Degree Final Dissertation

Telecommunications Engineering - Telematics

Shift scheduling and management service

Servicio para la gestión de actividades asistenciales complementarias

Author: Miguel Ángel González-Alorda Cantero

Tutor: Dra. Isabel Román Martínez

Dept. Telematics Engineering
Higher Technical School of Engineering
University of Seville

Seville, 2020

Degree Final Dissertation
Telecommunications Engineering

Shift scheduling and management service

Servicio para la gestión de actividades asistenciales complementarias

Author:

Miguel Ángel González-Alorda Cantero

Tutor:

Dra. Isabel Román Martínez

Dept. Telematics Engineering

Higher Technical School of Engineering

University of Seville

Seville, 2020

Proyecto Fin de Grado: Servicio para la gestión de actividades asistenciales complementarias (Shift scheduling and management service)

Autor: Miguel Ángel González-Alorda Cantero

Tutor: Dra. Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

ACKNOWLEDGEMENTS

I think time is our most valuable resource. For this reason, I would like to thank my tutor, Isabel Román, for her availability and dedication; and specially for her will to pass her knowledge on to me. This project would not have reached its current state if she had not contributed with her knowledge and experience. Moreover, I would like to thank Miguel Ángel Rico, doctor at *Hospital Universitario Virgen Macarena*, for taking his time to provide us with the requirements of the project. I would also like to thank all of my professors, and the University of Seville, for giving me the opportunity to study this bachelor in Telecommunications Engineering.

Regarding the technologies that I have used, I would like to give a special mention to the Spring team and community. It has been because of their excellent framework, and their plenty of useful guides and documentation, that I have been able to complete this project. Moreover, I would like to acknowledge the development team of Google ORTools. It has been because of their incredible tool that I have been able to easily solve the linear programming problem modelling the scheduling problem. I would also like to mention the development team of Postman. It has been by using their tool that I have been able to easily test the REST API designed and implemented in this project. Lastly, I would like to thank the creators of ERDplus, a free online application that makes the creation of ER diagrams very simple. This is the tool I have used to represent this application's ER model.

ABSTRACT

Currently, at the Internal Medicine Department at *Hospital Universitario Virgen Macarena* (HUVM), one person needs to spend between two to three days per month scheduling and managing the doctor's shifts. For this reason, the aim of this project is to design and implement a system that automates this tasks, reducing the amount of time needed to complete them. Even though the system has been designed to meet the requirements of the Internal Medicine Department at HUVM, these are very general needs that can be extended to either other hospitals or organizations.

The solution is composed of three different systems: A simple web application, to provide an interface to the users; a REST service, to provide access to the actual data of the application; and a service responsible for scheduling the shifts, according to the given requirements. These three separate systems cooperate as follows: the web application consumes the REST API to provide the user interface; and the REST service uses the scheduling service to assign the doctor's shifts.

The solution designed does not intend to be a new technology, but rather a combination of different already existent ones. Specifically, this project uses the Spring framework to implement both the Web application and the REST service, and the Google ORTools to solve the scheduling problem.

RESUMEN

Actualmente, en el Departamento de Medicina Interna del Hospital Universitario Virgen Macarena (HUVM), una persona debe dedicar entre dos y tres días al mes a la planificación de los turnos de los médicos (guardias y continuidades asistenciales). El objetivo de este proyecto es diseñar e implementar un sistema que permita automatizar estas tareas, reduciendo el tiempo necesario para completarlas. Por otra parte, aunque el sistema vaya a ser diseñado para cumplir con los requisitos concretos del Departamento de Medicina Interna del HUVM, estos son suficientemente genéricos como para que el sistema pueda ser útil a otros hospitales u organizaciones.

En concreto, el sistema a diseñar se va a dividir en tres partes: Una aplicación web sencilla, que va a proporcionar una interfaz a los usuarios; un servicio REST, que va a ofrecer acceso a la información del sistema; y un servicio responsable de la planificación de los turnos acorde a los requisitos. Estos tres sistemas se comunican de la siguiente forma: La aplicación web utiliza la interfaz REST, y el servicio REST utiliza el servicio de planificación.

La solución que ha sido diseñada no pretende ser una nueva tecnología, sino una combinación de varias ya existentes. En concreto, en este proyecto se va a hacer uso del entorno Spring para el desarrollo de la aplicación web y del servicio REST, y de la herramienta Google ORTools para resolver el problema de planificación.

INDEX

Acknowledgements .... **I**

Abstract .... **III**

Resumen .... **V**

Index .... **VII**

List of Tables .... **XI**

List of Figures .... **XIII**

List of code snippets .... **XVII**

Notation .... **XIX**

1. Introduction **1**
1.1. Description of the problem *1*
1.1.1. The scheduling problem 1
1.1.2. The pager assignment problem 3
1.1.3. The management problem 3
1.2. Current situation *4*
1.3. Scope of the project *4*
1.4. Description of next chapters *5*

2. State of technology **7**
2.1. Linear programming *7*
2.2. UML *8*
2.3. Design pattern *9*
2.4. Python *10*
2.5. Java *13*
2.5.1. Lambda expressions 13
2.5.2. Streams API 15
2.5.3. Annotations 16
2.6. REST *17*
2.7. HAL *17*
2.7.1. Templated links 18
2.8. Spring *19*
2.8.1. Spring Web 21

2.8.1.1. Simple GET example........21
2.8.1.2. POST example........22
2.8.1.3. Path parameter and ResponseEntity example........23
2.8.1.4. Exception handling example........24
2.8.1.5. Dependency injection........25
2.8.2. Lombok........26
2.8.3. Spring Data JPA........27
2.8.4. Spring HATEOAS........30
2.8.4.1. Server side........30
2.8.4.2. Client side........33
2.8.5. Thymeleaf........34

3. Requirements........37
3.1. Actors........37
3.2. Use cases........38
3.2.1. Current situation........38
3.2.2. Desired situation........40
3.3. Behavioural requirements........42
3.3.1. Scheduling problem........43
3.3.2. Pager assignment........45
3.3.3. Management problem........47
3.4. Information requirements........48

4. Solution designed........51
4.1. Designed procedure........51
4.2. Division in subsystems........52
4.3. Entity-Relation model........54
4.4. REST service........57
4.4.1. Service's resources........57
4.4.2. Structural design........72
4.4.2.1. Model classes........74
4.4.2.2. View classes........76
4.4.2.3. DAO interfaces........78
4.4.2.4. Controller classes........78
4.4.3. Behavioural design........80
4.4.4. Communication with the Scheduler........86
4.5. Scheduler........87
4.5.1. Cyclic-shift scheduling algorithm........87
4.5.2. Non-cyclic-shift linear programming problem........88
4.5.2.1. Definitions........88
4.5.2.2. The linear programming problem........89
4.5.2.3. The objective function........90
4.5.2.4. CS implies a NCS........90
4.5.2.5. Only one shift per day........91
4.5.2.6. Maximums and minimums per doctor........91
4.5.2.7. Minimums per day........91
4.5.3. Scheduler's design........91
4.6. Web application........95
4.6.1. Structural design........95
4.6.2. Behavioural design........97

5. Conclusions and future work........105
5.1. Future work........105

5.1.1. Scheduling problem....106
5.1.1.1. Edit a schedule....106
5.1.1.2. Doctor's absences....111
5.1.1.3. Mandatory / Unavailable shifts....111
5.1.1.4. Consultation preferences....111
5.1.2. Pager assignment problem....111
5.1.3. Management problem....111
5.1.3.1. Notifying doctors of their schedule....112
5.1.3.2. Allowing shift changes....112

References....113

Appendix A – Code....1

Appendix B – Web application usage....1
1. Create a new doctor....1
2. Edit a doctor....4
3. Generate a new schedule....6

Appendix C – Deployment....1
1. Modifying the application's configuration files....2
2. Moving the different files to their corresponding locations....3
3. Configuring the database....4
4. Configuring permissions on the application's files....5

LIST OF TABLES

Table 1: A-01 - Doctor..37

Table 2: A-02 – S*hift manager*..37

Table 3: A-03 - Manager..38

Table 4: A-04 - Secretary..38

Table 5: BR-01 – Doctor's identity..42

Table 6: BR-*SCH*-01 - Types of shifts..43

Table 7: BR-*SCH*-02 - CS rate..43

Table 8: BR-*SCH*-03 - NCS allowed days..43

Table 9: BR-*SCH*-04 - Minimum regular-shifts per doctor..44

Table 10: BR-*SCH*-05 - Maximum NCS per doctor..44

Table 11: BR-*SCH*-06 – Consultations per doctor..44

Table 12: BR-*SCH*-07 - A CS implies a *regular-shift*..44

Table 13: BR-*SCH*-08 - Minimum number of regular-shifts and consultations per day..44

Table 14: BR-*SCH*-09 - Shift preferences..45

Table 15: BR-SCH-10 - History of schedules..45

Table 16: BR-SCH-11 - NCS only when CS..45

Table 17: BR-PG-01 – Pager *allowed* days..45

Table 18: BR-PG-02 - Doctors per day..46

Table 19: BR-PG-03 – Pager allowed doctors..46

Table 20: BR-PG-04 - Maximum assignments per month..46

Table 21: BR-PG-05 - History of pagers....46
Table 22: BR-PG-06 - Second pager assignment order....46
Table 23: BR-MGMT-01 - Doctor's notification....47
Table 24: BR- MGMT-02 - Awareness of changes....47
Table 25: BR-MGMT-03 - Shift changes allowed....47
Table 26: BR-MGMT-04 - Shift changes authorised....47
Table 27: BR-MGMT-04 - Shift changes registered....47
Table 28: IR-01 - Doctor....48
Table 29: IR-02 - Calendar....48
Table 30: IR-03 - Schedule....49
Table 31: Root resource....58
Table 32: Doctors resource....60
Table 33: Doctor resource....61
Table 34: ShiftConfigs resource....63
Table 35: ShiftConfig resource....64
Table 36: AllowedShifts resource....65
Table 37: Calendars resource....66
Table 38: Calendar resource....67
Table 39: Schedules resource....69
Table 40: Schedule resource....70

LIST OF FIGURES

Figure 1: Cyclic-shifts example (One cycle)....2

Figure 2: Cyclic-shifts example (Following cycles)....2

Figure 3: Excel schedule example....4

Figure 4: Class diagram example....9

Figure 5: DTO pattern example....10

Figure 6: Current use cases....38

Figure 7: Current scheduling procedure....39

Figure 8: Current shift change procedure....39

Figure 9: Desired use cases....40

Figure 10: Desired create/edit doctor procedure....40

Figure 11: Desired scheduling procedure....41

Figure 12: Desired shift change procedure....42

Figure 13: Designed use cases....51

Figure 14: Designed scheduling procedure....52

Figure 15: Designed create/edit doctor procedure - Systems communication....53

Figure 16: Designed scheduling procedure - Systems interaction....53

Figure 17: Deployment diagram....54

Figure 18: IR-01 Doctor - ER Model....55

Figure 19: IR-02 - Calendar - ER Model....56

Figure 20: IR-03 - Schedule - ER Model....57

Figure 21: REST Service - Overview class diagram....73

Figure 22: REST Service - Doctor controller overview....74

Figure 23: REST Service - Model class diagram - Doctor, Absence, DoctorStatus, ShiftConfiguration and AllowedShift....75

Figure 24: REST Service - Model class diagram - Calendar and DayConfiguration....75

Figure 25: REST Service - Model class diagram - Schedule, ScheduleStatus, ScheduleDay....76

Figure 26: REST Service - View class diagram - doctor....77

Figure 27: REST Service - View class diagram - shiftConfig and allowedShift....77

Figure 28: REST Service - View class diagram - calendar and schedule....77

Figure 29: REST Service - DAO class diagram - DoctorRepository....78

Figure 30: REST Service - ScheduleController class diagram....79

Figure 31: REST Service - DoctorController.newDoctor sequence....81

Figure 32: REST Service - DoctorController.getDoctors sequence....82

Figure 33: REST service - ScheduleController.generateSchedule sequence - Overview....83

Figure 34: REST Service - ScheduleController.generateSchedule sequence - Upper part....84

Figure 35: REST Service - ScheduleController.generateSchedule sequence - Bottom part....85

Figure 36: REST service - Communication with the Scheduler....86

Figure 37: Web application - Class diagram....96

Figure 38: Web application - DoctorsController.getDoctors sequence....97

Figure 39: Web application - DoctorsController.newDoctorForm sequence....98

Figure 40: Web application - DoctorsContrller.newDoctorForm sequence - left part....98

Figure 41: Web application - DoctorsController.newDoctorForm sequence - right part....99

Figure 42: Web application - DoctorsController.newDoctor sequence....100

Figure 43: Web application - DoctorsController.newDoctor sequence - Submit form....101

Figure 44: Web application - DoctorsController.newDoctor sequence - Persist doctor....101

Figure 45: Web application - DoctorsController.newDoctor sequence - Persist shift configuration....102

Figure 46: Web application - DoctorsController.newDoctor sequence - Response....102

Figure 47: Web application - ScheduleController.newSchedule sequence....103

Figure 48: Web application - ScheduleController.newSchedule sequence - User submits form............103

Figure 49: Web application - ScheduleController.newSchedule sequence - Request schedule generation 104

Figure 50: Web application - ScheduleController.newSchedule sequence - Redirect user......................104

Figure 51: Desired scheduling procedure - Edit schedule...106

Figure 52: Excel file example..107

Figure 53: Schedule2ExcelService class..108

Figure 54: ScheduleController.downloadExcelFor...109

Figure 55: ScheduleController.updateSchedule..110

LIST OF CODE SNIPPETS

Python – List and dictionary comprehensions example 10
Python – json module – Example 11
Python – json module – Read from file 11
Python – ortools library example 12
Java – Lambda expression – Syntax 13
Java – Lambda expressions – Runnable example 13
Java – Lambda expressions – Runnable example with lambda expression 14
Java – Lambda expressions – Another thread with MyRunnable 14
Java – Lambda expressions – Another thread with a lambda expression 14
Java – Streams API example 15
Java – Annotation declaration example 16
Java – Annotation usage example 17
REST – Request example 17
REST – Response example 17
HAL – Request example 17
HAL – Request response 18
HAL – Templated link – Optional query parameter 18
HAL – Templated link – Mandatory query parameter 19
HAL – Templated link – Path parameter 19
Spring – Directory structure 19
Spring – Spring Web – Simple GET example – GreetingController 21
Spring – Spring Web – Simple GET example – MyApplication 21
Spring – Spring Web – Build and Run the application 22
Spring – Spring Web – POST example – Book 22
Spring – Spring Web – POST example – BookController 22

Spring – Spring Web – POST example – POSTing a Book resource....23

Spring – Spring Web – Path parameter and ResponseEntity example – getBook method....23

Spring – Spring Web – Path parameter and ResponseEntity example – GETting a Book....24

Spring – Spring Web – Exception handling example – BookNotFoundException....24

Spring – Spring Web – Exception handling example – Modified getBook method....24

Spring – Spring Web – Exception handling example – MyAdviceController....24

Spring – Spring Web – Exception handling example – Querying the server....25

Spring – Spring Web – Dependency injection – BookDAO....25

Spring – Spring Web – Dependency injection – Injecting BookDAO to BookController....26

Spring – Lombok – Book redefinition with @Data....26

Spring – Lombok - @Slf4j log annotation....27

Spring – Lombok – Logging configuration....27

Spring – Spring Data JPA – Usage of @Entity....27

Spring – Spring Data JPA – Usage of JpaRepository....28

Spring – Spring Data JPA – Validation....28

Spring – Spring Data JPA - @Valid....29

Spring – Spring Data JPA – Entity's relations....29

Spring – HATEOAS – Server side – UserController....30

Spring – HATEOAS – Server side – RepresentationModelAssembler....30

Spring – HATEOAS – Server side – UserController.getTestUser method....31

Spring – HATEOAS – Server side – Request the representation state of a User in HAL....31

Spring – HATEOAS – Server side – toCollectionModel....32

Spring – HATEOAS – Server side – UserController.getTestUsers method....32

Spring – HATEOAS – Server side – Representation of a collection of resources states....32

Spring – HATEOAS – Client side – Traverson usage....33

Thyemleaf – Template example....34

NOTATION

HUVM	*Hospital Universitario Virgen Macarena*
CS	Cyclic-shift (*Jornada Complementaria*)
NCS	Non-cyclic-shift (*Continuidad Asistencial*)
NSP	Nurse Scheduling Problem
$\sum$	Sum operator
$\geq$	Greater than or equal to
$\leq$	Less than or equal to
$\forall$	For all
Mod	Modulus operator
$\mathbb{N}$	The set of natural numbers
$\subset$	Subset of
$\in$	Belong to
^{i}	Superscripts will be used to represent footnotes.
[1]	References will be represented between square brackets.
(1)	Formulas identifiers will be represented between round brackets.

1. INTRODUCTION

The aim of this project is to design and implement a system that schedules and manages medical doctor's shifts. More precisely, the system will be designed to meet the specific requirements of the internal medicine department at *Hospital Universitario Virgen Macarena* (HUVM)[i]. However, these are quite general; so the use of the system can be extended to other hospitals or organizations.

1.1. Description of the problem

The problem of scheduling and managing doctor's shifts has three different parts. The first one is scheduling the doctor's shifts. This is, assigning a set of doctors to each day and each type of shift of any given month, meeting some requirements. The second part would be, after shifts have been scheduled, assigning a pager each working day to a doctor. The third part would be, after shifts have been scheduled and pagers assigned, the actual management of these shifts. This is, notifying doctors of their shifts, and allowing them to change them with one another.

To better understand these three problems, we will study them separately:

1.1.1. The scheduling problem

The scheduling problem consists on assigning shifts to the doctors of the hospital, but meeting certain requirements. This is a common problem, also known as the Nurse Scheduling Problem, or NSP in short. Particularly, in the problem we are currently concerned with, each day may have two different types of shifts:

- Cyclic-shifts (*Jornadas Complementarias*):

  These refer to certain shifts that occur periodically, and independently of any other restriction Particularly, they correspond to the doctor's guards that start at 20.00 of a certain day, and end at 8.00 of the following day. Cyclic-shifts will be referred to as CS throughout the rest of the document.

  For example, let's say there are six different doctors whose shifts are to be scheduled: A, B, C, D, E and F; and we know that each day there has to be two shifts. Now, let's say we know the CSs that took place the first three days of a month:

[i]All the information regarding the requirements and the current situation has been directly received from the HUVM. Specifically, from the shift manager of the internal medicine department at the time this project is being developed.

Day 1	A	B
Day 2	C	D
Day 3	E	F

Figure 1: Cyclic-shifts example (One cycle)

The previous picture represents that doctors A and B had a shift on day 1, doctors C and D had a shift on day 2, and doctors E and F had a shift on day 3.

Now, if we know that CSs are repeated every three days, we can easily predict that shifts for days four to nine will be as follows:

Day 4	A	B
Day 5	C	D
Day 6	E	F
Day 7	A	B
Day 8	C	D
Day 9	E	F

Figure 2: Cyclic-shifts example (Following cycles)

The rate at which doctors have CSs is the same for all doctors.

- Non-cyclic-shifts (*Continuidades Asistenciales*):

  These refer to any other shifts that do not occur periodically, and are subject to different restrictions. Particularly, these shifts represent the ones that start at 15.00 of a certain day, and end at 20.00 of the same day. Throughout the rest of the document, they will be referred to as NCS.

  There are two types of NCSs: Consultations (*Consultas*) and Regular-shifts (*Continuidades Asistenciales*[ii]).

  The restrictions that apply to these shifts are as follows:

  1. There are certain doctor who only have CSs, others who only have NCSs, and others who have both CSs and NCSs.

  2. NCSs can only be scheduled on working days. E.g. there cannot be any consultations or regular-shifts on Saturdays or Sundays, neither on holidays.

  3. Each day, there is a minimum number of NCSs that have to be assigned.

  4. From all available doctors for NCSs, only some of these do consultations. Specifically, the doctors who do have consultations have to have a certain amount of them each month. For example, doctor A does consultations and has to have 2 consultations each month.

  5. If a doctor has a CS a certain day, they have to have a regular-shift that day (only if

[ii] Note that NCSs and regular-shifts are called identically in Spanish. The reason for this is, most NCSs are regular-shifts.

the doctor can have NCSs).

6. Some doctors can only have NCSs the same days they have CSs.

7. Each doctor has a minimum and a maximum number of NCSs they can work in each month.

8. Doctors can have certain NCSs preferences. For example, doctor B would like to have their regular-shifts on Thursdays and their consultations on Tuesdays; doctor D would not like to have their shifts on Wednesdays; doctor F would like to have a regular-shift the 5th day of next month.

1.1.2. The pager assignment problem

Each working day, one doctor needs to be responsible for a pager (*Busca*). The assignment of the pager has some restrictions:

1. Pagers only have to be assigned on working days.

2. Exactly one pager has to be assigned to one doctor each working day.

3. On a certain day, the pager has to be assigned to a doctor having a CS that day (Hence, the scheduling problem as to be solved before pagers are assigned).

4. Doctors should only have the pager assigned once per month. However, if a certain month all doctors have had it once already, some doctors have to have it twice. To decide which doctors will have the pager twice, we need the history of pager assignment[iii]. Then, the doctors who had the pager assigned twice the longest time ago should now have it assigned twice. E.g. let's say there are three doctors, A, B and C. Now, let's say doctor A had the pager assigned two days this month, and doctor B had it two days last month. Then, if a doctor needs to have the pager assigned two days the following month, it should be doctor C who has it.

1.1.3. The management problem

The management problem consists on two different parts. The first one is notifying doctors of their shifts, and the second one is allowing doctors to change their shifts with one another. Both of these will occur after the scheduling of a certain month has been completed.

Notifying the doctors of their shifts means doctors should always be able to know, for each day of the month, which doctors will be having a CS or a NCS, and which doctor will be having the pager.

With regards to the second part of the problem, if a doctor would like to change one of their CS or NCS, they have to follow these steps:

1. The doctor willing to change a shift should find another doctor who will take it.

2. After two doctors have agreed on changing a shift, they have to make a request to be reviewed by the doctor's manager.

3. The doctor's manager can either accept or decline the request.

4. In case the request is accepted, all doctors have to be notified of the change.

[iii] The history of pager assignment is just a record of which doctor had the pager assigned each day of each month.

