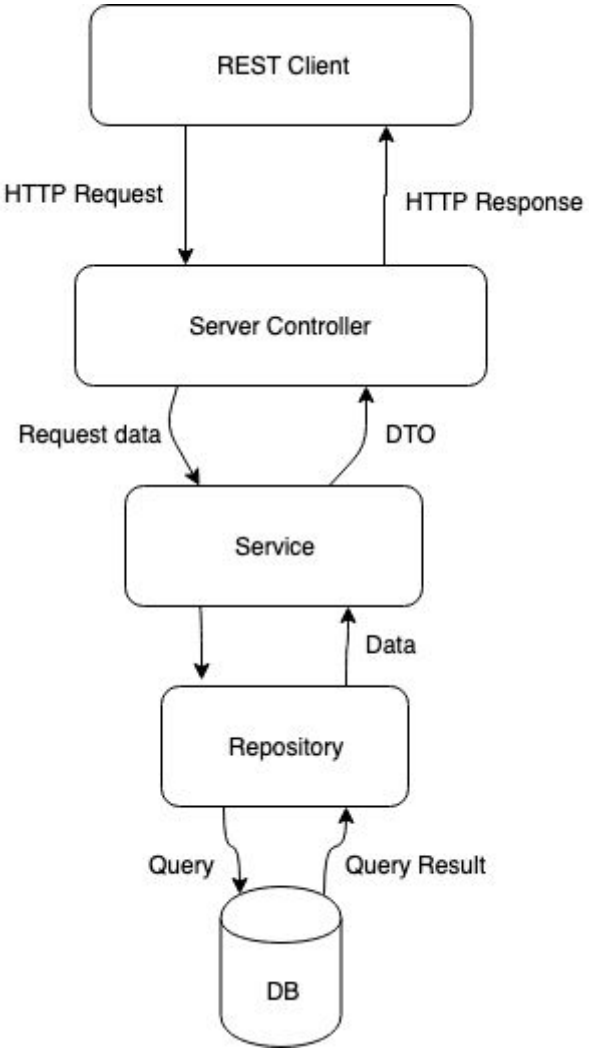
Portability:

- Description : Ability to be reused in another environment
- Solution : The system me be fully J2EE compliant and thus can be deploy onto any J2EE application server

Security:

- Description : Authentication and authorization mechanisms
- Solution : J2EE native security mechanisms will be reused

9. Deployment View



The user interacts with our interface to send HTTP requests to our server using the Http module on Angular framework. There is then a controller in the server that manages the request based on the endpoint which is implemented using the Java Spring Framework. The controller then manages the requests by calling the service associate which calls the needed repository. The latter will then query and return some value to our service which in turn sends back to controller. The controller then sends the requested data back to the client via a Response Entity object.