1.2. Current situation

Currently, all three problems described above are solved manually. There is a person (which we will refer to as “shift manager”) responsible for scheduling and managing shifts. More precisely, the shift manager has to spend between two to three days per month scheduling and managing shift changes.

The scheduling problem is solved using an Excel file. This means, a table is created in an Excel file and, for each day of the month, the shift manager will assign shifts according to the restrictions. To have a better idea on this procedure, this is an extract of the schedule of a certain month (note that names have been changed to letters on purpose, to avoid revealing confidential information):

16	A	B	A #	C #	
17	D	E	F #	E #	
18	G	H	I #	J #	
19	I	K	I #	K #	L #
20	M	N	M #	O #	
21	P	Q			
22	F	R			

Figure 3: Excel schedule example

The image above shows the cycle (first two columns) and non-cycle (the rest of columns) shifts of days 16th to 22nd of a certain month. The colours have a specific meaning, but it is not currently relevant to this introduction. Pagers are assigned on another Excel file in a similar fashion.

Then, after the Excel file has been produced, it is sent via email to all the doctors, and lastly it is printed and attached to a notice board.

Next, if two doctors agree on changing a shift, they have to sign a document[iv] to confirm the will. Afterwards, the document has to be reviewed and signed by the doctor’s manager. Finally, if the change is accepted, the printed version of the Excel is manually changed on the notice board.

Note that shift changes will not be emailed back to all other doctors, so it would be up to them to check the notice board if they wanted to know the most updated version of the schedule.

1.3. Scope of the project

The scope of this project is to automate and reduce the amount of time needed to schedule the doctor’s shifts. This is, the shift manager should only need to configure the restrictions applied and the schedule should be automatically generated. Moreover, the restrictions should be saved so that, on the following schedule generations, no configuration has to be done.

The system to be designed should also allow changing any configuration related to the scheduling. This is, from something as simple as introducing a new doctor to the system, to something more sophisticated like changing which day a certain doctor prefers their regular-shifts to take place.

The system should also be able to retrieve the schedules’ history. This is, the shift manager may want to check the schedule of a certain month. For example, they may want to check the schedule of December 2019 to know which doctors had a shift on Christmas day.

[iv] The document to change a shift has a specific format, and it is provided by the HUVM. However, it is not relevant to the project, so it will not be shown.

It will **not** be in the scope of this project to solve neither the pager assignment problem nor the management problem. These will be left as further improvements that can be done to the designed system. Moreover, the most important goal of the project is to make a first working version of the system, so that we can have feedback from the doctors. With this goal in mind, the implemented version of the system might not meet all the requirements, for no other reason but to provide an initial functioning prototype.

1.4. Description of next chapters

The rest of this document will be divided into four different chapters as follows:

- 2 State of technology: A description of the key concepts needed to understand this project. Moreover, this chapter will explain how to use the technologies needed to develop this project.
- 3 Requirements: A formal description of the requirements of the system.
- 4 Solution designed: This chapter will describe the solution that has been decided for the problem. This is, the subsystems in which the solution has been divided, and the design of each of these subsystems.
- 5 Conclusions and future work: This chapter will briefly summarize the previous ones, and will give an overview on the future steps to be taken to continue with this project.

2. STATE OF TECHNOLOGY

This chapter will give an overview on the key concepts and technologies that are needed to understand this project. It will also briefly explain how to use the different technologies needed to develop the project's application.

2.1. Linear programming

Linear programming is a mathematical approach to optimally allocate certain resources, regarding a specific set of constraints. A linear programming model is defined in terms of a function of N variables to be maximized or minimized, and certain constraints that restrict the values of these variables. The function is usually referred to as Objective Function. If all the variables are restricted to integer values, the problem is called Integer Programming. [1]

Example:

$Max\, x_1+x_2-3x_3+4x_4$	Objective Function
$x_1-3x_2+x_4\leq 5$	Constraint 1
$x_1+x_3+3x_4\leq 3$	Constraint 2
$x_2\leq 1$	Constraint 3
$x_1, x_2, x_3, x_4\in \mathbb{R}$	Constraint 4

A common problem involving linear programming is the Nurse Scheduling Problem (NSP). It consists on allocating a set of workers to some shifts, according to certain restrictions. Usually, this kind of problem is modelled as an Integer Programming problem with binary variables. This is, variables can only have values '0' or '1'. [2] [3]

For example, these could be the restrictions applied to schedule the shifts in a certain bar:

- There are 8 employees in the bar.
- There are two different types shifts: afternoon and evening.

- Each day, there are two afternoon and two evening shifts.
- Each shift has to be assigned to exactly one employee.
- If an employee had an afternoon shift, they should not have an evening shift.
- An employee should not have shifts assigned three consecutive days.
- Employees may have requests for certain shifts. For example, they might want to have one of their shifts on Monday's afternoons.

Given these problem, we could define binary variables:

$$x_{ijk} \in \{0,1\} \forall i=1,2,\ldots,8\ j=1,2,\ldots,7\ k=1,2$$

Where *i* represents the employees, *j* the days of the week (1=Monday, 2=Tuesdays...) and *k* the type of shift (1=afternoon, 2=evening). This is, if $x_{1,3,2}=1$, then the employee number 1 would have a shift on Wednesday's evenings.

Then, we could define the constraints with these variables. For example, "Each day, there are two afternoon and two evening shifts" and "Each shift has to be assigned to exactly one employee" could be defined as:

$$\sum_{i=1}^{8} x_{ijk} = 2 \forall\ j=1,2,\ldots,7\ k=1,2$$

Lastly, we can define the objective function to maximize the number of requests met:

$$Max \sum_{i=1}^{8} \sum_{j=1}^{7} \sum_{k=1}^{2} r_{ijk}\, x_{ijk}$$

Where $r_{ijk} \in \{0,1\} \forall i=1,2,\ldots,8\ j=1,2,\ldots,7, k=1,2$ represents whether employee *i* wants the *k* shift on the *j* day of the week.

2.2. UML

The Unified Modelling Language is a standard developed by the OMG (Object Management Group) used to analyse, design and implement software-base systems.

For example, it can be used to represent relations between classes in a class diagram:

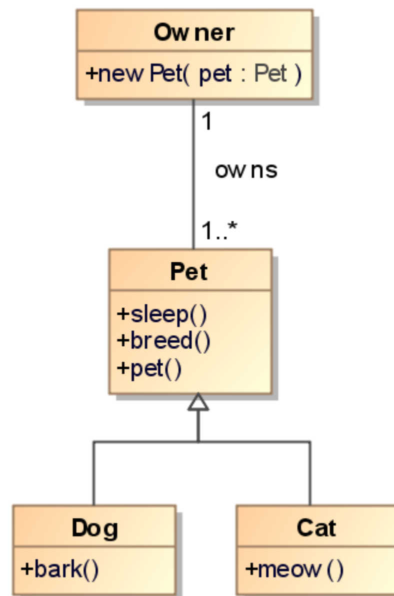


Figure 4: Class diagram example

In the previous example, an Owner is related to one or more Pets, and a Pet is related to exactly one owner. A Pet and can be either a Dog or a Cat.

More information about UML can be found in its specification [4].

2.3. Design pattern

A design pattern is a description of the most accepted solution to a specific and common design problem [5]. The following patterns are the ones relevant to the project:

- MVC Pattern: The Model-View-Controller pattern divides responsibilities within a system into three categories:
  - Model: The model classes represent the information or data of a system. E.g. a *User* class responsible for containing a *username* and an *email* would comply with the model role.
  - View: The view classes are responsible for representing the information of the model to another system or to a user. E.g. a class responsible for presenting a graphical interface would comply with the view role.
  - Controller: The controller classes contain the business logic concerning the application. E.g. a Servlet responding to HTTP requests would comply with the controller role.

  We can find an implementation of this pattern explained at [6].

- DAO Pattern: The Data Access Object pattern isolates the responsibility of data persistence. E.g. let's say we have a system whose persistence relies on a relational database. Then, a class implementing the DAO pattern would be responsible for communicating with that database to either extract or persist information. This way, the rest of the application would not need to know the communication process with the database. We can find a simple implementation of this pattern in the example at [7].

- DTO Pattern: The Data Transfer Object pattern consists on separating the internal and external representations of an object. Understanding internal representation as the classes that model information inside a system, and external representation as the model information that gets serialized and sent to a different system. This is easier to understand with an example:

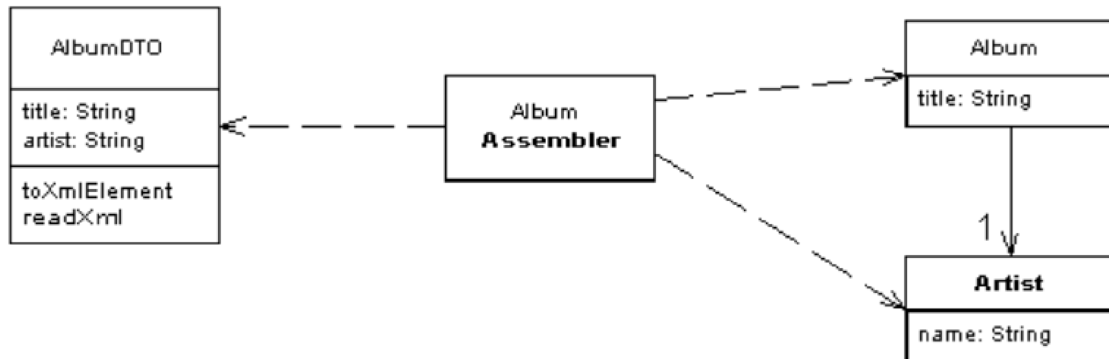


Figure 5: DTO pattern example

In the previous example, the internal representation of an album is composed of two classes: *Album* and *Artist*. However, its external representation should just be the fields *title* and *artist*. For this purpose, the *AlbumDTO* class is created. This example has been taken from [8].

2.4. Python

A scripted object-oriented strongly and dynamically typed language. Some of the most basic features of Python can be found at [9]. Particularly, for this project, we will use Python 3.7. Its official documentation can be found at [10].

To understand the code written for this project, we need to understand two advanced features of the language: list and dictionary comprehensions. These two features allow creating `lists` [11] and `dicts` [12] in just one line of code:

Python – List and dictionary comprehensions example

```
numbers = [i for i in range(10)]
# Result: numbers from 0 to 9 -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# The previous list comprehension is equivalent to:
numbers = [] # Declare an empty list
for i in range(10): # 'i' will have the values 0, 1,..., 9
  numbers.append(i) # Add 'i' to the end of the list

evenNumbers = [num for num in numbers if num%2 == 0]
# Result: even numbers in the 'numbers' list -> [0, 2, 4, 6, 8]
# Note '%' is the modulus operator in python
# The previous list comprehension is equivalent to:
evenNumbers = []
for num in numbers:
  if num%2 == 0:
    evenNumber.append(num)

isOdd = {num: num%2 != 0 for num in range(5)}
```

```
# Result: a dict with boolean values indicating whether a
# number is odd or not
# -> {0: False, 1: True, 2: False, 3: True, 4: False}
# The previous dict comprehension is equivalent to:
isOdd = {} # Declare an empty dict
for num in range(5):
  isOdd[num] = num%2 != 0 # Assign to the key 'num' the
                          # value 'num%2 != 0'
```

More information on these two features can be found at [13] and [14].

There are also two important python modules needed to understand this project:

- json [15]:A module from the standard library used to encode and decode JSON [16] strings.

  Some basic examples taken from the official documentation are:

  Python – json module – Example
  ```
  import json

  # Convert from a list to an encoded string
  json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
  # Result: '["foo", {"bar": ["baz", null, 1.0, 2]}]'

  # Convert from encoded string to a list
  json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
  # Result: ['foo', {'bar': ['baz', None, 1.0, 2]}]
  ```

  This module will be used in the project to read and write JSON files. For example:

  Python – json module – Read from file
  ```
  import json

  with open('doctors.json') as doctorsFile:
    doctors = json.loads(doctorsFile.read())
  ```

  The previous example reads the 'doctors.json' file and parses it as JSON. The resulting value is assigned to the `loggingConf` variable.

  Where the 'doctors.json' file content could be the one defined in 4.5.3 Scheduler's design section.

- ortools [17]: An open source software suite developed by Google that allows solving optimisation problems. Specifically, we are interested in the `ortools.sat.python.cp_model` module [18].

  From this module, we are interested in four classes: `CpModel`, `CpSolver`, `IntVar` and `Constraint`.

  - `CpModel` provides methods for defining a Constraint Programming (CP) problem.
  - `IntVar` will represent the integer variables of the problem.
  - `Constraint` will represent, as its name suggests, the constraints of the problem.

- `CpSolver` is responsible for finding the optimal solution of a given `CpModel`.

The usage of these classes is better illustrated with an example. First, let's define a simple binary programming problem:

$$Max \sum_{i=0}^{4} x_i$$

$$5x_0 + 2x_1 - 3x_3 \leq 3$$

$$x_2 + x_4 = 1$$

$$x_i \in \{0,1\} \forall i = 0,1,2,3,4$$

Now, we will present the code that can be used to define and solve the problem:

Python – ortools library example

```
from ortools.sat.python import cp_model

# Create the model instance
model = cp_model.CpModel()

# Create five boolean variables
vars = [model.NewBoolVar(f'var{i}') for i in range(5)]

# Create the constraints
model.Add(5*vars[0] + 2*vars[1] - 3*vars[3] <= 3)
model.Add(vars[2] + vars[4] == 1)

# Define the objective function
model.Maximize(sum(vars))

# Solve the problem
solver = cp_model.CpSolver()
status = solver.Solve(model)
# The most common status' of a solution are:
# cp_model.FEASIBLE, cp_model.OPTIMAL or cp_model.INFEASIBLE

# Obtain the value of a variable
solver.Value(vars[0])
```

There are two comments we can add to the above code:

- We are creating the variables of the problem with the factory method `NewBoolVar` from our `CpModel` object. Particularly, this method creates and instance of an `IntVar` whose values are restricted to 0 or 1 (just like we defined it in the mathematical model). Note this method takes a string as an argument, which is the name of the created variable (`var1, var2...` in our example). To create this string, we are using f-strings [19].

- Note that we are seamlessly creating instances of the `Constraint` class by operating on the variables (`vars[2] + vars[4] == 1`). The reason for this is the `IntVar` class has overloaded the mathematical operators and the comparison operators.

Google provides a more advanced example on the usage of their module at [20].

2.5. Java

An interpreted object-oriented language. A basic introduction to the language and some of its features can be found at [21]. Particularly, this project will be developed using Java 8. Its official documentation can be found at [22].

To understand the code written for this project, we will briefly describe three advanced features of this language:

2.5.1. Lambda expressions

Lambda expressions provide a concise way of creating a one method anonymous class [23] instance. This is, with just one expression, we can declare and instantiate a class that only declares one method.

First of all, we will show the syntax of lambda expressions, and then provide examples of its usage:

Java – Lambda expression – Syntax

```
(arg1, arg2...) -> {
  // Inside these brackets, we can write regular Java code
  arg1.doSomething();
  arg2.doSomethingElse();
  ...
  return ...;
}
```

`arg1`, and `arg2` would be the parameters of the method being declared. If the method has no parameters, the brackets will be empty "`() -> {...}`". Note that, as in regular method declarations, if the method being declared returns `void`, the `return` statement is not necessary.

For example, let's say we need a class that implements the java.lang.Runnable interface (it only declares one method `run(void)->void`). However, this class will only be used once in our code. For example, to create a java.lang.Thread. Then, a possible solution to this could be:

Java – Lambda expressions – Runnable example

```
// We first declare a class that implements the interface
public class MyRunnable implements Runnable {
  public void run() {
    System.out.println("Hello, world!");
    // Some other code
  }
}

// Then, in some other class
...
Thread myThread = new Thread(new MyRunnable());
myThread.start();
...
```

However, this previous code can be simplified by using a lambda expression (the code that is simplified has been highlighted in yellow):

Java – Lambda expressions – Runnable example with lambda expression

```
...
Thread myThread = new Thread(() -> {
  System.out.println("Hello, world!");
  // Some other code
});
myThread.start();
...
```

Note that it is specially important knowing that this implementation of the Runnable interface will only be used once in our code. The reason for this is the created class is anonymous and cannot be instantiated again. E.g. if we wanted to create a new thread, in the first example we would only need to create another instance of `MyRunnable`. However, on the second example, we would have to create a new anonymous class using a lambda expression, which would lead to duplicated code:

Java – Lambda expressions – Another thread with MyRunnable

```
// In some class
...
Thread myThread = new Thread(new MyRunnable());
...

// In some other class
...
Thread anotherThread = new Thread(new MyRunnable());
...
```

Java – Lambda expressions – Another thread with a lambda expression

```
// In some class
...
Thread myThread = new Thread(() -> {
  System.out.println("Hello, world!");
  // Some other code
});
...

// In some other class
...
Thread anotherThread = new Thread(() -> {
  // This code would be duplicated
  System.out.println("Hello, world!");
  // Some other code
});
...
```

If the method being declared only contains one statement, the syntax can be further simplified:

```
Thread myThread = new Thread(() -> System.out.println("Hello!"));
// In general (arg1, arg2...) -> someStatement...
```

With this simplified syntax, if the method being declared had a return type other than void, the value returned by `someStatement` would be returned. For example, we can use a lambda expression to create an instance of a class implementing the java.util.Comparator<E> interface:

```
Comparator<Integer> comp = (num1, num2) -> num1 - num2;
```

For further information on Lambda expressions, refer to the Oracle documentation at [24].

2.5.2. Streams API

Briefly described, the Streams API allows performing operations in sequences of elements from a declarative programming [25] perspective. From the official documentation [26]:

“Streams differ from collections in several ways:

- **No storage**. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.

- **Functional in nature**. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

- **Laziness-seeking**. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.

- **Possibly unbounded**. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.

- **Consumable**. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.”

Stream operations can be divided into *intermediate* and *terminal* operations. This operations are combined to form a *stream pipeline*, which consists of a data source such as a Collection; then zero or more *intermediate* operations such as Stream.map; and a *terminal* operation such as Stream.reduce.

One of the Streams API common use cases is applying the Filter/Map/Reduce design pattern [27]. For example, we can calculate the sum of the squares of all even numbers from 1 to 10 as:

Java – Streams API example

```
// First, we create the list
List<Integer> myList = new ArrayList<>();
for (int i = 1; i <= 10; i++) {
  myList.add(i);
}
// Apply the streams api to get the result
int result = myList.stream()
  .filter(i -> i%2 == 0)
  .map(i -> i*i)
  // The first argument of reduce is partialSum’s initial value
  .reduce(0, (partialSum, i) -> partialSum + i);
```

Note: `filter` requires an instance of a class implementing the java.util.function.Predicate interface, `map` requires an instance of a class implementing the java.util.function.Function interface and `reduce`’s second argument has to be an instance of a class implementing the java.util.function.BiFunction interface. For all of these, we are using lambda expressions.

For further information, refer to the official documentation [26].

2.5.3. Annotations

From the Oracle tutorial at [28]:

"Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

- Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
- Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.
- Runtime processing — Some annotations are available to be examined at runtime."

We can annotate, among others, method declarations, class declarations and properties of a class. To use an annotation, we prefix the annotation name with the at sign (@). E.g. (`@Entity`)

A common use of annotations is to declare a method is being overridden:

```
@Override
String toString() {...}
```

The `@Override` annotation informs the compiler we are trying to override an element declared in a superclass. This annotation is not required. However, it will make the compiler generate an error if the element does not correctly override a method on one of its superclasses. This allows us to easily detect an error that could otherwise be harder to find.

Now, let's say we want to create an annotation of our own that will be used to declare metadata about a class, such as its author, the last modified date… Then, the information in this annotation has to be included in the generated javadoc (We know the standard javadoc already provides tags to include this information, but this is just an example). To do this, we use `@interface`:

Java – Annotation declaration example

```
import java.lang.annotation.Documented;

// This will make the information in ClassPreamble appear
// in Javadoc-genereted documentation
@Documented
// We declare the annotation
@interface ClassPreamble {
   String author();
   String date();
   // Annotation's elements can have default values
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
   // Note use of array
   String[] reviewers();
}
```

Now, we can use the annotation as follows:

Java – Annotation usage example

```
@ClassPreamble (
   author = "John Doe",
   date = "3/17/2002",
   currentRevision = 6,
   lastModified = "4/12/2004",
   lastModifiedBy = "Jane Doe",
   // Note array notation
   reviewers = {"Alice", "Bob", "Cindy"}
)
public class SomeClass extends SomeOtherClass {…}
```

This example has been taken from the previously mentioned Oracle tutorial. Refer to it for further information on annotations.

2.6. REST

REST stands for Representational State Transfer [29], and it is an architectural design pattern that can be used to communicate distributed systems. Specifically, this pattern states we can identify *Resources* within a system (which represent the system's information). Then, exchanged data corresponds to the representation of the state of a specific *Resource.* Lastly, the basic CRUD (Create, Read, Update and Delete) operations should be provided to be able to act upon these resources.

Usually, a REST service uses HTTP [30], and takes advantage of its methods to provide CRUD operations on the resources: GET, to read; POST, to create; PUT, to update; DELETE, etc. Then, the resources are usually encoded as JSON [16] or XML [31] and included in the body of the requests. For example:

REST – Request example

```
GET users/1 HTTP/1.1
Accept: application/json
```

REST – Response example

```
HTTP/1.1 200 OK
Content-type: application/json

{“username”: “example1234”, “email”: “example@example.com”...}
```

2.7. HAL

HAL stands for JSON Hypertext Application Language [32]. Simply put, HAL defines certain JSON properties to represent resources and their related hyper-links. E.g. the property *__links* represents links related to a resource. To better understand HAL, the following example represents a GET request of the state of an order resource, and its response:

HAL – Request example

```
GET /orders/523 HTTP/1.1
Host: example.org
```

```
Accept: application/hal+json
```

HAL – Request response

```
HTTP/1.1 200 OK
Content-Type: application/hal+json

{
   "_links": {
     "self": { "href": "/orders/523" },
     "warehouse": { "href": "/warehouse/56" },
     "invoice": { "href": "/invoices/873" }
   },
   "currency": "USD",
   "status": "shipped",
   "total": 10.20
}
```

The state of the resource is accessed by the URI "/orders/523", with links to its "warehouse" and "invoice" associated resources. This example has been taken from the HAL specification.

Note that the inclusion of hypertext (*__links*) will simplify the client's development. For instance, the client would not need to evaluate resource's states, but could just make decisions based to the response's links. E.g. let's imagine the order from the previous example had a state in which it could be cancelled. Then, if the order was in that state the server would include a "cancel" link in the response. This way the client knows an order can be cancelled only when this link is present.

2.7.1. Templated links

A relevant feature to this project is that link representations is HAL can be templated. This is, they can contain parameters whose values will be chosen by the client before making a request. The process of choosing the values for the template parameters will be referred to as building a link.

Now, we will provide different examples on the types of link parameters that will be used in this project. However, in this section, we will not explain the meaning of these links, but just show how they are represented and how to build them. To know the meaning of the links, refer to 4.4.1 Service's resources.

For example, a link such as:

HAL – Templated link – Optional query parameter

```
“doctors”: {
  “href”: “DOMAIN/guardians/doctors/{?email}”,
  “templated”: true
}
```

Represents a link with an optional query parameter "`email`". The client could then decide to ignore this parameter and build the link as:

```
DOMAIN/guardians/doctors/
```

or could give it a value and build the link as:

```
DOMAIN/guardians/doctors/?email=example@example.com
```

Query parameters can also be mandatory, such as in this example:

HAL – Templated link – Mandatory query parameter

```
"newDoctor": {
  "href": "DOMAIN/guardians/doctors/startDate={?startDate}",
  "templated": true
}
```

This way, to build the link, the client would have to assign a value to "`startDate`". For example:

```
DOMAIN/guardians/doctors/?startDate=2020-08-15
```

One last comment on templated links is they can also contain path parameters. This is, a parameter needed to complete the URI. For example:

HAL – Templated link – Path parameter

```
"doctor": {
 "href": "DOMAIN/guardians/doctors/{doctorId}",
  "templated": true
}
```

This way, the client would have to assign a value to "`doctorId`". For example:

```
DOMAIN/guardians/doctors/1
```

2.8. Spring

Spring is an open source framework for programming in Java, with useful libraries to (among others) develop REST services. This section describes the most relevant features of this framework needed to understand this project, as it has been developed using this framework. For further information, refer to the Spring's official site at [33].

First of all, to start a Spring project from scratch, the website at [34] provides with a graphical interface to select the project type and its dependencies. This project will use Maven [35] as its management tool. All of the dependencies of this project can be found on the repositories at Appendix A – Code. Still, we will explain the most relevant ones in this section.

Before that, to be able to understand the following sections, we need to know the basic structure of Spring projects:

Spring – Directory structure

```
projectMainDirectory/
 |- src/
 |   |- main/
 |   |   | # The packages inside the project's main package have
 |   |   | # been chosen arbitrarily, and can be different on
 |   |   | # different applications. These will be the ones used
 |   |   | # in the examples of this section
 |   |   |- my/project/package/
 |   |   |   |- controllers/
 |   |   |   |   |- GreetingController.java
 |   |   |   |   |- BookController.java
 |   |   |   |   |- MyAdviceController.java
 |   |   |   |   |- ...
```

```
|   |   |   |- exceptions/
|   |   |   |   |- BookNotFoundException.java
|   |   |   |   |- ...
|   |   |   |- entities/
|   |   |   |   |- Book.java
|   |   |   |   |- ...
|   |   |   |- daos/
|   |   |   |   |- BookDAO.java
|   |   |   |   |- ...
|   |   |   |- MyApplication.java
|   |   |   |- ...
|   |- resources/
|   |   |- static/ # Contains static HTML, JS and CSS
|   |   |   |- index.html
|   |   |   |- ...
|   |   |- templates/ # See 2.8.5 Thymeleaf
|   |   |   |- myTemplate.html
|   |   |   |- ...
|   |   |- application.properties # Main configuration file
|   |   |- ...
|   |- test/
|   |   |- my/project/package/  # Contains Test classes
|   |   |   |- controllers/
|   |   |   |   |- GreetingControllerTests.java
|   |   |   |   |- BookControllerTests.java
|   |   |   |   |- MyAdviceControllerTests.java
|   |   |   |   |-...
|   |   |   |- exceptions/
|   |   |   |   |- BookNotFoundExceptionTests.java
|   |   |   |   |- ...
|   |   |   |- entities/
|   |   |   |   |- BookTests.java
|   |   |   |   |- ...
|   |   |   |- MyApplicationTests.java
|   |   |   |- ...
|- mvnw    # These two last files will be present only if we use
|- pom.xml # Maven as the project's management tool.
```

Besides the explanations that will be given in the following sections, the following guides have been key to develop this project:

- Building a RESTful service with Spring [36].
- REST Beyond the Obvious – Oliver Gierke [37].
- Accessing data with MySQL [38].
- How To Do *@Async* In Spring | Baeldung [39].
- Guide To Internationalization In Spring Boot | Baeldung [40].
- Spring Boot: Customize Whitelabel Error Page | Baeldung [41] (E.g. 404 Not Found page).

We will now explain how to use the Spring's features relevant to this project. Note we will only explain how to use the features, but not how they are implemented internally. For further information with this

regards, refer to the documentation of each library.

2.8.1. Spring Web

The Spring Web project provides an implementation of the Model-View-Controller pattern. This section explains how to use the different annotations provided by this project to configure and run a web application. For further information on the underlying logic, refer to the official documentation at [42]. The post at [43] also provides a general overview on the internals of the Spring web project.

2.8.1.1. Simple GET example

In this example, we will explain how to receive and respond to a simple GET request. First of all, we will create a controller class that will respond to GET requests to the /hello/world URI:

Spring – Spring Web – Simple GET example – GreetingController

```
package my.project.package.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller // Declares this class will handle requests
@RequestMapping("/hello") // Handles request starting with /hello
public class GreetingController {
  @GetMapping("/world") // Handle GET request to /hello/world
  @ResponseBody // The return value will be the HTTP response body
  public String greeting() {
    return "Hello, World!";
  }
}
```

Lastly, we will have to create the entry point of the application:

Spring – Spring Web – Simple GET example – MyApplication

```
package my.project.package;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {
  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

This application class will be common to all the following examples. It uses the Spring Boot autoconfiguration to create a Servlet container (Apache Tomcat [44] by default) listening to HTTP requests (on port 8080 by default). For further information on Spring Boot, refer to its official documentation at [45].

Now, to run the application, we can build the project as a jar file:

Spring – Spring Web – Build and Run the application

```
./mvnw clean install  # Run from the project main directory
java -jar target/GENERATED_FILE.jar
```

The generated jar file will contain all the project dependencies, so that we only need a JVM (Java Virtual Machine) to run our application. Note the name of the generated file will depend `artifactdId` and `version` configured on pom.xml.

We can check it is working using `curl [46]`:

```
>> curl localhost:8080/hello/world
Hello, World!
```

2.8.1.2. POST example

Now, we will configure a controller that will handle POST messages and will respond with a JSON object. First of all, we will create a simple POJO (Plain Old Java Objet) to represent a book:

Spring – Spring Web – POST example – Book

```
package my.project.package.entities;

public class Book {
  private Integer id;
  private String title;
  private String author;

  // Setters and Getters...
}
```

Now, we will create the controller:

Spring – Spring Web – POST example – BookController

```
pacakge my.project.package.controllers;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController // Short for @Controller and @ResponseBody
@RequestMapping("/books")
public class BookController {
  @PostMapping("") // Handle POST requests to /books/
  // The request body has to have a Book object encoded
  public Book newBook(@RequestBody Book book,
      // There will be an optional query parameter called
      //‘salutation’
      @RequestParam(required = false) String salutation) {
    System.out.println("Salutation is: " + salutation);
    book.setId(1);
    // The book will be sent back in the response body
```

```
      return book;
    }
  }
```

As mentioned before, all applications need an entry point like the one in 2.8.1.1 Simple GET example. They will also have to be built and launched the same way. So this step will be skipped in the following explanations.

Now we can test the controller:

Spring – Spring Web – POST example – POSTing a Book resource

```
# First, without the query parameter
>> curl -X POST -H 'Content-Type: application/json' \
>> localhost:8080/books \
>> -d '{"title":"Lord of the Rings", "author": "J.R.R. Tolkien "}'
{"id":1,"title":"Lord of the Rings","author":"J.R.R. Tolkien "}
# On the server console: 'Salutation is: null'

# Now, with the query parameter
>> curl -X POST -H 'Content-Type: application/json' \
>> localhost:8080/books?salutation=hello \
>> -d '{"title":"Lord of the Rings", "author": "J.R.R. Tolkien "}'
{"id":1,"title":"Lord of the Rings","author":"J.R.R. Tolkien "}
# On the server console: 'Salutation is: hello'
```

Note that, by default, Spring will serialize the response body object as JSON.

2.8.1.3. Path parameter and ResponseEntity example

Now, continuing with the previous example, we will add a GET mapping to the BookController:

Spring – Spring Web – Path parameter and ResponseEntity example – getBook method

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
...

@GetMapping("{id}") // Get requests to '/books/{id}'
// The id argument will be mapped from the {id} parameter in
// the path
public ResponseEntity<Book> getBook(@PathVariable Integer id) {
  if (id != 1) {
    // Respond with 404 Not Found
    return ResponseEntity.notFound().build();
  } else {
   // Respond 200 Ok and the book in the response body
   return ResponseEntity.ok(new Book(1, "Lord of the Rings",
                                    "J.R.R. Tolkien"));
  }
}
```

Now, we can test the controller:

Spring – Spring Web – Path parameter and ResponseEntity example – GETting a Book

```
>> curl localhost:8080/books/1 -v
...
# Server Response
HTTP/1.1 200
Content-Type: application/json
...

{"id":1,"title":"Lord of the Rings","author":"J.R.R. Tolkien"}

>> curl localhost:8080/books/2 -v
...
# Server Response
HTTP/1.1 404
Content-Length: 0
...
```

2.8.1.4. Exception handling example

Let's suppose we do not want to deal with the ResponseEntity on our controller, as in the previous example. Then, we can throw an exception, and handle it from a RestControllerAdvice. First, let's create the exception:

Spring – Spring Web – Exception handling example – BookNotFoundException

```
package my.project.package.controller.exceptions;

public class BookNotFoundException extends RuntimeException {
  public BookNotFoundException(Integer bookId) {
    super("Could not find book with id: " + bookId);
  }
}
```

Now, we will change the GET handler on our BookController:

Spring – Spring Web – Exception handling example – Modified getBook method

```
@GetMapping("{id}/")
public Book getBook(@PathVariable Integer id) {
  if (id != 1) {
    throw new BookNotFoundException(id);
  } else {
    return new Book(1, "Lord of the Rings", "J.R.R. Tolkien");
  }
}
```

And lastly, we have to define a RestControllerAdvice (These type of controllers handle thrown exceptions):

Spring – Spring Web – Exception handling example – MyAdviceController

```
package my.project.package.controllers;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
```

```
import org.springframework.web.bind.annotation.ResponseStatus;
import
  org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice // @ControllerAdvice plus @ResponseBody
public class MyAdviceController {
  @ExceptionHandler(BookNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  public String bookNotFound(BookNotFoundException e) {
    return e.getMessage();
  }
}
```

We can now test the controller advice:

Spring – Spring Web – Exception handling example – Querying the server

```
>> curl localhost:8080/books/2
...
# Server Response
HTTP/1.1 404
Content-Length: 30
...
Could not find book with id: 2
```

2.8.1.5. Dependency injection

As explained at [47] dependency injection allows for loose coupling of components, and moves the responsibility of managing components onto the framework. This improves modularity, and makes testing easier (as it allows injecting mocks). Specifically, in Spring, dependency injection is controlled with two annotations: @Autowired and @Component.

Continuing with the previous example, let's say we want to persist our Books. To do this, we are going to create a DAO responsible for dealing with the database:

Spring – Spring Web – Dependency injection – BookDAO

```
package my.project.package.daos;

import org.springframework.stereotype.Component;

// We declare de BookDAO as a component of our application
@Component
public class BookDAO {
  public Book persist(Book book) {
    // Logic to persist a Book...
  }

  public Book findById(Integer id) {
    // Logic to find a Book...
  }

 ...
}
```

Now, if we need an instance on this class in the BookController, we just have to inject it with the

Autowired annotation:

Spring – Spring Web – Dependency injection – Injecting BookDAO to BookController

```
import org.springframework.beans.factory.annotation.Autowired;
...

@RestController
@RequestMapping("/books")
public class BookController {
  // This will have Spring inject an instance of BookDAO
  @Autowired
  private BookDAO bookDao;
  ...
}
```

This way, Spring would analyse the components of our application, their dependencies, and would then inject them. For further information on other uses and the implementation of dependency injection in Spring, refer to its official documentation at [48].

2.8.2. Lombok

Lombok [49] is a Java library that provides annotations to reduce boilerplate code[v]. Specifically, in this project, we will make use of two of its annotations:

- lombok.Data:

  This class level annotation generates Setter and Getter methods for all properties of the class. It also generates a toString, equals and hashcode methods. For example, we can now declare our Book class as follows:

  Spring – Lombok – Book redefinition with @Data

  ```
  package my.project.package.entities;

  import lombok.Data;

  @Data
  public class Book {
    private Integer id;
    private String title;
    private String author;
  }
  ```

  If we convert a value to string:

  ```
  System.out.prinln(
      new Book(1, "Lord of the Rings", "J.R.R. Tolkien"));
  ```

  We get:

  ```
  Book(id=1, title=Lord of the Rings, author=J.R.R. Tolkien )
  ```

- lombok.extern.slf4j.Slf4j:

[v] Boilerplate code refers to a section of code that is widely repeated with few modifications. For example, getters and setters in a POJO (Plain Old Java Object) class.

This class level annotation injects an org.slf4j.Logger [50] into the annotated class. The name of the generated property will be *log*. Usage example:

Spring – Lombok - @Slf4j log annotation

```
package my.project.package.controllers;

import lombok.extern.slf4j.Slf4j;
...

@Slf4j // This will declare a private final static
       // property called log
public class BookController {
  @PostMapping("")
  public Book newBook(@RequestBody Book book,
    @RequestParam(required = true) Optional<String>
salutation) {
    log.info("Salutation is: " + salutation);
    log.debug("The sent book is: " + book);
    book.setId(1);
    return book;
  }
  ...
}
```

The log can be configured in the application.properties file. For example:

Spring – Lombok – Logging configuration

```
logging.level.my.project.packages = debug
logging.level.my.promect.packages.controllers = info
logging.level.my.project.packages.controllers.exceptions = \
  warn
logging.file.name = myProject.log
```

2.8.3. Spring Data JPA

From the official Spring Data JPA home page [51]:

"Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically."

To start using the Spring Data JPA repositories, we need to define our entities using the Java Persistence API [52] (JPA). For example, we can define a simple User entity as:

Spring – Spring Data JPA – Usage of @Entity

```
package my.project.package.entities;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity // Declare User as a JPA entity
```

```
public class User {
    @Id // The id property will be the primary key of User
    @GeneratedValue // The value of id will be autogenerated
    private Integer id;
    private String username;
    private String email;

    // Getters and Setters
}
```

Then, we can declare a JpaRepository [53] that will allow us to perform all CRUD operations on this entity:

Spring – Spring Data JPA – Usage of JpaRepository

```
package my.project.package.daos;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository
      extends JpaRepository<User, Integer> {
    // There is no need to declare methods on the interface
    // The basic CRUD operations are already provided with the
    // 'save(User)->User', 'findById(Integer)->User',
    // 'findAll()->List<User>' anb 'deleteById(Integer)->void'
    // methods.
    // These are only some of the methods declared by the
    // JpaRepository interface. Refer to its documentation for more.
}
```

JPA will then auto-generate a class implementing this simple interface. This implementation of this will depend on the configured database of the project.

Now, let's say we want to add some validation logic to the User. For example, the username cannot be empty, and the email has to be valid:

Spring – Spring Data JPA – Validation

```
package my.project.package.entities;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;

@Entity
public class User {
    @Id
    @GeneratedValue
    private Integer id;
    @NotBlank
    private String username;
    @Email
    private String email;

    // Getters and Setters
}
```

These new annotations will be used by the javax.validation.Validator [54] [55] to check whether an

instance of an entity is valid or not. Particularly, entities will always be validated before they are persisted. Moreover, we can make our Controllers only accept valid entities by using the @Valid [56] annotation:

Spring – Spring Data JPA - @Valid

```
// Inside some controller
...
@PostMapping
public User newUser(@Valid User user) {
  // Code inside this function will only be called if the received
  // User is valid. Otherwise, an error message will be
  // automatically returned.
}
...
```

The available validation annotations can be found at [57]. Custom validation annotations can also be added. Refer to [58] for an explanation on how to create them.

We can also define more relations between entities using JPA. For example, let's add an Address and a list of friends to our User:

Spring – Spring Data JPA – Entity's relations

```
package my.project.package.entities;
// New user class
import javax.persistence.ManyToMany;
import javax.persistence.OneToOne;

@Entity
public class User {
  @Id
  @GeneratedValue
  private Integer id;
  private String username;
  private String email;

  @OneToOne // One User entity is related to exactly one Address
  private Address address;

  @ManyToMany // Each User can be related to many Users
  private List<User> friends;
}

// Address class
import javax.persistence.MapsId;

@Entity
public class Address {
  @Id
  private Integer userId;
  @MapsId // Define userId as a foreign key of the User entity
  @OneToOne
  private User user;
}
```

For further explanations on how to map entities to classes, refer to [59]. For information on how to declare composite primary keys (made of several properties) refer to [60].

2.8.4. Spring HATEOAS

HATEOAS stands for Hypermedia as the Engine of Application State. As from the project's home page [61]:

"Spring HATEOAS provides some APIs to ease creating REST representations that follow the HATEOAS principle when working with Spring and especially Spring MVC. The core problem it tries to address is link creation and representation assembly."

We can find further information on the HATEOAS principle at [62]. However, this section will focus on how to use the Spring HATEOAS libraries for link creation and resource assembly.

2.8.4.1. Server side

The main classes involved in resource representation on Spring HATEOAS are *EntityModel* [63] and *CollectionModel* [64]. As their names suggests, they represent a single entity, along with their relations; and a collection of entities, along with its relations. To convert an entity in our application to their *EntityModel* representation, we need a resource assembler: *RepresentationModelAssembler* [65].

To provide an example on link creation, we will add new mappings to the *UserController* from our previous User example:

Spring – HATEOAS – Server side – UserController

```
...
@RestController
@RequestMapping("/users")
@Slf4j
public class UserController {
  @Autowired
  private UserRepository userRepository;
  @GetMapping("")
  public List<User> getUsers(
        // We have added a new optional parameter
        @RequestParam(required = false) String salutation) {
    log.info("Salutation is: " + salutation);
    return userRepository.findAll();
  }

  // We have added a new GET mapping: /users/{id}
  @GetMapping("/{id}")
  public User getUser(@PathVariable Integer id) {
    return userRepository.findById(id).orElse(null);
  }
}
```

Now, we can create a *UserAssembler* that will create links pointing to these methods:

Spring – HATEOAS – Server side – RepresentationModelAssembler

```
package my.project.package.assemblers;

import static
  org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
import org.springframework.hateoas.EntityModel;
import
```

```
  org.springframework.hateoas.server.RepresentationModelAssembler;
import org.springframework.stereotype.Component;

@Component // This allows autowiring
public class UserAssembler
implements RepresentationModelAssembler<User, EntityModel<User>> {
  @Override
  public EntityModel<User> toModel(User entity) {
    return EntityModel.of(entity,
      linkTo(methodOn(UserController.class)
              .getUser(entity.getId()))
          .withSelfRel(),
      linkTo(methodOn(UserController.class).getUser(null))
          .withRel("user"),
      linkTo(methodOn(UserController.class).getUsers(null))
          .withRel("users"),
     linkTo(methodOn(UserController.class).getUsers("HelloWorld"))
          .withRel("usersWithSalutation"));
  }
}
```

The attributes passed on the methods are used by the HATEOAS library to construct the links. For example, on the first link, the path variable *id* from the link `/users/{id}` will get substituted by the user's id. However, on the second link, as the id parameter is *null*, it will not be substituted (see following example).

To try this assembler, let's add a new mapping to our *UserController*:

Spring – HATEOAS – Server side – UserController.getTestUser method

```
@GetMapping("/test-user")
public EntityModel<User> getTestUser() {
  User user = new User();
  user.setId(1);
  user.setUsername("example");
  user.setEmail("example@example.com");
  // Assume the UserAssembler has been autowired as
  // 'userAssembler'
  return userAssembler.toModel(user);
}
```

Now, to test it, let's make a get request:

Spring – HATEOAS – Server side – Request the representation state of a User in HAL

```
>> curl localhost:8080/users/test-user
{
 "id": 1,
...
 "_links": {
   "self": {
     "href": "http://localhost:8080/users/1"
   },
   "user": {
     "href": "http://localhost:8080/users/{id}",
     "templated": true
```

```
    },
    "users": {
      "href": "http://localhost:8080/users/{?salutation}",
      "templated": true
    },
    "usersWithSalutation": {
      "href": "http://localhost:8080/users/?salutation=HelloWorld"
    }
  }
}
```


Note that *EntityModel*s are serialized as HAL, and not just bare JSON.

The *RepresentationModelAssembler* has another useful method that can be overridden:

Spring – HATEOAS – Server side – toCollectionModel

```
// Inside the UserAssembler class
...
@Override
public CollectionModel<EntityModel<User>>
    toCollectionModel(Iterable<? extends User> entities) {
  List<EntityModel<User>> userEntities = new LinkedList<>();
  for (User user : entities) {
    userEntities.add(this.toModel(user));
  }
  return CollectionModel.of(userEntities,
    linkTo(methodOn(UserController.class).getUsers(null))
      .withSelfRel());
}
...
```

And now, we can define a new mapping on the *UserController* to test this method:

Spring – HATEOAS – Server side – UserController.getTestUsers method

```
@GetMapping("/test-users")
public CollectionModel<EntityModel<User>> getTestUsers() {
  List<User> users = new LinkedList<>();
  User user = null;
  for (int i = 1; i <=5; i++) {
    user  = new User();
    user.setId(i);
    user.setUsername("example" + i);
    user.setEmail("example " + i + "@example.com");
    users.add(user);
  }
  return userAssembler.toCollectionModel(users);
}
```

Lastly, we test it (this is the response body of a GET message to /users/test-users):

Spring – HATEOAS – Server side – Representation of a collection of resources states


```
{
  "_embedded": {
    "userList": [
```

```
        {
          "id": 1,
          ...
           "_links": {
              ...
           }
        },
        {
          "id": 2,
          ...
           "_links": {
              ...
           }
        },
        ...
      ]
    }
    "_links": {
     "self": {
       "href": "http://localhost:8080/users/{?salutation}",
       "templated": true
     }
  }
}
```

2.8.4.2. Client side

To consume HAL content with the Spring HATEOAS libraries, we can use the *Traverson* [66] class. This class allows us to navigate a REST API that produces HAL content just by knowing its base URI and the relations of the links. For example, if we were to request the user with id 1 from the previous example:

Spring – HATEOAS – Client side – Traverson usage

```
import java.net.URI;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.client.Hop;
import org.springframework.hateoas.client.Traverson;
...

Traverson traverson =
  // Base URI of the REST api
  new Traverson(URI.create("http://localhost:8080/"),
    MediaTypes.HAL_JSON);
  // As we cannot apply generic types to Class objects,
  // (EntityModel<User>.class is not valid syntax) we have to
  // create a ParameterizedTypeReference, to be able to decode
  // the response User
  ParameterizedTypeReference<EntityModel<User>>
  userTypeReference =
    new ParameterizedTypeReference<EntityModel<User>>() {};
  // The traverson will make a get request to the base URI, and
  // follow the relations from there
  EntityModel<User> userEntity = traverson
     // If a relation is templated, we can substitute its
```

```
     // parameters with this useful Hop builder method
    .follow(Hop.rel("user").withParameter("id", 1))
    .toObject(userTypeReference);
```

Note that, for the *Traverson* to be able to find the desired resources, a GET request to the base URI of our REST service has to return an *EntityModel* with links to the other resources in the API. In the previous example, we are supposing the base resource of the application contains a link as follows:

```
"user": {
   "href": "http://localhost:8080/users/{id}",
   "templated": true
 }
```

For further information on the use of a *Traverson*, refer to the *Client-side Support* section on the official documentation at [67].

Note that the *Traverson* only performs GET requests. For other types of requests, we can use a *RestTemplate* [68]. A useful guide to understand this class can be found at [69].

2.8.5. Thymeleaf

Thymeleaf is an XML [31] template engine. This is, it allows applying transformations to template files in order to display application data. As they describe it in their official documentation [70]:

"The main goal of Thymeleaf is to provide an elegant and well-formed way of creating templates. In order to achieve this, it is based on XML tags and attributes that define the execution of predefined logic on the *DOM (Document Object Model)*, instead of explicitly writing that logic as code inside the template."

To give an overview on the features of thymeleaf, let's provide a basic example template:

Thyemleaf – Template example

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title th:text="#{my-title}"></title>
</head>
<body>
  <h1 th:text="'User: ' + ${user.id}"></h1>
  <p th:if="${showUsername}" th:text="${user.username}"></p>
</body>
</html>
```

We can see we are using some special attributes that will be parsed by the Thymeleaf template engine. For example, `th:text` will set the text of the tagged element to be the result of parsing the inner thymeleaf standard expression. Or `th:if` will only include the tagged element if the inner expression evaluates to true.

The expressions that will be most commonly used for this project are:

`#{...}`: A message expression that will be evaluated and searched for in the configured message source according to the configured locale (refer to the "Using Texts" section in the official documentation).

`${...}`: A variable expression.

For example, `#{my-title}` will search for a message identified by “my-title”. And `${user.id}` will search for a JavaBean named “user” and get its property “id”.

For further information on the usage and configuration of Thymeleaf, refer to its official documentation at [70].

3. REQUIREMENTS

This chapter describes the requirements of this project. They have been directly taken from the doctor currently responsible for scheduling and managing shifts at the internal medicine department of the *Hospital Universitario Virgen Macarena (HUVM)*.

As this project is being developed, the only current requirements to the system are functional ones. They have already been explained in plain words in the introduction section. However, to be able to track them properly, we will describe them formally.

Note that, even if the scope of the project is to only solve the scheduling problem, this chapter will describe the requirements to solve all three problems. This will make it a useful guide to continue developing the system after this project is complete.

3.1. Actors

This section will define the different actors currently involved in the three stated problems:

Table 1: A-01 - Doctor

A-01	Doctor
Description	This actor will refer to any doctor whose shifts have to be scheduled.

Table 2: A-02 – Shift manager

A-02	Shift manager
Description	This actor refers to the person in charge of scheduling the shifts and assigning the pager to all doctors. This person is also responsible for registering changes of shifts.

Table 3: A-03 - Manager

A-03	Manager
Description	This actor refers to the doctor's manager. To the concerns of this project, this is just the person responsible for accepting or declining shift changes.

Table 4: A-04 - Secretary

A-04	Secretary
Description	To the concerns of this project, this actor refers to the person in charge of sending via email the schedule to all the doctors.

3.2. Use cases

3.2.1. Current situation

The following image represents the different procedures that currently need to be handled manually by the shift manager:

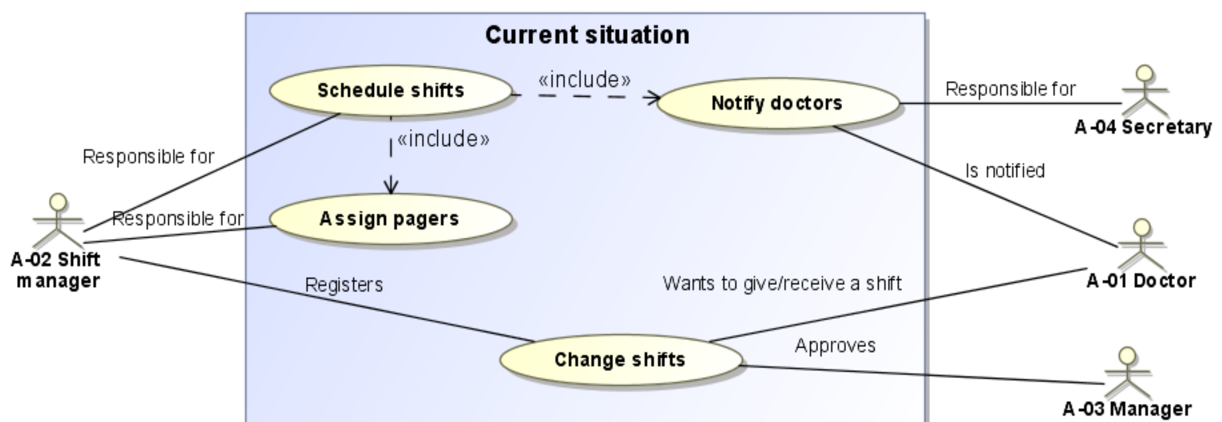


Figure 6: Current use cases

Note that currently, there is no software system responsible for implementing the uses cases in the diagram above. The diagram just represent the different activities that have to be performed.

Each of these procedures are as follows:

1. Scheduling of shifts:

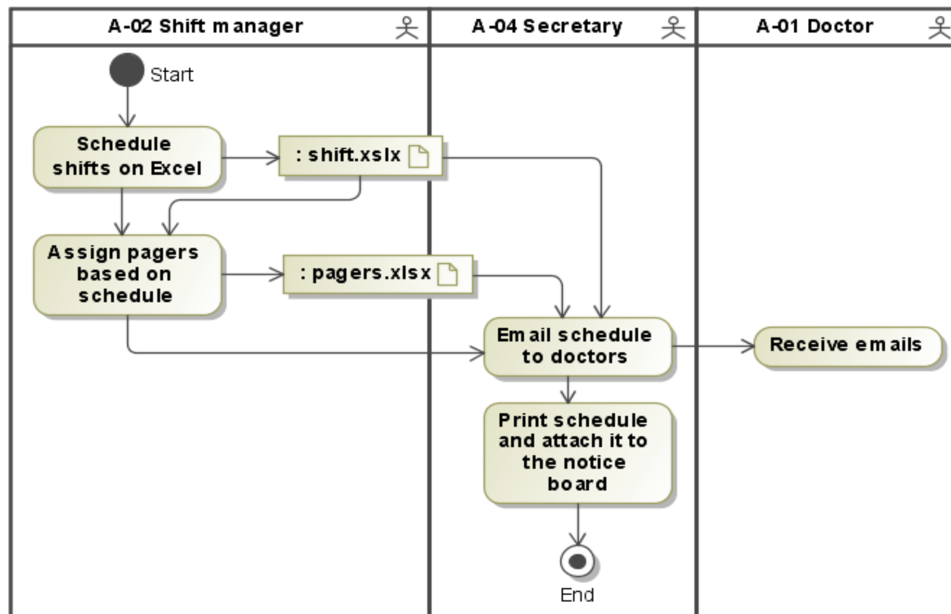


Figure 7: Current scheduling procedure

Note that, as mentioned before, there is no dedicated software being used for scheduling. This is the reason why all the participants in the current processes are actors.

2. Shift changes:

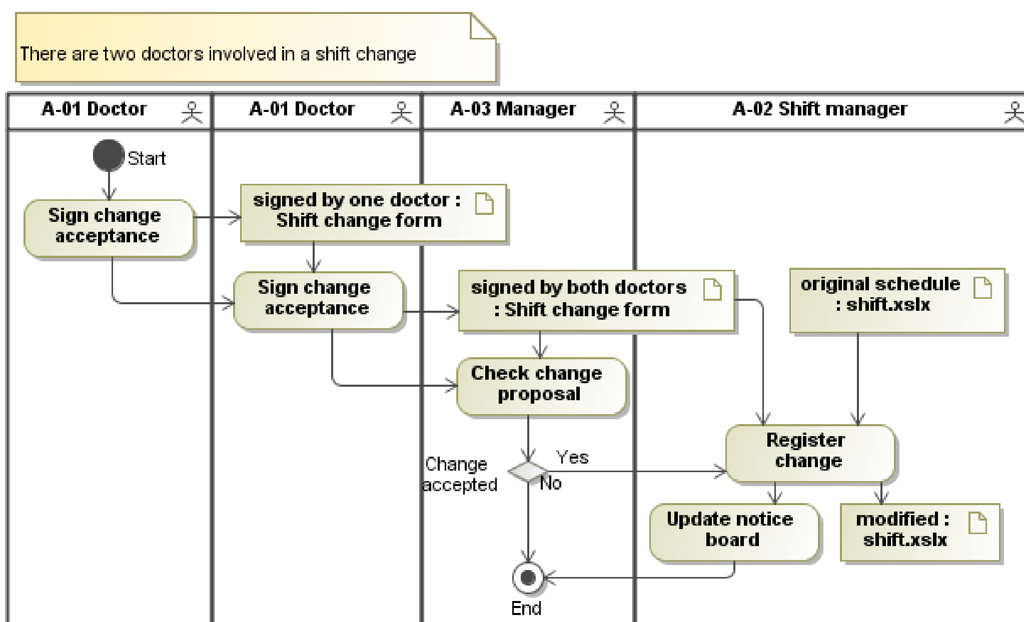


Figure 8: Current shift change procedure

3.2.2. Desired situation

The following image shows the desired use cases to be implemented by the system being designed:

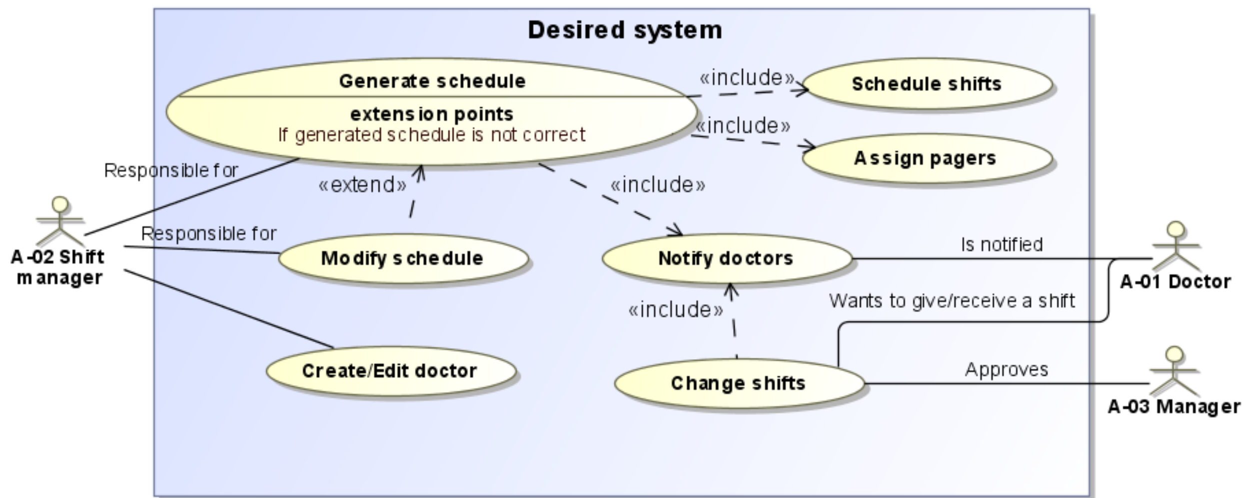


Figure 9: Desired use cases

Note the role of the secretary will no longer be needed, and the shift manager will no longer need to be responsible for registering shift changes. Also note that, as the design of the *Change Shifts* use case is not in the scope of this project, it has not been decomposed into other use cases.

From a very general perspective, the previous procedures should be as follows:

1. Create/Edit doctor:

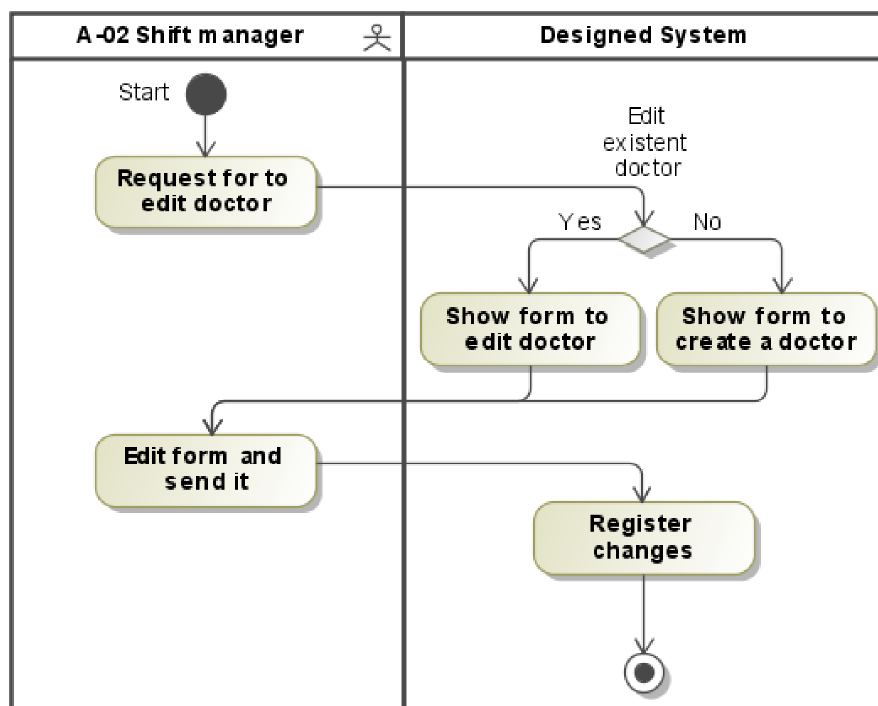


Figure 10: Desired create/edit doctor procedure

2. Generate schedule:

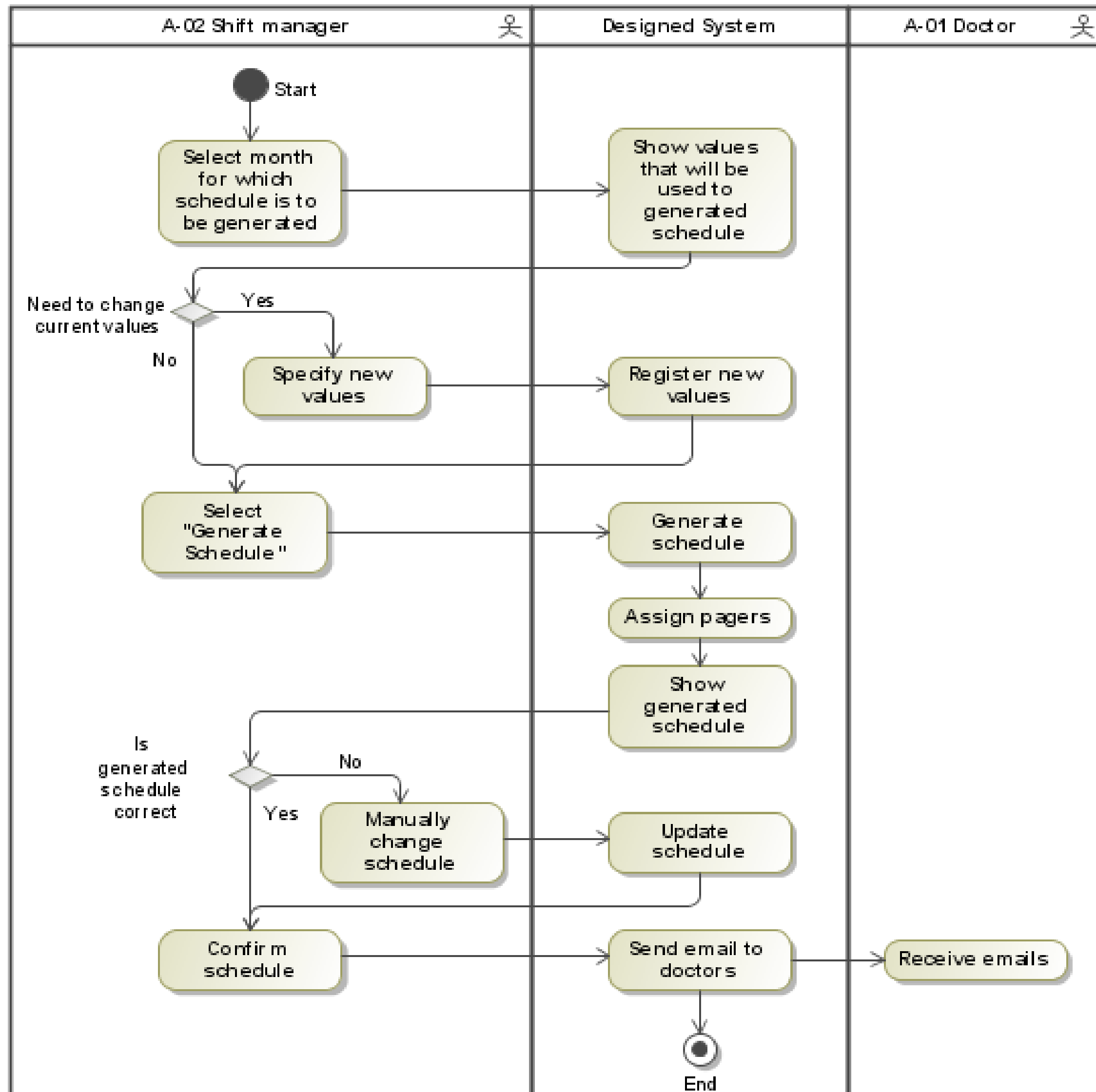


Figure 11: Desired scheduling procedure

Note the interaction between the shift manager and the system will be through a graphical user interface. The exact nature of this interface will be described in the chapter “Solution Designed”.

3. Change shift:

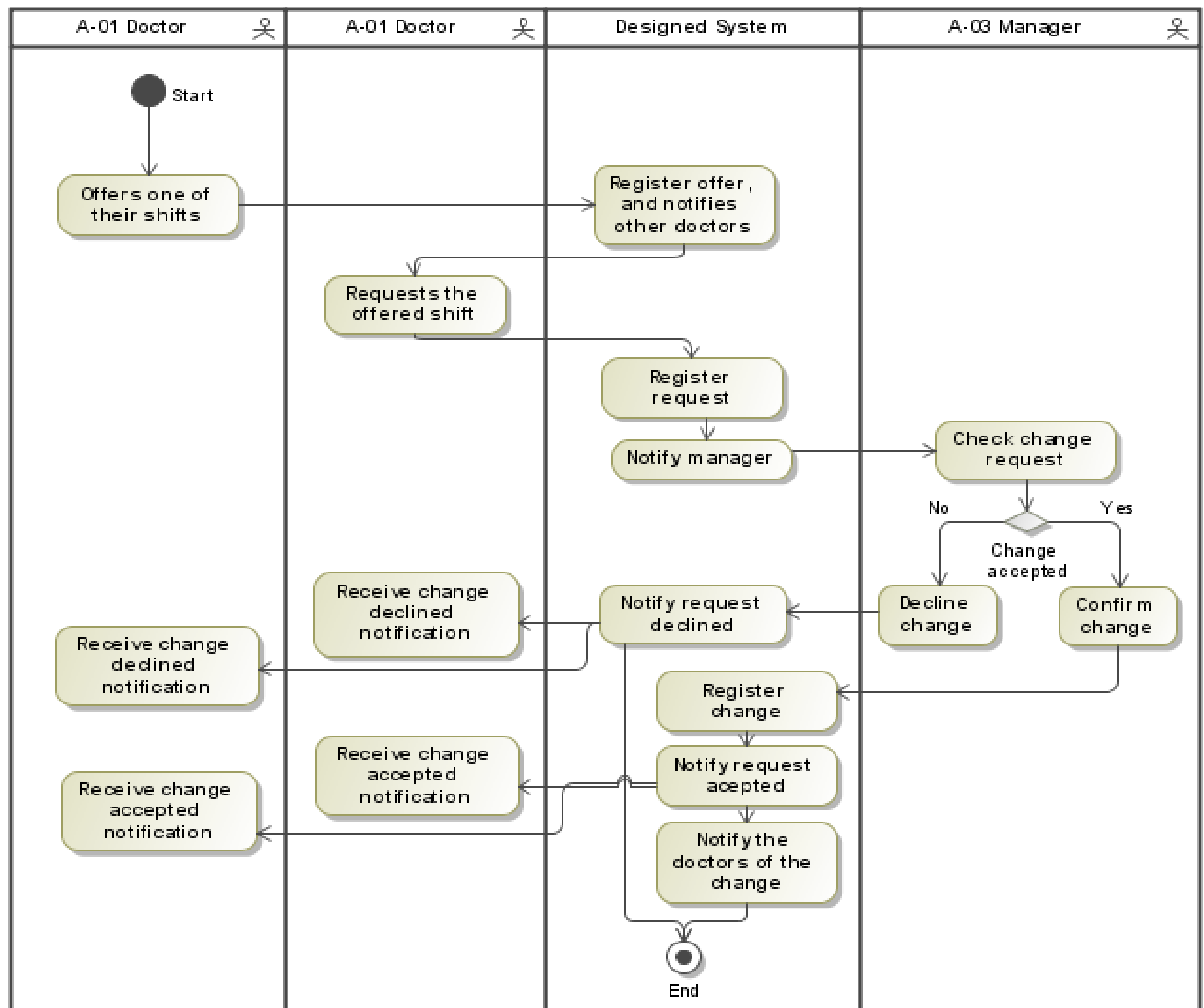


Figure 12: Desired shift change procedure

The previous diagram is just a suggestion on how the management problem could be solved. Still, note the role of the shift manager would no longer be needed, as the system would automatically register and notify the changes.

3.3. Behavioural requirements

Behavioural requirements will be named as BR.

Table 5: BR-01 – Doctor's identity

BR-01	Doctor's identity
Description	All doctors registered in the system have to be uniquely identifiable.

Before proceeding any further, and just as a reminder, we will explain the two main acronyms been used:

- CS: A Cyclic-shift (*Jornada Complementaria*) refers to the shifts that take place from 20.00 of a certain day, until 8.00 of the following day.
- NCS: A Non-Cyclic-Shift (*Continuidad Asistencial*) refers to the shifts that take place from 15.00 of a certain day, until 20.00 of the same day.

3.3.1. Scheduling problem

These requirements will be named as BR-SCH (Behavioural Requirement – Scheduling problem).

Table 6: BR-SCH-01 - Types of shifts

BR-SCH-01	Types of shifts
Description	There are two different types of shifts: CS and NCS. Most of the doctors can have both types of shifts. However, there are some doctors who only have either CS or NCS.
Note	The doctors who can have both CS and NCS belong to the internal medicine department.

Table 7: BR-SCH-02 - CS rate

BR-SCH-02	CS rate
Description	CS occur at a certain rate. This rate is the same one for all doctors.
Example	All CS are repeated every 10 days. This is, if doctor A had a CS the 1st of May, they will also have a CS the 11th and 21st of May.
Note	To calculate CSs, a reference date is needed. In the example above, we knew the doctor had a CS the 1st of May.

Table 8: BR-SCH-03 - NCS allowed days

BR-SCH-03	NCS allowed days
Description	NCS can only take place on working days. For example, there cannot be any NCS on a Saturday, a Sunday or on national Holidays.

Table 9: BR-SCH-04 - Minimum regular-shifts per doctor

BR-SCH-04	Minimum regular-shifts per doctor
Description	Each doctor has to have a minimum number of regular-shifts per month. This minimum can be different to each doctor.
Example	Doctor A has to have 3 or more regular-shifts per month.

Table 10: BR-SCH-05 - Maximum NCS per doctor

BR-SCH-05	Maximum NCS per doctor
Description	Each doctor has a maximum number of NCS they can have assigned each month. This maximum can be different to each doctor.

Table 11: BR-SCH-06 – Consultations per doctor

BR-SCH-06	Consultations per doctor
Description	Only some of the doctors who have NCS can have consultations. Specifically, the ones that do have consultations should have a certain number of them per month.
Note	This requirement has a lower priority than BR-SCH-05 - Maximum NCS per doctor. E.g. let's say doctor A has to have at least 3 regular-shifts per month, and has a maximum number of 5 NCS per month. Now, let's say doctor A does consultations, and has to have 2 per month. If a certain month, according to other restrictions (like BR-SCH-07 - A CS implies a regular-shift), doctor A has to have 4 regular-shifts, then they will only have 1 consultation.

Table 12: BR-SCH-07 - A CS implies a regular-shift

BR-SCH-07	A CS implies a regular-shift
Description	If a doctor has a CS a certain day, and they can have NCS, they have to have a regular-shift that day.

Table 13: BR-SCH-08 - Minimum number of regular-shifts and consultations per day

BR-SCH-08	Minimum number of regular-shifts and consultations per day
Description	Each working day, there is a minimum number of regular-shifts and a minimum number of consultations that have to be scheduled. These numbers may be different for each day.

Table 14: BR-SCH-09 - Shift preferences

BR-SCH-09	Shift preferences
Description	Each doctor may have certain preferences regarding the days they have their NCS.
Examples	• Doctor A would like to have their regular-shifts on Mondays or Tuesdays. • Doctor B would rather not have their regular-shifts on Thursdays. • Doctor A would like to have a regular-shift the 15th of May. • Doctor A would rather not have a regular-shift the 18th of May. • Doctor C would like to have their consultations on Wednesdays.

Table 15: BR-SCH-10 - History of schedules

BR-SCH-10	History of schedules
Description	The system has to keep a history of the schedules of each month. Only the final version of the schedule is needed. This is, after the shift manager makes changes (if any), and after applying shift changes (if any).

Table 16: BR-SCH-11 - NCS only when CS

BR-SCH-11	NCS only when CS
Description	Some doctors only have their NCS the same days they have their CS. This is, they will only have their NCSs assigned as per BR-SCH-07 - A CS implies a regular-shift; and no more shifts than that.

3.3.2. Pager assignment

These requirements will be named as BR-PG (Behavioural Requirements – Pager assignment).

Table 17: BR-PG-01 – Pager allowed days

BR-PG-01	Pager allowed days
Description	The pager can only be assigned on working days.

Table 18: BR-PG-02 - Doctors per day

BR-PG-02	Doctors per day
Description	Each allowed day (BR-PG-01 – Pager allowed days), the pager has to be assigned to exactly one doctor.

Table 19: BR-PG-03 – Pager allowed doctors

BR-PG-03	Pager allowed doctors
Description	Each day, the pager can only be assigned to a doctor having a CS that day.

Table 20: BR-PG-04 - Maximum assignments per month

BR-PG-04	Maximum assignments per month
Description	Doctors should only be assigned the pager once per month.
Note	The BR-PG-02 - Doctors per day requirement has a higher priority than this one. This is, if necessary, some doctors may have the pager assigned more than one day each month.

Table 21: BR-PG-05 - History of pagers

BR-PG-05	History of pagers
Description	A history of the assigned of pagers has to be kept for each month.

Table 22: BR-PG-06 - Second pager assignment order

BR-PG-06	Second pager assignment order
Description	By meeting requirement BR-PG-02 - Doctors per day, some doctors may need to have the pager assigned two days of a certain month. To decide which doctors will have the pager twice, we need the history of pager assignment (BR-PG-05 - History of pagers). Then, the doctors who had the pager assigned twice the longest time ago, should now have it assigned twice.
Example	Let’s say there are three doctors, A, B and C in the system. Now, let’s say doctor A had the pager assigned two days this month, and doctor B had it two days last month. Then, if a doctor needs to have the pager assigned two days the following month, it should be doctor C who has it.

3.3.3. Management problem

These requirements will be named as BR-MGMT (Behavioural Requirements – Management problem).

Table 23: BR-MGMT-01 - Doctor's notification

BR-MGMT-01	Doctor’s notification
Description	Doctors have to be notified after shifts and pagers of a month have been assigned. The notification has to be via email, but other means of notifications can also be accepted.
Example	Besides the email, doctors could be notified through an instant messaging app.

Table 24: BR- MGMT-02 - Awareness of changes

BR-MGMT-02	Awareness of changes
Description	All doctors have to be able to know the most updated version of the schedule. This is, the schedule of a month after shift changes have been applied.

Table 25: BR-MGMT-03 - Shift changes allowed

BR-MGMT-03	Shift changes allowed
Description	Doctors should be allowed to change their shifts with one another.
Example	Let’s say doctor A has a CS the 11th of June, and doctor B has a CS the 17th of June. Then, they should be allowed to change them so that doctor A does the CS the 17th of June and doctor B does the CS the 11th of June.

Table 26: BR-MGMT-04 - Shift changes authorised

BR-MGMT-04	Shift changes authorised
Description	Before a shift change is confirmed, it has to be authorised by the manager (A-03 - Manager).

Table 27: BR-MGMT-04 - Shift changes registered

BR-MGMT-04	Shift changes registered
Description	After a shift change has been confirmed, it has to be registered on its corresponding schedule.

3.4. Information requirements

From all the above, we can extract certain information requirements. These will be named as IR.

Table 28: IR-01 - Doctor

IR-01	Doctor
Information required	• Id: Integer (BR-01 – Doctor’s identity) • Full name: String • Email: String (BR-MGMT-01 - Doctor's notification) • Does CS: Boolean (BR-SCH-01 - Types of shifts) • Has regular-shifts only when CS: Boolean (BR-SCH-11 - NCS only when CS) • Min. Regular-shifts: Integer (BR-SCH-04 - Minimum regular-shifts per doctor) • Max. NCS: Integer (BR-SCH-05 - Maximum NCS per doctor) If set to zero, it represents this doctor does not have NCS (BR-SCH-01 - Types of shifts). • Num. Consultations: Integer (BR-SCH-06 – Consultations per doctor) If set to zero, it represents this doctor does not have consultations. • Ref. Date: Date Used to calculate the CSs of this doctor. (BR-SCH-02 - CS rate) • Wanted Regular-shifts: List of Strings [Optional] (BR-SCH-09 - Shift preferences) • Unwanted Regular-shifts: List of Strings [Optional] (BR-SCH-09 - Shift preferences) • Wanted Consultations: List of Strings [Optional] (BR-SCH-09 - Shift preferences) The Strings of these lists will be days of the week: “Monday”, “Tuesday”...

Table 29: IR-02 - Calendar

IR-02	Calendar
Information required	• Year and Month (Integers) • For each day of the month: ○ Whether this day is a working day: Boolean (BR-SCH-03 - NCS allowed days)

	○ Min. Number of Regular-shifts: Integer (BR-SCH-08 - Minimum number of regular-shifts and consultations per day) ○ Min. Number of Consultations: Integer (BR-SCH-08 - Minimum number of regular-shifts and consultations per day) ○ Wanted regular-shifts: List of Doctors [Optional] (BR-SCH-09 - Shift preferences) ○ Unwanted regular-shifts: List of Doctors [Optional] (BR-SCH-09 - Shift preferences) These lists of Doctors will contain the doctors who would /would not like to have a regular-shift this day. This allows specifying month specific shift preferences (BR-SCH-09 - Shift preferences).

Table 30: IR-03 - Schedule

IR-03	Schedule
Information required	• Year and Month (Integers) • For each day of the month: ○ CS: List of Doctors ○ Regular-shifts: List of Doctors [Optional] ○ Consultations: List of Doctors [Optional] ○ Pager: Doctor [Optional] (BR-PG-02 - Doctors per day)

4. SOLUTION DESIGNED

This chapter describes the systems in which the application has been divided to solve the scheduling problem. Each of these systems will have different responsibilities. Note that, as this project does not intend to solve neither the pager assignment nor the management problem, the solution is to be designed as flexible as possible. This will allow future additions and changes to be simpler and easier to perform.

4.1. Designed procedure

The use cases that will be designed and implemented are:

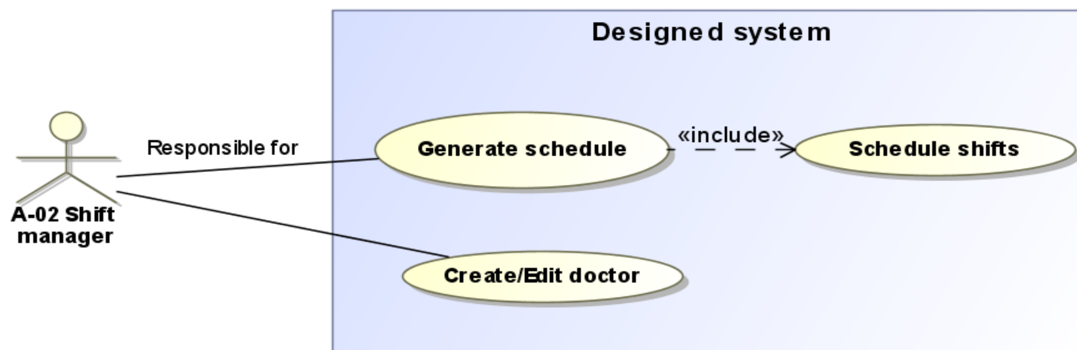


Figure 13: Designed use cases

And the procedure to generate the schedule will be as follows (The procedure to create/edit a doctor would be just like in Figure 10: Desired create/edit doctor procedure, page 40):

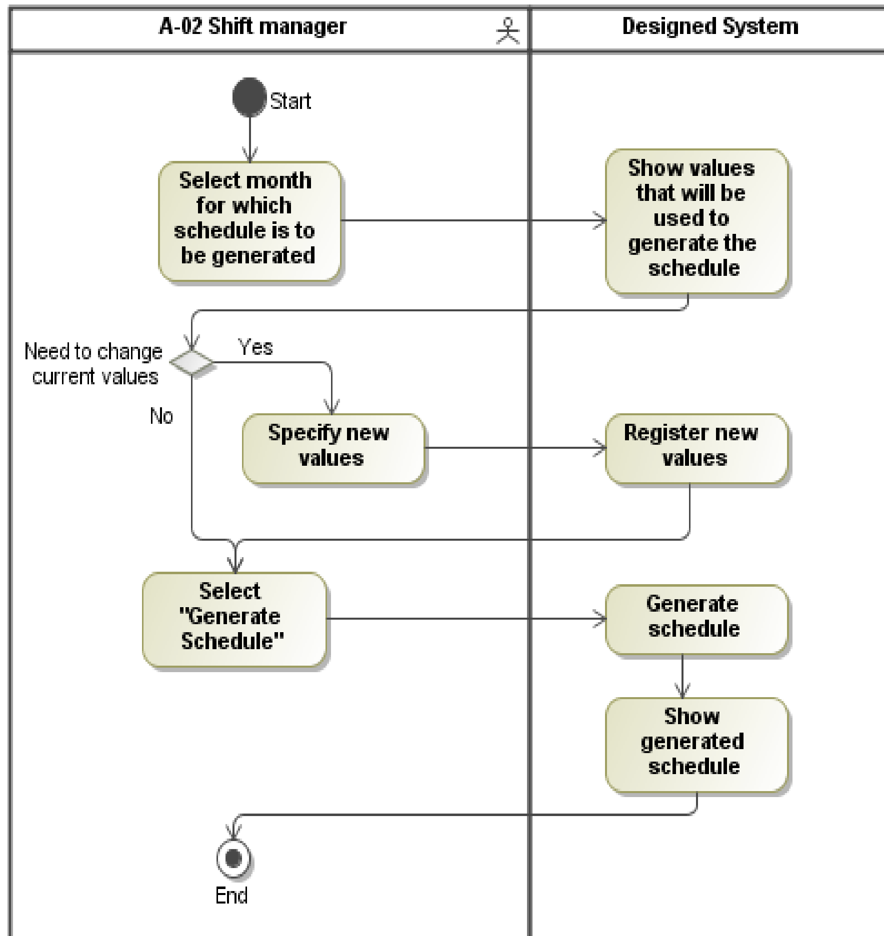


Figure 14: Designed scheduling procedure

Note that, as mentioned in the scope of the project, neither the assignment of pagers nor the management problem will be solved yet. For this reason, after a schedule is generated, it will still have to be sent manually via email to the doctors.

Also note that, currently, the edit schedule use case will not be implemented. This can be justified by keeping in mind the main goal of this project: presenting a first prototype on which to have feedback. Moreover, implementing this use case after the system is designed, should be as simple as adding a new form the web application, parse its data and send a request to the REST service (see the following section).

4.2. Division in subsystems

As it is a common practise, we will fProcirst divide the problem into different parts. Each of them with different responsibilities. Particularly, the problem will be divided into three subsystems:

- Scheduler: Responsible for scheduling shifts. This is, given the information IR-01 - Doctor and IR-02 - Calendar, it should produce the IR-03 - Schedule.
- REST service: Responsible for allowing CRUD (Create, Read, Update and Delete) operations on the information of the system.
- Web application: Responsible for providing an interface for the user to interact with the system.

The project will also make use of an already existent implementation of a DBMS (Database Management System) to store the data: MySQL [71].

To have an idea on how these systems cooperate, the following diagram briefly summarizes their interaction on the use cases being implemented:

1. Create/Edit doctor:

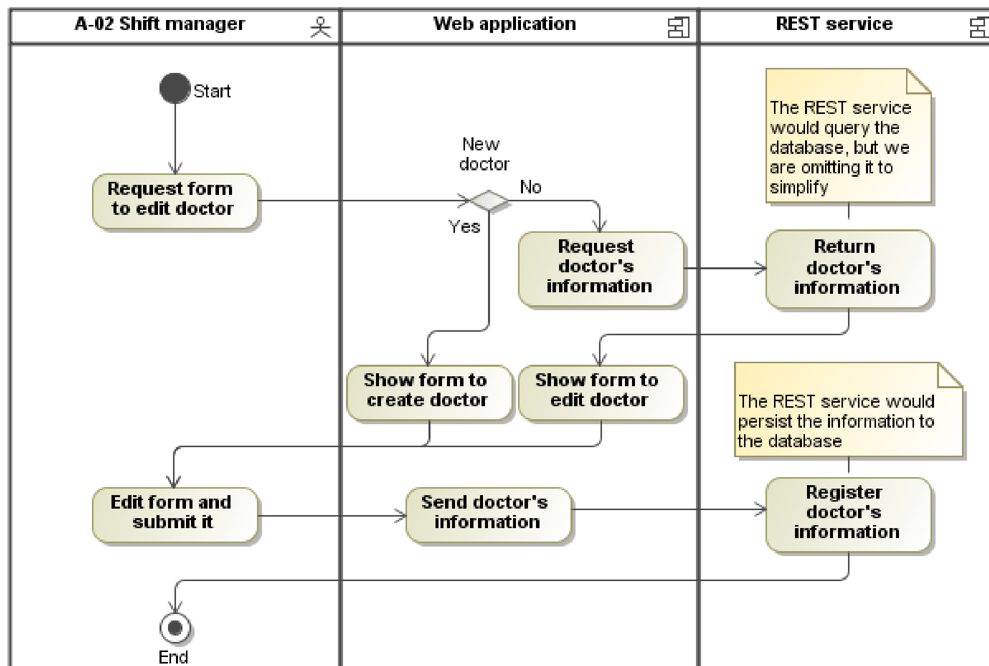


Figure 15: Designed create/edit doctor procedure - Systems communication

2. Generate schedule:

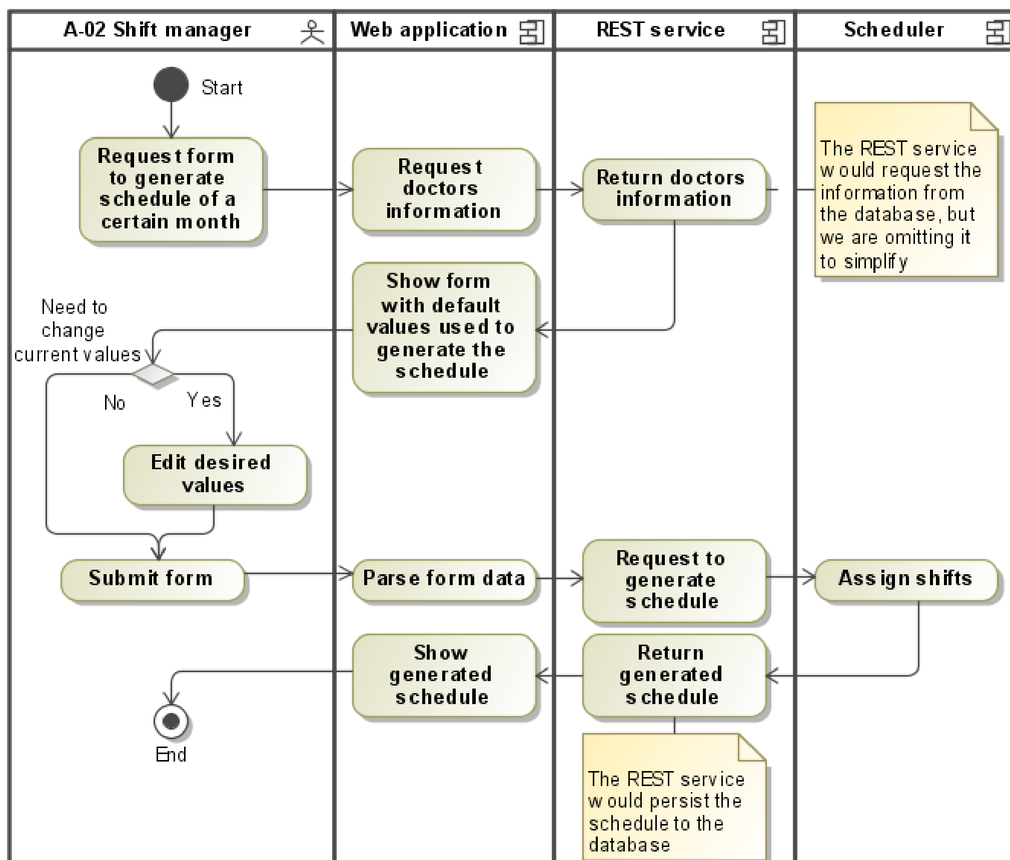


Figure 16: Designed scheduling procedure - Systems interaction

These three systems and their relations can be seen in the following deployment diagram:

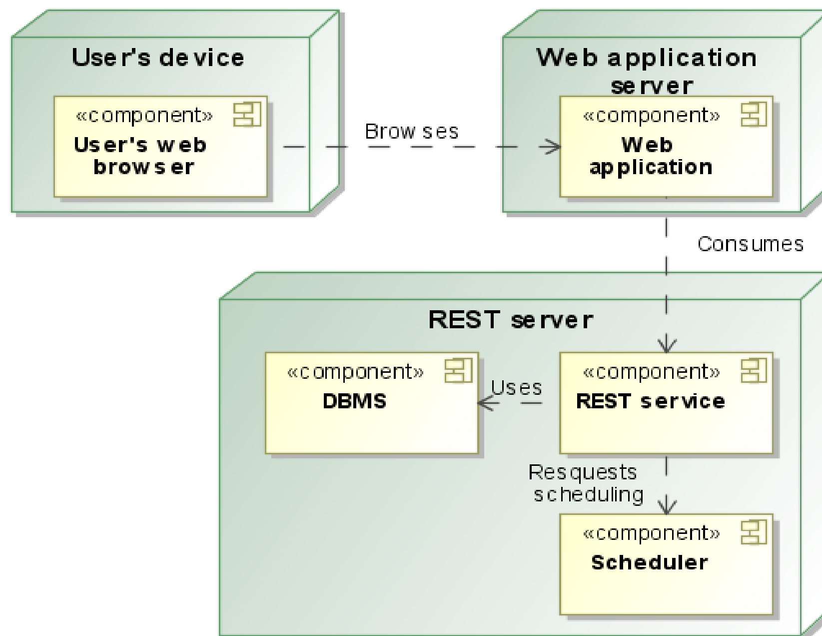


Figure 17: Deployment diagram

The user's device could be anything from a computer to a mobile phone. However, for the sake of simplicity, this project will target a computer's web browser. This is because the main goal of the project is the actual functionality or back-end, rather than the presentation to the user or front-end.

The web application server and the REST server have to be understood in the broadest possible way. Its four components could be deployed in a single machine, or they could be distributed over different servers. However, the deployment to be performed will have all components running in a single server. This is just to keep the deployment simple.

Note that, as mentioned before, the architecture of the service has been designed to be as flexible as possible. This is, for example, if a new presentation layer was to be developed (for example, an Android application), it would only have to consume the provided REST interface.

The design and implementation of the three mentioned subsystems will be explained in the following chapters.

4.3. Entity-Relation model

Before enumerating the resources, we will first analyse the information requirements and divide them into different entities. We will describe these entities and their relations using ER diagrams.

The model to meet IR-01 - Doctor is as follows:

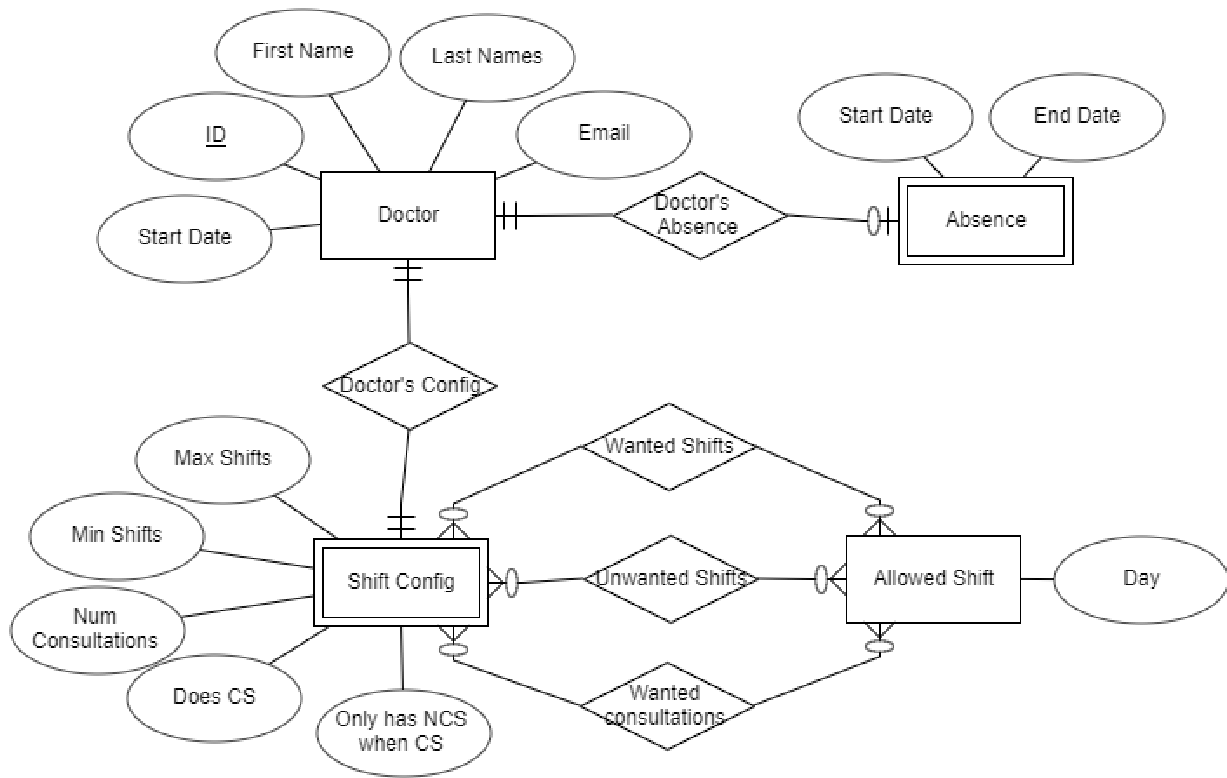


Figure 18: IR-01 Doctor - ER Model

Three concepts have been included to make the design more flexible. These are:

- The absence of a doctor: A doctor can be absent for a certain period of time due to an illness or to holidays.
- The status of a doctor: This will allow to mark doctors as deleted, without having to delete all of their information from the system. This is necessary to meet requirement BR-SCH-10 - History of schedules.
- The allowed shift entity: Although the BR-SCH-03 - NCS allowed days indicates that NCS can only be scheduled on working days, this entity would allow to easily change the system in case this requirement changed.

The model to meet IR-02 - Calendar is:

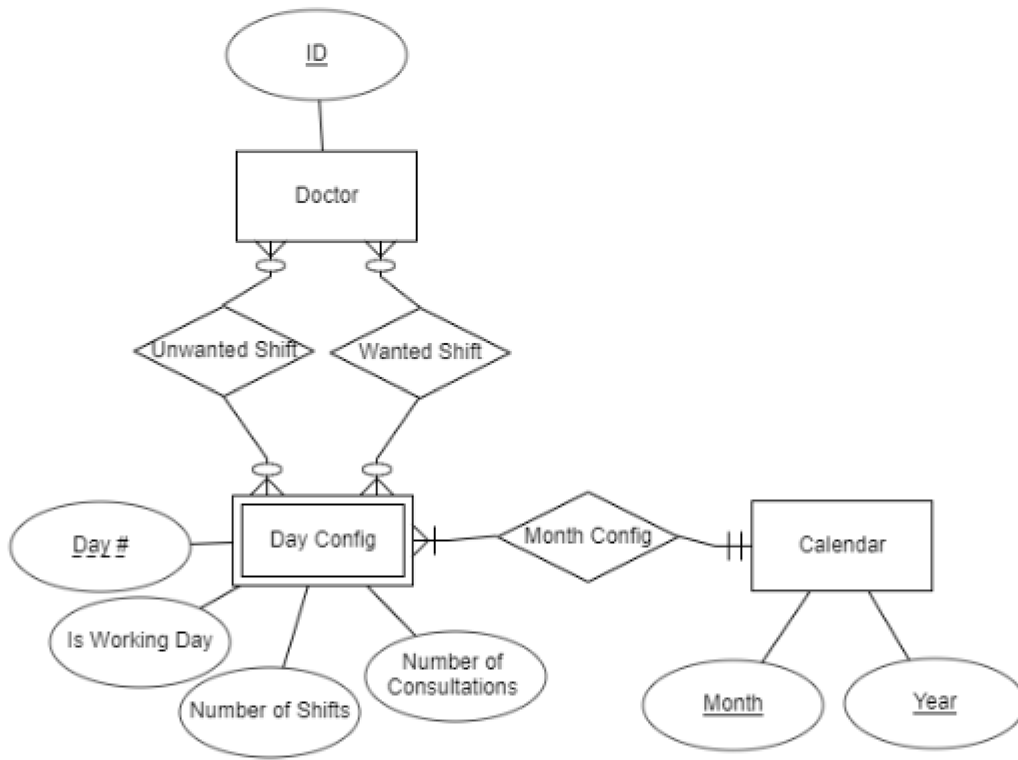


Figure 19: IR-02 - Calendar - ER Model

Note that the relations "Unwanted Shift" and "Wanted Shift" will allow creating month specific shift preferences (BR-SCH-09 - Shift preferences). E.g. some doctor would like to have a regular-shift the 15th of next month.

The modelling of IR-03 - Schedule is as follows:

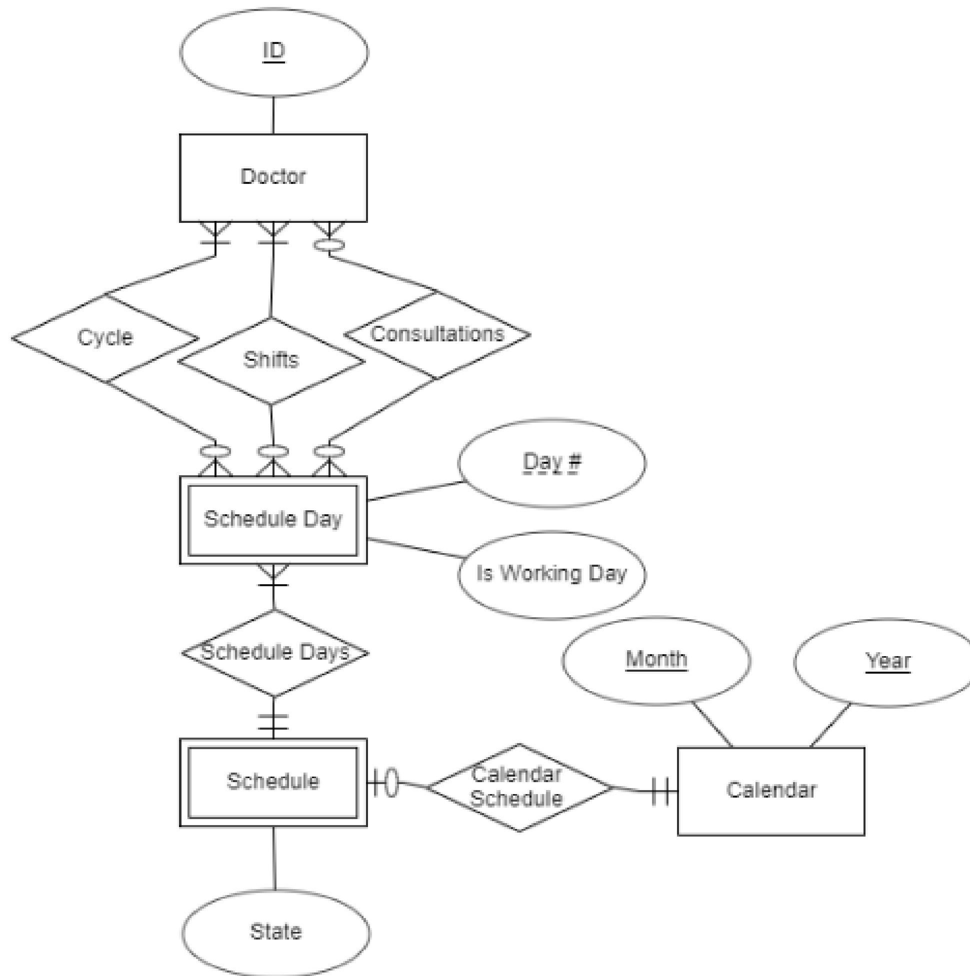


Figure 20: IR-03 - Schedule - ER Model

Note the *state* attribute of a Schedule represents whether it has been confirmed or not. A schedule is said to be confirmed if the shift manager considers it valid.

4.4. REST service

The REST service will be, as mentioned before, responsible for providing access to the resources states' representations. For this, we first have to define the Resources and the HTTP method that will allow accessing their states. Afterwards, we will present the system's design.

4.4.1. Service's resources

Before starting the definition of the resources, and to better understand the communication process with the REST service, we need to know the resources will be serialized according to the HAL specification [32].

Note: Throughout the following explanations, we will refer to "valid" representations of resources as a representation that contains all required properties, and the values of these properties are valid. For example, if the doctor status can only be "AVAILABLE" or "DELETED", any other value is considered invalid. If necessary, further explanations on valid values will be given in the properties of each resource.

Table 31: Root resource

Resource	root
URI	/guardians
Description	This resource will not have any properties, but just links to the service's resources. This will allow navigating through the whole API by just knowing its URI.
Actions	• GET Responses: ◦ 200 OK
Properties	-
Links	• self • doctors: This is a link to *doctors* resource. It has an optional query parameter `email` that allows searching by email. Hence, to retrieve all *doctors* we have to make a GET request to `DOMAIN/guardians/doctors/` and to search by email a GET request to `DOMAIN/guardians/doctors/?email=example@example.com` (Where "example@example.com" would be the email of the desired doctor). • doctor: A link to a *doctor* resource. It has a required path variable `doctorId`. For example, to GET *doctor* with id 1, we would make a request to `DOMAIN/guardians/doctors/1`. Refer to Table 33: Doctor resource to know all supported HTTP methods. • newDoctor: This is a link to create a new *doctor* resource. It has a required query parameter `startDate` that represents the date in which the new doctor would have their first CS. The date has to be represented according the ISO 8601 format. For example "`DOMAIN/guardians/doctors/?startDate=2020-06-14`". To create a *doctor*, we have to send a POST request to this link, and include the desired doctor representation in the request body. • shiftConfigs: A link to *shiftConfigs* resource. Refer to Table 34: ShiftConfigs resource to know the HTTP methods it supports. • shiftConfig: A link to a *shiftConfig* resource. It has a required path variable `doctorId` that represents the id of the doctor associated to this resource. For example, to GET the *shiftConfig* of the *doctor* with id 1, we would make a request to `DOMAIN/guardians/doctors/shift-configs/1`. To know all supported HTTP methods, refer to Table 35: ShiftConfig resource. • calendars: A link to *calendars* resource. Refer to Table 37: Calendars resource to know all supported HTTP methods. • calendar: A link to a *calendar* resource. It has a required path variable `yearMonth` representing the month and year of the corresponding calendar. They have to be represented according to the ISO 8601 standard. For example, we can GET the calendar of June 2020 by making a request to `DOMAIN/guardians/calendars/2020-06`. To know all supported HTTP methods, refer to Table 38: Calendar resource.

- schedules: A link to *schedules* resource. Refer to Table 39: Schedules resource to know all supported HTTP methods.
- schedule: A link to a *schedule* resource. It has a required path variable `yearMonth` representing the year and month of the corresponding schedule. They have to be represented according to the ISO 8601 standard. For example, to GET the schedule of June 2020 we would make a request to `DOMAIN/guardians/calendars/schedules/2020-06`. To know all supported HTTP methods, refer to Table 40: Schedule resource.
- allowedShifts: A link to *allowedShifts* resource. Refer to Table 36: AllowedShifts resource to know all supported HTTP methods.

Example[vi]

```
{
  “_links”: {
    “self”: {
      “href”: “DOMAIN/guardians/”
    },
    “doctors”: {
      “href”: “DOMAIN/guardians/doctors/{?email}”,
      “templated”: true
    },
    "doctor": {
      "href": "DOMAIN/guardians/doctors/{doctorId}",
      “templated”: true
    },
    “newDoctor”: {
      “href”: “DOMAIN/guardians/doctors/startDate={?startDate}”,
      “templated”: true
    },
    “shiftConfigs”: {
      “href”: “DOMAIN/guardians/doctors/shift-configs/”
    },
    "shiftConfig": {
      "href":
        "DOMAIN/guardians/doctors/shift-configs/{doctorId}",
      “templated”: true
    },
    "shiftConfigs": {
      "href": "DOMAIN/guardians/doctors/shift-configs/"
    },
    “calendars”: {
      “href”: “DOMAIN/guardians/calendars/”
    },
    “calendar”: {
      “href”: “DOMAIN/guardians/calendars/{yearMonth}”,
      “templated”: true
    },
    “schedules”: {
      “href”: “DOMAIN/guardians/calendars/schedules”
    },
    “schedule”: {
      “href”:
        “DOMAIN/guardians/calendars/schedules/{yearMonth}”,
      “templated”: true
    },
    “allowedShifts”: {
```

[vi] Note that *DOMAIN* will be used in the links to refer to the domain name of the server hosting the service.

	``` "href": "DOMAIN/guardians/allowed-shifts" } } } ```

Table 32: Doctors resource

Resource	doctors
URI	/guardians/doctors
Description	This resource will be a list of all *doctor* resources available in the service.
Actions	• GET Responses: ◦ 200 OK: The response body contains the list of *doctor*s. • POST: Used to create a new *doctor*. The message's body should contain all the mandatory properties of a *doctor* resource except the id (which will be assigned upon creation). See Table 33: Doctor resource. This request has a mandatory query parameter `startDate`, which will be a string representing a date in the ISO 8601 format. Responses: ◦ 200 OK: The response body contains the created *doctor* resource representation, including the id it has been assigned. ◦ 400 BAD REQUEST: If the *doctor* resource is not valid, or a *doctor* with the same given *email* already exists.
Properties	• doctors: The list of *doctor* resources
Links	• self • root: A link to GET the *root* resource.
Example	``` { "__embedded": { "doctors": [firstDoctorResource, # See the doctor resource secondDoctorResource, ...] }, "__links": { "self": { "href": "DOMAIN/guardians/doctors/{?email}", "templated": true ```

	`},`
	`"root": {`
	`"href": "DOMAIN/guardians/"`
	`}`
	`}`
	`}`

Table 33: Doctor resource

Resource	doctor
URI	/guardians/doctors/{doctor-id}
Description	This resource represents a single doctor in the system.
Actions	• GET Responses: ○ 200 OK: The response body contains the *doctor* resource representation. ○ 404 NOT FOUND: If given doctor-id does not exist. • PUT: Used to update all the information related to an already existent *doctor*. The request body should be the desired representation of the *doctor*. Responses: ○ 200 OK: The response body contains the updated *doctor* resource representation. ○ 400 BAD REQUEST: If the given *doctor* is not valid or the given *email* is already being used by a different *doctor*. ○ 403 FORBIDDEN: If the current *doctor* status is "DELETED". ○ 404 NOT FOUND: If given doctor-id does not exist. • DELETE: The *doctor* status will be changed to "DELETED". After this request, no client will be able to update its related information any more. And this doctor will not be scheduled any shifts. Note: The doctor's information will not be deleted as it is a requirement to be able to access all previous schedules. Responses: ○ 204 NO CONTENT: If the *doctor* was successfully marked as deleted. ○ 404 NOT FOUND: If the given doctor-id does not exist.
Properties	• id: An integer representing the unique identifier of the doctor.

	• firstName: A string.
	• lastNames: A string.
	• email: A string. Two doctors cannot have the same email.
	• status: A string indicating the doctor’s status. Its values can be “AVAILABLE” or “DELETED”.
	• absence: An object [Optional]. Represents a period in which the doctor will be absent. This property can be null or not present. It has two properties:
	  ○ start: A string representing the first day (included) in which the doctor will be absent. The date has to be represented according to the ISO 8601 format.
	  ○ end: A string representing the last day (included) in which the doctor will be absent. The date has to be represented according to the ISO 8601 format. The end date has to be after the start date. E.g. start=“2020-05-15” and end=“2020-05-20”.

Links	• self • doctors: This link has the same meaning and usage as the one in *root* resource. • shiftConfig: The associated shift configuration to this doctor • updateDoctor: An optional link. It will only be present if this *doctor* can be updated or deleted.
Example	(see below)

```
{
  “id”: 1,
  “firstName”: “John”,
  “lastNames”: “Smith”,
  “email”: “johnsmith@example.com”,
  “status”: “AVAILABLE”,
  “absence”: {
    “start”: “2020-06-20”,
    “end”: “2020-07-01”
  },
  “__links”: {
    “self”: {
      “href”: “DOMAIN/guardians/doctors/1”
    },
    “doctors”: {
      “href”: “DOMAIN/guardians/doctors/{?email}”,
      “templated”: true
    },
    “shiftConfig”: {
      “href”: “DOMAIN/guardians/doctors/shift-configs/1”
    },
    “updateDoctor”: {
      “href”: “DOMAIN/guardians/doctors/1”
    }
  }
}
```

Table 34: ShiftConfigs resource

<table>
<tr><td>Resource</td><td>shiftConfigs</td></tr>
<tr><td>URI</td><td>/guardians/doctors/shift-configs</td></tr>
<tr><td>Description</td><td>A list of all existing shiftConfig resources.</td></tr>
<tr><td>Actions</td><td>

GET
Responses:

200 OK: The list of existing shiftConfig resource representations.

POST: Create a shiftConfig of an already existent doctor. The value sent in the request should be the desired shiftConfig representation (and the doctorId property should be the desired doctor’s id). See Table 35: ShiftConfig resource.
Responses:

200 OK: The response body contains the created shiftConfig resource representation.
400 BAD REQUEST: If the given shiftConfig is not valid or the associated doctor already has a shiftConfig.
403 FORBIDDEN: If the associated doctor’s status is “DELETED”.
404 NOT FOUND: If the given doctorId does not exist.

</td></tr>
<tr><td>Properties</td><td>

shiftConfis: The list of shiftConfig resources.

</td></tr>
<tr><td>Links</td><td>

self
root: A link to GET the root resource.

</td></tr>
<tr><td>Example</td><td>

```
{
  “__embedded”: {
    “shiftConfigs”: [
      firstShiftConfigResource, # See the shiftConfig resource
      secondShiftConfigResource,
      ...
    ]
  },
  “__links”: {
    “self”: {
      “href”: “DOMAIN/guardians/doctors/shift-configs”
    },
    “root”: {
      “href”: “DOMAIN/guardians/”
    }
  }
}
```

</td></tr>
</table>

Table 35: ShiftConfig resource

Resource	shiftConfig
URI	/guardians/doctors/shift-configs/{doctor-id}
Description	This resource represents the information related to the shift configuration of a doctor.
Actions	• GET: Responses: ○ 200 OK: The response body contains the *shiftConfig* resource representation. ○ 404 NOT FOUND: If the requested *shiftConfig* does not exist. • PUT: Update an already existent *shiftConfig*. The body of the request should be the desired representation of the shift configuration. Responses: ○ 200 OK: The response body contains the updated *shiftConfig* resource representation. ○ 400 BAD REQUEST: If the given *shiftConfig* is not valid. ○ 403 FORBIDDEN: If the associated *doctor* is marked as deleted. ○ 404 NOT FOUND: If the *doctor* identified by *doctorId* does not exist or it does not already have an existent *shiftConfig*.
Properties	• doctorId: An integer. The id of the *doctor* associated to this shift configuration. • maxShifts: An integer greater than or equal to zero. The maximum number of shifts the associated doctor can have each month. • minShifts: An integer greater than or equal to zero. The minimum number of shifts the associated *doctor* has to have each month. *minShifts* has to be less than or equal to *maxShifts*. • numConsultations: An integer greater than or equal to zero. The number of consultations the associated *doctor* should have each month. • doesCycleShifts: A boolean. Whether the associated *doctor* can have CSs. • hasShiftsOnlyWhenCycleShifts: A boolean. Whether the associated *doctor* must only have NCSs the same days they have CSs. • unwantedShifts: A list of objects. • wantedShifts: A list of objects. • wantedConsultations: A list of objects. All *unwantedShifts*, *wantedShifts,* and *wantedConsultations* are lists of objects. These objects have to be representations of existent *allowedShift* resources.

<table>
<tr><td>Links</td><td>self
doctor: A link to the associated doctor resource.
shiftConfigs:This link has the same meaning and usage as the one in root resource.
allowedShifts: This link has the same meaning and usage as the one in root resource.</td></tr>
<tr><td>Example</td><td><pre>{
  "doctorId": 1,
  "maxShifts": 4,
  "minShifts": 3,
  "numConsultations": 0,
  "doesCycleShifts": true,
  "hasShiftsOnlyWhenCycleShifts": false,
  "unwantedShifts": [{“id”: 1, “shift”: “Monday”}],
  "wantedShifts": [],
  "wantedConsultations": [{“id”: 2, “shift”: “Tuesday”}],
  "_links": {
    "self": {
      "href": "DOMAIN/guardians/doctors/shift-configs/1"
    },
    "shiftConfigs": {
      "href": "DOMAIN/guardians/doctors/shift-configs/"
    },
    "doctor": {
      "href": "DOMAIN/guardians/doctors/1"
    },
    "allowedShifts": {
      "href": "DOMAIN/guardians/allowed-shifts/"
    }
  }
}</pre></td></tr>
</table>

Table 36: AllowedShifts resource

<table>
<tr><td>Resource</td><td>allowedShifts</td></tr>
<tr><td>URI</td><td>/guardians/allowed-shifts</td></tr>
<tr><td>Description</td><td>This resource will represent the days in which doctors can have NCSs.</td></tr>
<tr><td>Actions</td><td>GET
Responses:
200 OK: The response body contains the list of allowed shifts.</td></tr>
<tr><td>Properties</td><td>allowedShifts: The list of all allowed shifts.</td></tr>
<tr><td>Links</td><td>self</td></tr>
</table>

<table>
<tr><td></td><td>root: A link to GET the root resource.</td></tr>
<tr><td>Example</td><td><pre>{
  "__embedded": {
    "allowedShifts": [
      {
        "id": 1,
        "shift": "Monday"
      },
      {
        "id": 2,
        "shift": "Tuesday"
      },
      ...
    ]
  },
  "__links": {
    "self": {
      "href": "DOMAIN/guardians/allowed-shifts"
    },
    "root": {
      "href": "DOMAIN/guardians/"
    }
  }
}</pre></td></tr>
</table>

Table 37: Calendars resource

<table>
<tr><td>Resource</td><td>calendars</td></tr>
<tr><td>URI</td><td>/guardians/calendars</td></tr>
<tr><td>Description</td><td>This resource will represent a summary of all the existent calendar resources in the system.</td></tr>
<tr><td>Actions</td><td>GET

Responses:
200 OK: The response body contains the list of calendar resource representations.
POST: Create a new calendar resource. The body of the request should be the desired calendar representation. See Table 38: Calendar resource.

Responses:
200 OK: The response body contains the created calendar resource representation.
400 BAD REQUEST: If the given calendar is not valid or a calendar for the given month and year already exists.</td></tr>
<tr><td>Properties</td><td>calendars: The list with all existent calendar resources. Note these will not be the</td></tr>
</table>

	complete representations of the calendars but just a summarized representation (See the example below).
Links	• self • root: A link to GET the *root* resource.

Example	<pre>{
	  "__embedded": {
	    "calendars": [
	      {"year": 2020, "month":4, "__links": {...}},
	       {"year": 2020, "month":5, "__links": {...}},
	      ...
	    ]
	  },
	  "__links": {
	    "self": {
	      "href": "DOMAIN/guardians/calendars"
	    },
	    "root": {
	      "href": "DOMAIN/guardians/"
	    }
	  }
	}</pre>

Table 38: Calendar resource

Resource	calendar
URI	/guardians/calendars/{yyyy-mm}
Description	This resource will represent the configuration for month *mm* and year *yyyy*.
Actions	• GET Responses: ○ 200 OK: The response body contains the *calendar* resource representation. ○ 404 NOT FOUND: If no calendar was found for the given *month* and *year*. • PUT: Update an already existent *calendar*. The body of this request should be the representation of the desired *calendar*. Responses: ○ 200 OK: The response body contains the updated *calendar* resource representation. ○ 400 BAD REQUEST: If there given *calendar* is not valid. ○ 404 NOT FOUND: If no calendar was found for the given *month* and *year*.

<table>
<tr>
<td>Properties</td>
<td>

month: Integer between 1 and 12.
year: Integer greater than or equal to 1970.
dayConfigurations: List of objects. Each of these objects will contain the following attributes:

day: Integer between 1 and 31.
isWorkingDay: Boolean.
numShifts: Integer greater than or equal to zero.
numConsultations: Integer greater than or equal to zero.
wantedShifts: List of doctors [Optional]
unwantedShifts: List of doctors [Optional]

This list has to contain exactly all days of the desired month. For example, if the month is May, this list should contain one entry for day=1, another one for day=2, …, until day=31.

</td>
</tr>
<tr>
<td>Links</td>
<td>

self
calendars: This link has the same meaning and usage as the one in root resource.
schedule: A link to the associated schedule resource.

</td>
</tr>
<tr>
<td>Example</td>
<td>

```
{
  "month": 5,
  "year": 2020,
  "dayConfigurations": [
    {
      "day": 1,
      "isWorkingDay": true,
      "numShifts": 2,
      "numConsultations": 0,
      "unwantedShifts": [{“id”: 1}],
      "wantedShifts": []
    },
    {
      "day": 2,
      "isWorkingDay": false,
      "numShifts": 0,
      "numConsultations": 0,
      "unwantedShifts": [],
      "wantedShifts": []
    },
    ...
  ],
  "_links": {
    "self": {
      "href": "DOMAIN/guardians/calendars/2020-05"
    },
    "calendars": {
      "href": "DOMAIN/guardians/calendars/"
    },
```

</td>
</tr>
</table>

<table>
<tr><td></td><td><pre>   "schedule": {
      "href": "DOMAIN/guardians/calendars/schedules/2020-05"
    }
  }
}</pre></td></tr>
</table>

Table 39: Schedules resource

<table>
<tr><td>Resource</td><td>schedules</td></tr>
<tr><td>URI</td><td>/guardians/calendars/schedules</td></tr>
<tr><td>Description</td><td>This resource represents a summarized list of all existent Schedule resources.

By summarized, we understand that not all the information related to the Schedule resources will be included in the list (See the example below).</td></tr>
<tr><td>Actions</td><td>GET

Responses:
200 OK: The response body contains the list of schedule resource representations.</td></tr>
<tr><td>Properties</td><td>A list of schedules</td></tr>
<tr><td>Links</td><td>self
root: A link to GET the root resource.</td></tr>
<tr><td>Example</td><td><pre>{
  “__embedded”: {
    “schedules”: [
      {
        "month": 4,
        "year": 2020,
        "status": "PENDING_CONFIRMATION",
        "_links": {…}
      },
      {
        "month": 5,
        "year": 2020,
        "status": "CONFIRMED",
        "_links": {…}
      },
      ...
    ]
  },
  “__links”: {
    “self”: {
      “href”: “DOMAIN/guardians/calendars/schedules”
    },
    “root”: {</pre></td></tr>
</table>

<table>
<tr><td></td><td><pre>
        “href”: “DOMAIN/guardians/”
      }
    }
}
</pre></td></tr>
</table>

Table 40: Schedule resource

<table>
<tr><td>Resource</td><td>schedule</td></tr>
<tr><td>URI</td><td>/guardians/calendars/schedules/{yyyy-mm}</td></tr>
<tr><td>Description</td><td>This resource will represent the schedule for month mm and year yyyy.</td></tr>
<tr><td>Actions</td><td>

GET

Responses:

200 OK: The response body contains the schedule resource representation.
404 NOT FOUND: If the calendar of month and year does not exist. Note that, if the calendar exists, even if the schedule is not generated yet, a schedule resource representation will be returned. In this case, the state of the schedule will be NOT_CREATED.

POST: This request, with an empty body, will start the generation of the schedule. The corresponding Calendar resource of month mm and year yyyy has to exist.

Responses:

202 ACCEPTED: The response body will be empty.
400 BAD REQUEST: If a schedule for the given month and year already exists.
404 NOT FOUND: If the associated calendar resource does not exist.

PUT: Update the schedule. The desired representation of the schedule should be sent in the request body. The schedule can only be updated if it has not yet been confirmed.

Responses:

200 OK: The response body will contain the updated schedule resource representation.
400 BAD REQUEST: If the given schedule representation is not valid.
404 NOT FOUND: If the given schedule resource does not exist.
409 CONFLICT: If the given schedule representation has a different status than the current schedule, and the status transition is not allowed. For example, if the current schedule is confirmed or being generated, it cannot be updated.

A valid transition would be from the PENDING_CONFIRMATION status to the CONFIRMED status. Another valid transition would be from the GENERATION_ERROR status to the PENDING_CONFIRMATION status.

</td></tr>
</table>

	• DELETE: Delete the schedule if it has not yet been confirmed. Responses: ◦ 204 NO CONTENT: If the *schedule* was deleted correctly. Note this status code will also be returned if the schedule is NOT_CREATED. ◦ 403 FORBIDDEN: If the *schedule* has been confirmed. ◦ 404 NOT FOUND: If the *schedule* of *year -month* does not exist.
Properties	• month: Integer between 1 and 12. • year: Integer greater than or equal to 1970. • status: String. Can be "NOT_CREATED", "BEING_GENERATED", "PENDING_CONFIRMATION", "CONFIRMED" or "GENERATION_ERROR". • days: List of objects. Each object will have the following properties: ◦ day: Integer between 1 and 31. ◦ isWorkingDay: Boolean ◦ cycle: List of Doctors ◦ shifts: List of Doctors [if not isWorkingDay, this has to be an empty list] ◦ consultations: List of Doctors [Optional]
Links	• self • calendar: A link to the associated *calendar* resource. • confirm: A PUT to this link will transition the schedule to the CONFIRMED status. This link will only be present when this transition is allowed.

Example	`{`
	`  "month": 5,`
	`  "year": 2020,`
	`  "status": "PENDING_CONFIRMATION",`
	`  "days": [`
	`    {`
	`      "day": 1,`
	`      "isWorkingDay": true,`
	`      "cycle": [doctorResource1, doctorResource2],`
	`      "shifts": [doctorResource1, doctorResource3],`
	`      "consultations": [doctorResource5]`
	`    },`
	`    {`
	`      "day": 2,`
	`      "isWorkingDay": false,`
	`      "cycle": [doctorResource3, doctorResource4],`
	`      "shifts": [],`
	`      "consultations": []`
	`    },`

	...], "_links": { "self": { "href": "DOMAIN/guardians/calendars/schedules/2020-05/" }, "calendar": { "href": "DOMAIN/guardians/calendars/2020-05" }, "confirm": { "href": "DOMAIN/guardians/calendars/schedules/2020-05/confirmed" } } }

4.4.2. Structural design

First, we will present the complete class diagram of the application:

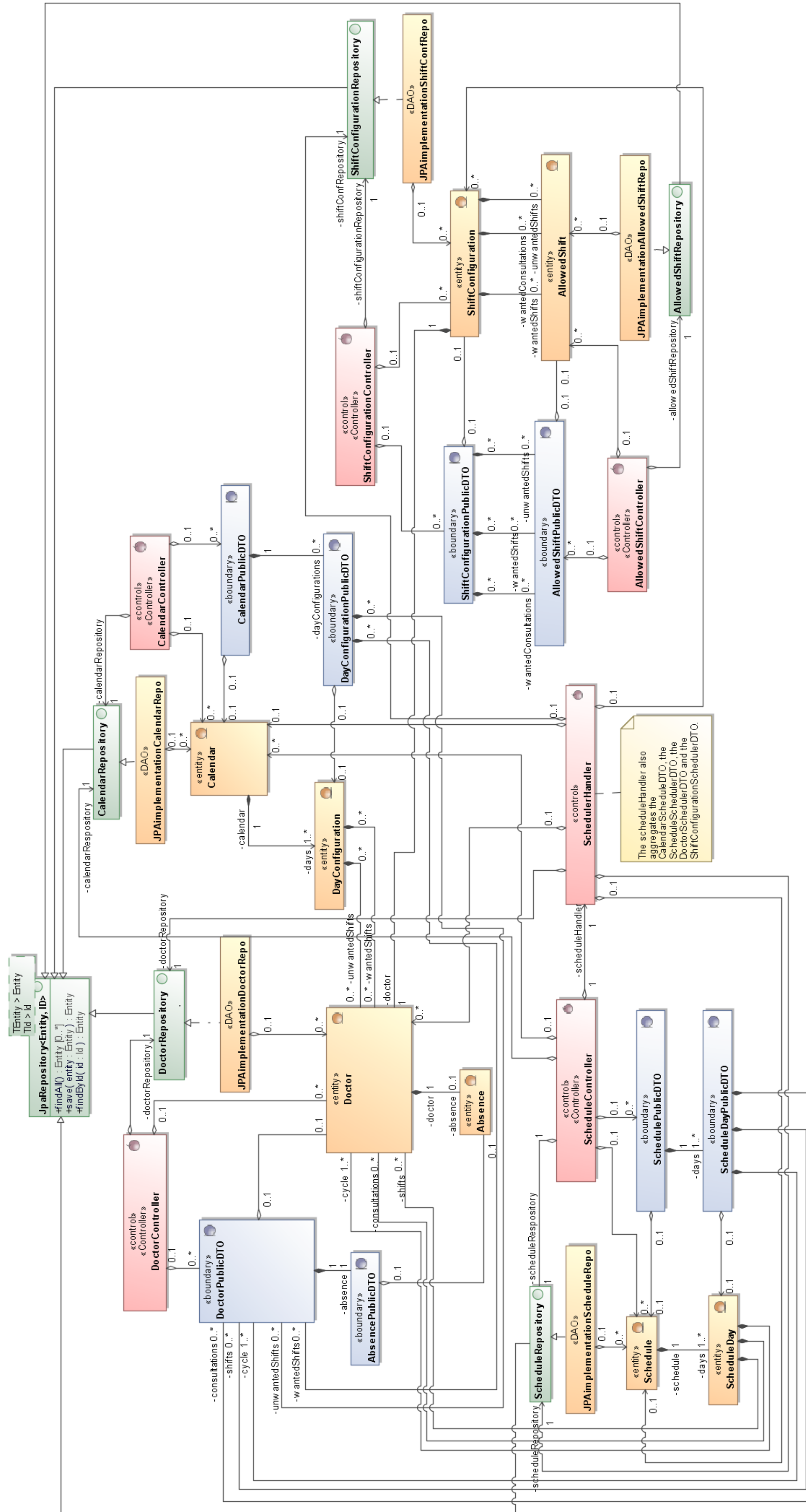


Figure 21: REST Service - Overview class diagram

As the previous diagram might only be readable by zooming in, we show one of its main controller's relations:

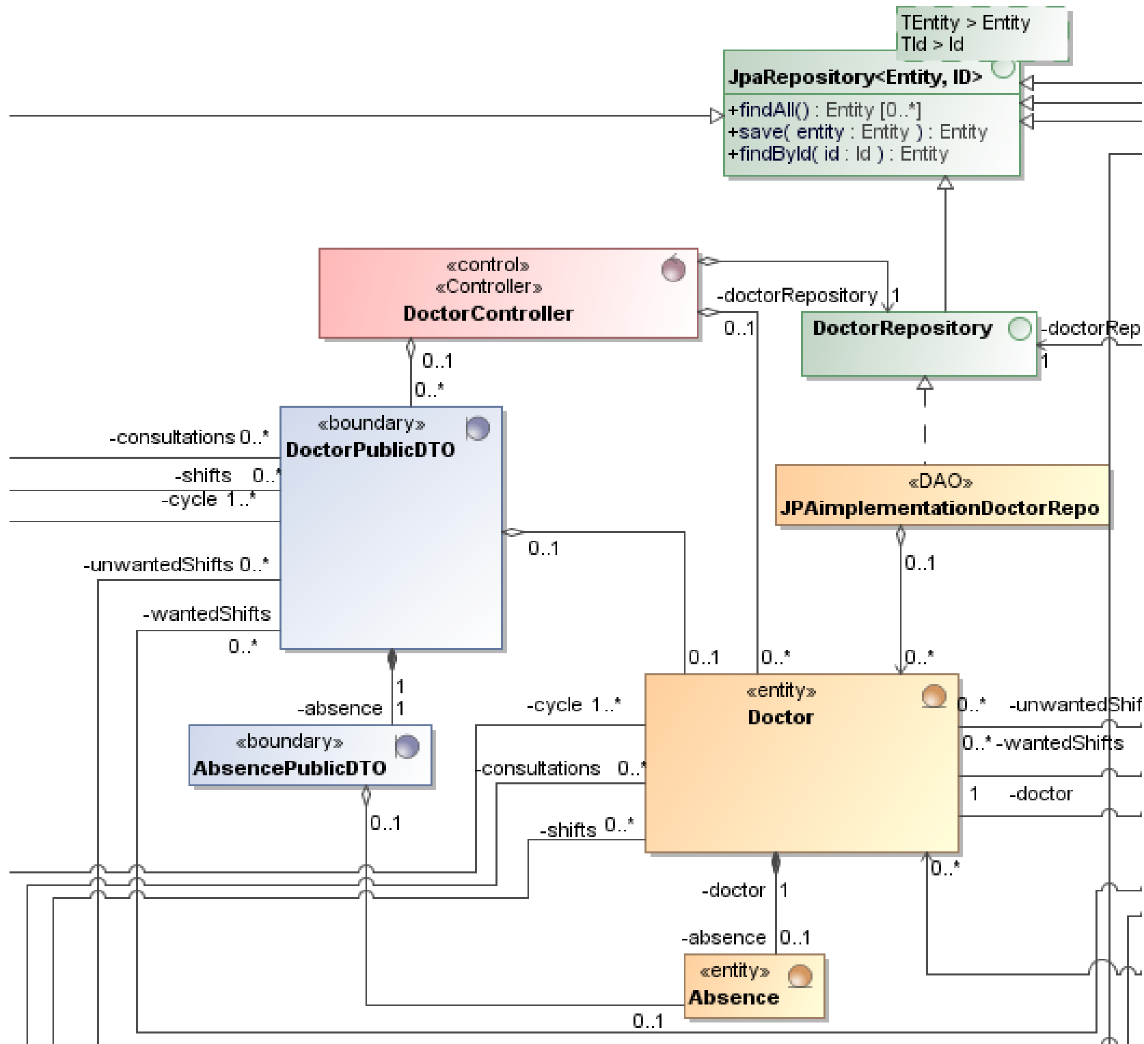


Figure 22: REST Service - Doctor controller overview

The REST service is made up of five different controllers (DoctorController, CalendarController, ShiftConfigurationController, ScheduleController and AllowedShiftController). All of these, have a similar structure as the one presented above: Each controller aggregates their corresponding repository (DAO), their associated entities, and the DTOs associated to these entities.

Note that classes' properties and methods have been hidden to summarize the diagram (they will be shown in the following sections). Moreover, to keep the diagram simpler, HATEOAS classes and validation classes have not been represented. However, these have been modelled just as it has been explained in 2.8.4 Spring HATEOAS and 2.8.3 Spring Data JPA respectively.

4.4.2.1. Model classes

The model classes will contain the information related to the resources. E.g. the information related to a doctor. They will also contain the different constraints that may apply to them. A restriction, for example, is that the doctor's emails have to be valid. E.g. "example@example.com" will be considered a valid email, whereas "example.com" will not.

We will begin with a class diagram describing the D*octor*, *Absence*, S*hiftConfiguration* and *AllowedShift* classes:

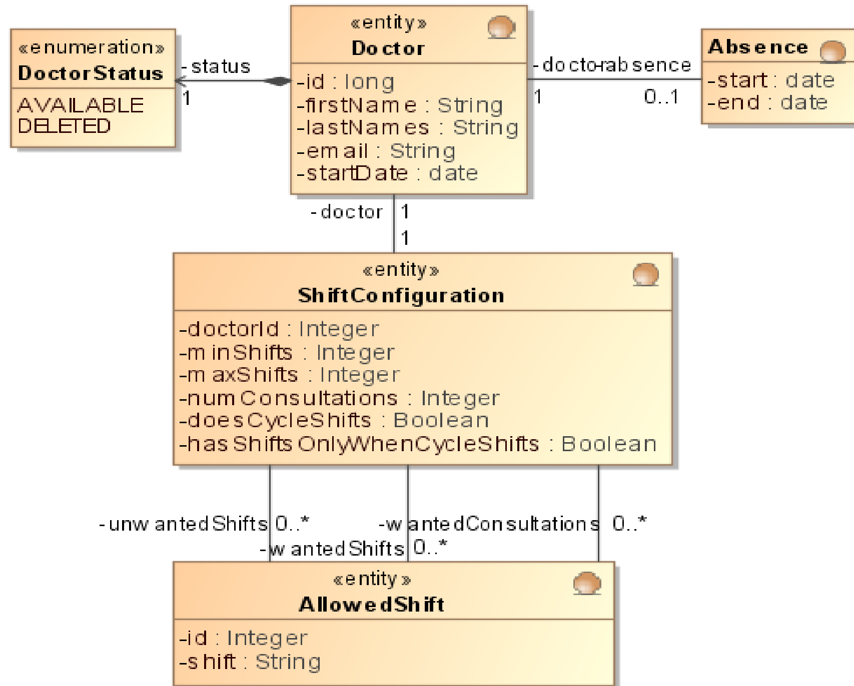


Figure 23: REST Service - Model class diagram - Doctor, Absence, DoctorStatus, ShiftConfiguration and AllowedShift

The restrictions that apply to these classes are:

- *Doctor*: The *email* has to be a valid email.
- *Absence*: The start date has to be before the end date. E.g. *start*="2020-05-20", *end*="2020-06-15" would be valid values, but *start*="2020-06-20", *end*="2020-05-15" would not.
- *ShiftConfiguration*: *minShifts* has to be less than or equal to *maxShifts*. m*inShifts*, *maxShifts*, and *numConsultations* have to be greater than or equal to zero.

Now, we will introduce the class diagram describing the *Calendar* and *DayConfiguration* classes:

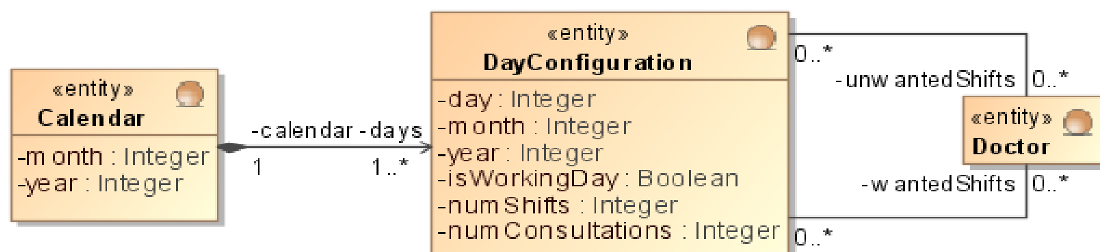


Figure 24: REST Service - Model class diagram - Calendar and DayConfiguration

Note that *DayConfiguration* is the class being used to represent the list of days associated to a *Calendar*, as described in IR-02 - Calendar. Also, note that the date of a *DayConfiguration* is represented as three different fields. This has to do with JPA, as it is a simple way for *DayConfiguration* to share the primary key of the *Calendar*.

The restrictions that apply to these classes are:

- *Calendar*: The *days* list has to contain one and exactly one *DayConfiguration* for each day of the *month*

and *year*. E.g. if *month*=5 and *year*=2020, then *days* has to contain the days from 1 to 31. Leap years have to be taken into account. Moreover, *month* has to represent a valid month number. E.g. *month*=13 or *month*=0 would not be valid.

- *DayConfiguration*: The values *day*, *month* and *year* have to correspond to an existing date. E.g. *day*=15, *month*=5, *year*=2020 would be valid, but day=31, *month*=6, *year*=2020 would not (June has only 30 days).

Lastly, the class diagram describing the *Schedule* class is as follows:

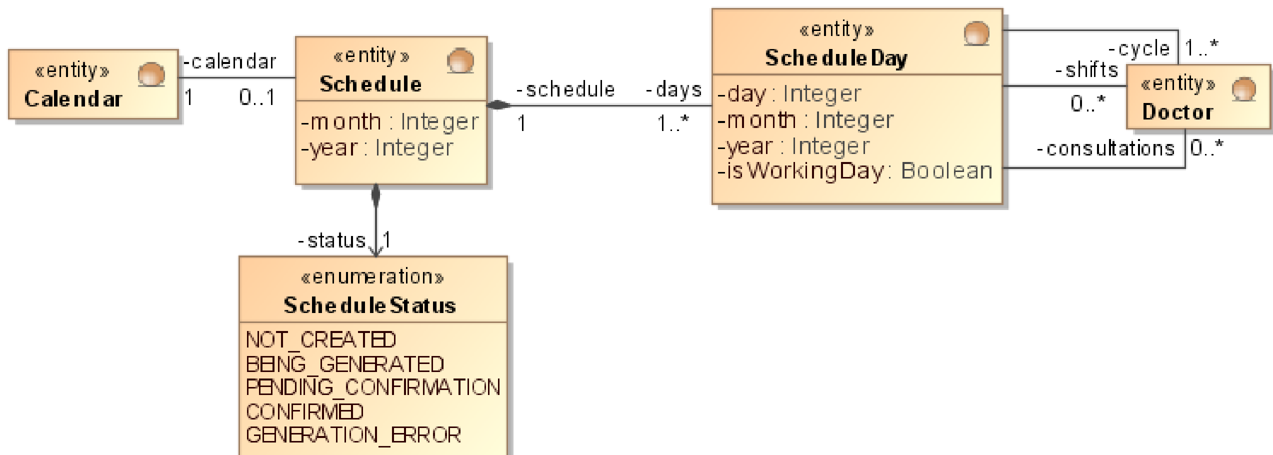


Figure 25: REST Service - Model class diagram - Schedule, ScheduleStatus, ScheduleDay

For the same reason mentioned in the *Calendar* class diagram, the date of *ScheduleDay* is represented as three different fields. Note the names of the relations between ScheduleDay and Doctor from the ER model have been mapped as follows:

- CS → cycle
- Regular-shifts → shifts
- Consultations → consultations

The same restrictions that applied to *Calendar* and *DayConfiguration* apply to *Schedule* and *ScheduleDay* respectively. There is only one difference: *Schedule* cannot contain any *days* if it's status is NOT_CREATED, BEING_GENERATED or GENERATION_ERROR; and has to contain all *days* if it's status is PENDING_CONFIRMATION or CONFIRMED.

Note that each model class will have the corresponding getters and setters for all of their fields.

4.4.2.2. View classes

The views of the REST service will be the classes serialized into HAL [32]. For this reason, and to remove any serialization logic from the model classes, we will apply the DTO pattern. Specifically, as we will have two different interfaces (The public REST interface, and the one used to communicate with the Scheduler), we will have two types of DTOs. As both of them will be fairly similar, we will present the class diagrams that model the public interface, and then explain the interface with the Scheduler.

The representation of a *doctor* is as follows:

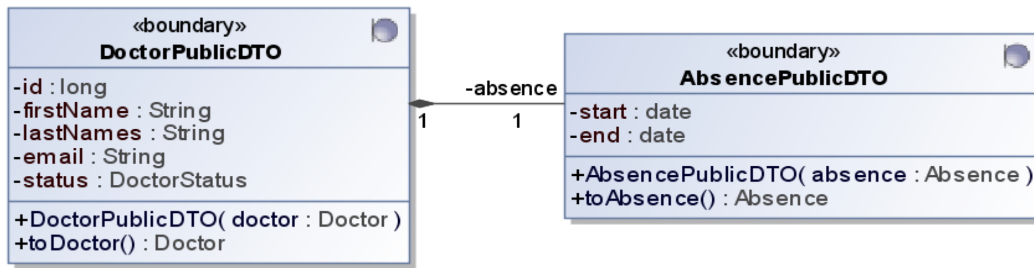


Figure 26: REST Service - View class diagram - doctor

We can see that the start date is not publicly exposed, as that information will be saved upon the doctor's creation. We can also see that the absence does not need a reference to *DoctorPublicDTO*.

Also note the DTOs will be responsible for converting between the model and the view classes (E.g. *DoctorPublicDTO* constructor method takes a *Doctor* as an argument, that will be used to create the DTO. Then, an instance of a *DoctorPublicDTO* can be converted back with *toDoctor*).

Now, these are the DTOs for the *shiftConfig* and *allowedShift* resources:

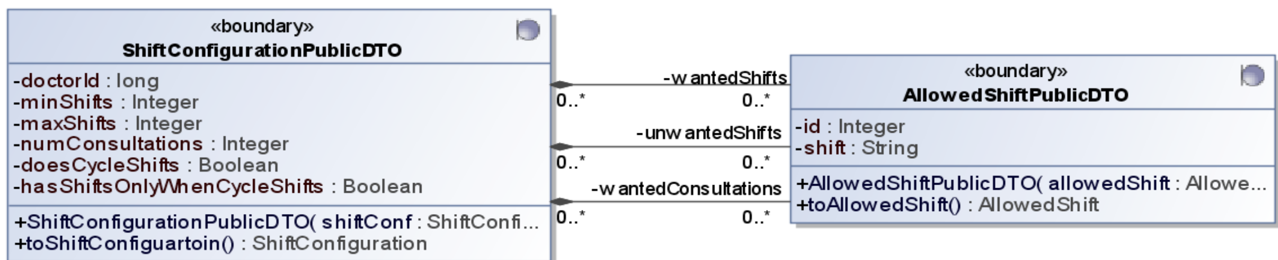


Figure 27: REST Service - View class diagram - shiftConfig and allowedShift

Lastly, we will show the representations of a *calendar* and a *schedule*:

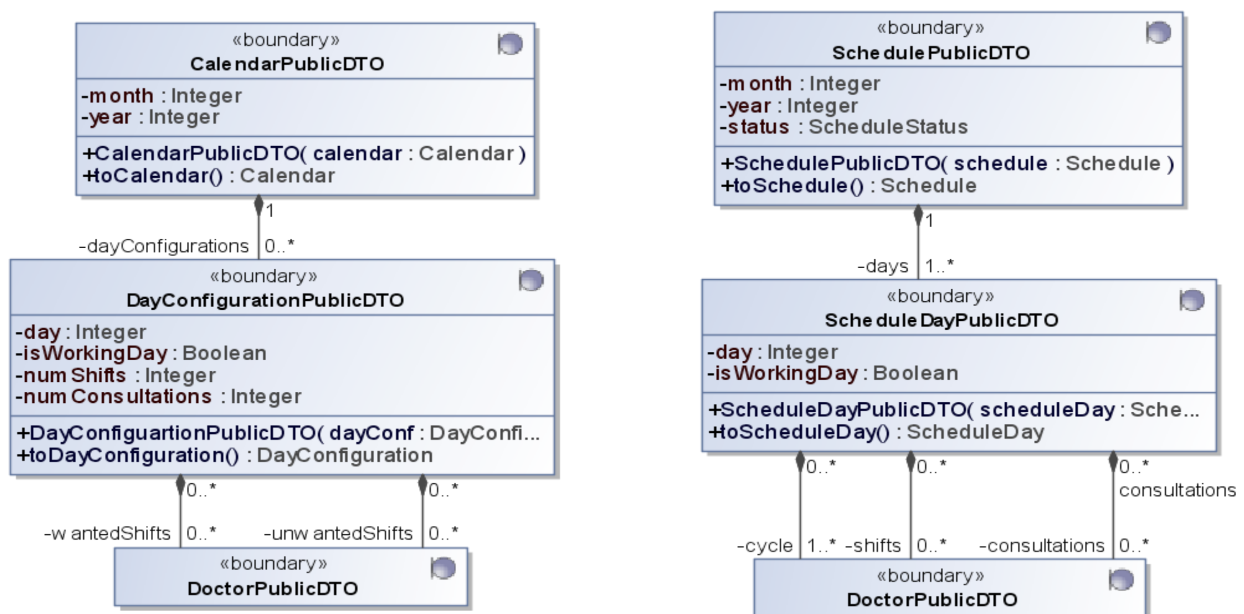


Figure 28: REST Service - View class diagram - calendar and schedule

Note the *DayConfigurationPublicDTO* and *ScheduleDayPublicDTO* do not need references to their corresponding *Calendar* and *Schedule*. Moreover, these classes no longer need the *month* and *year* fields, as that

information is already in their corresponding *CalendarPublic*DTO and *SchedulePublicDTO*.

Currently, the only difference these classes have with the Scheduler DTOs is the serialization of the *Doctor:* The Scheduler does not need the personal information of doctor (*firstName*, *lastNames* and *email*), just their *id*s, their *status* and their *absence*s. However, we will create DTOs for the rest of the model classes. The reason for this is, their construction is different depending on the representation of the *Doctor*. For example, the ScheduleDay DTO can be composed of either DoctorPublicDTOs or DoctorSchedulerDTOs.

Note this design can be improved. For example, we could define a generic type on the DTOs composed of doctors. This generic type would represent the desired doctor DTO, and would be set depending on the context. Then, we would provide the constructors of the DTOs with a java.util.function.Supplier [72]. This *Supplier* would provide either the Public or the Scheduler representation of the *Doctor* depending on the context. However, this work will be left as a future improvement of the project.

One last comment regarding the pattern applied: The DTOs are not just useful to isolate the model entity from its serialization logic, but they also make changing the model entities very simple. For instance, let's say we want to add a new field to the *Doctor* class: *homeAddress*. However, this information is confidential and should only be sent on some authorised requests. If we had not separated the serialization from the data, for every response that contains a doctor resource, we would have to decide whether the *homeAddress* field is serialized or not. However, with the DTO pattern, we would only have to create a new class like *DoctorConfidentialDTO* containing the needed information. Then, on authorised requests, instead of sending the *DoctorPublicDTO*, we would send the *DoctorConfidentialDTO*.

4.4.2.3. DAO interfaces

To isolate the persistence from the rest of the service, we will make use of the DAO pattern. For this reason, we will define interfaces extending the org.springframework.data.jpa.repository.JpaRepository [53] interface. As all the interfaces will be very similar, we will only show the *DoctorRespository*:

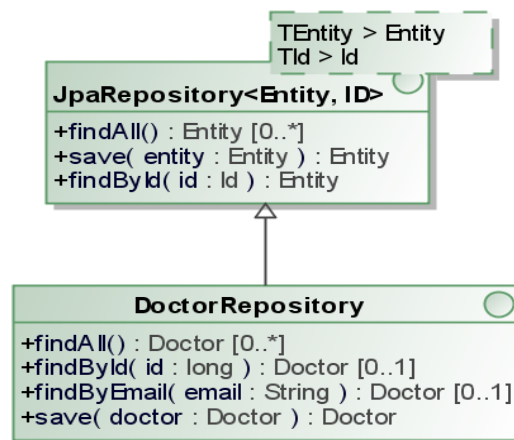


Figure 29: REST Service - DAO class diagram - DoctorRepository

The *DoctorRepository* is the only DAO that provides the *findByEmail* method. The other DAOs will only provide the *findAll*, *findById* and *save* methods.

These interfaces will be implemented automatically by the JPA library. As it has been explained in 2.8.3 Spring Data JPA.

4.4.2.4. Controller classes

The controllers will be responsible for handling the requests to the REST API. This is, responding to the HTTP GET, POST, PUT and DELETE requests.

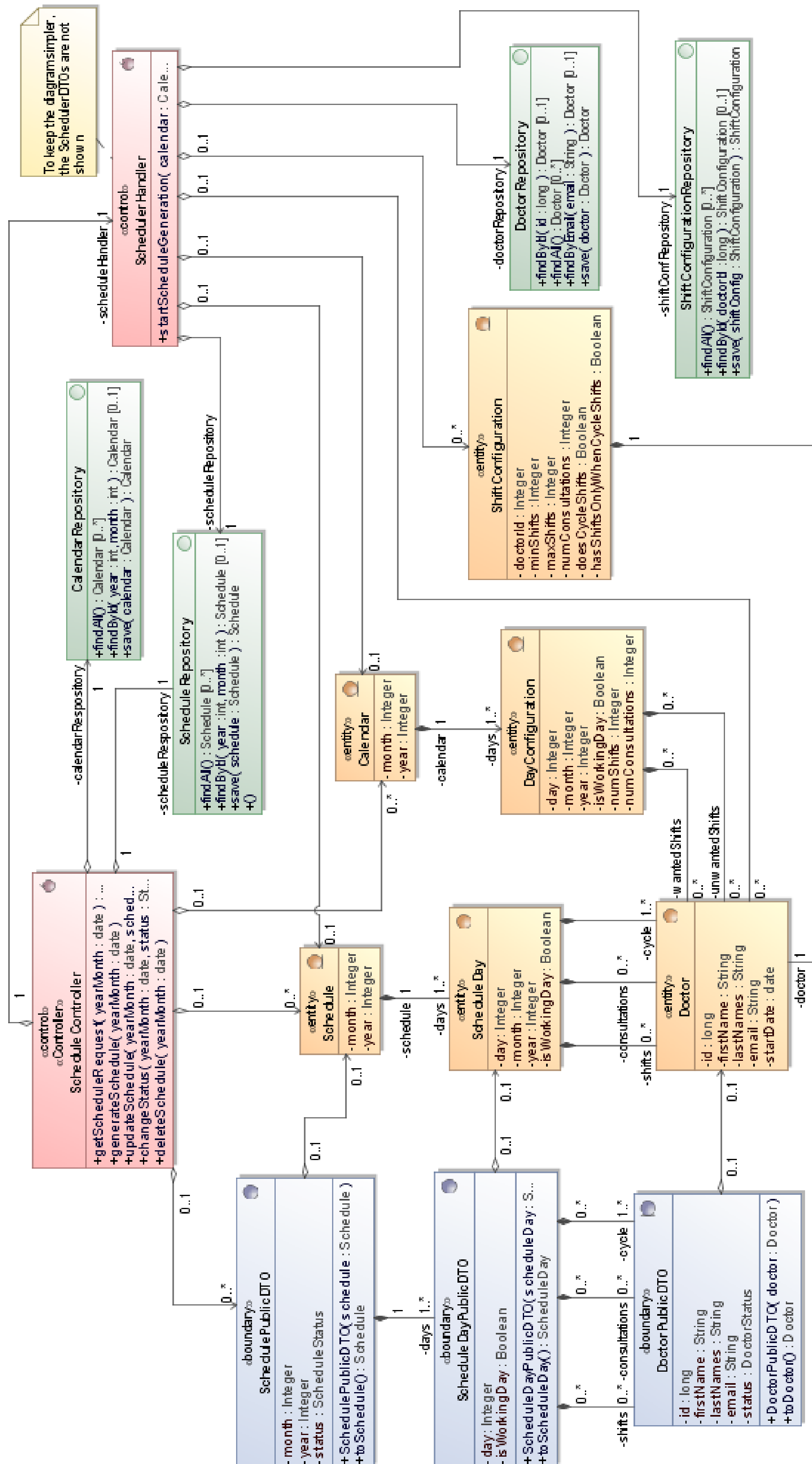


Figure 30: REST Service - ScheduleController class diagram

As all controllers will have a fairly similar structure, we will show only the ScheduleController. The other ones will have the same structure: They will aggregate their corresponding repository, their corresponding entity, and the entity's corresponding public DTO.

Note the schedule controller is also composed of a *ScheduleHandler*. This class will be responsible for the communication process with the Scheduler system. Its behaviour is described in 4.4.4 Communication with the Scheduler.

4.4.3. Behavioural design

The behaviour of all controllers will be fairly similar. For this reason, we will only represent their most relevant operations.

Note that, on the following diagrams, validations and exceptions have not been represented to keep them simple. For example, before creating a doctor, the DoctorController will check that there is no other doctor with the same email. If there was another doctor with the same email, a 400 BAD REQUEST HTTP response would be sent to the Web Application, with an error message in the body such as "Another doctor already has the email example@example.com".

Another feature that has not been represented is the conversion to the *RepresentationModel* of the DTOs. This conversion will occur just as it has been shown in the 2.8.4 Spring HATEOAS section (with the corresponding *RepresentationModelAssembler*s).

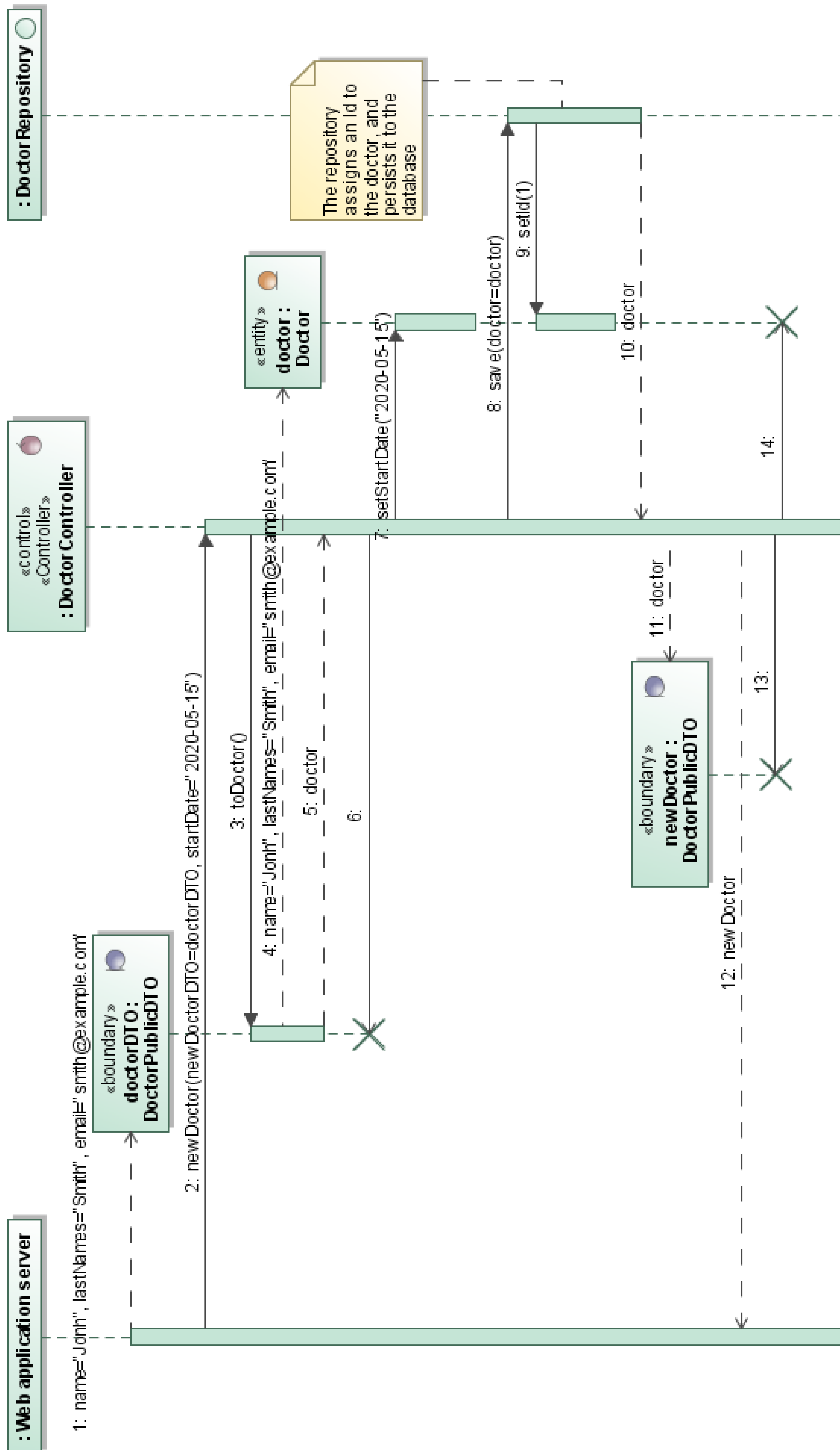


Figure 31: REST Service - DoctorController.newDoctor sequence

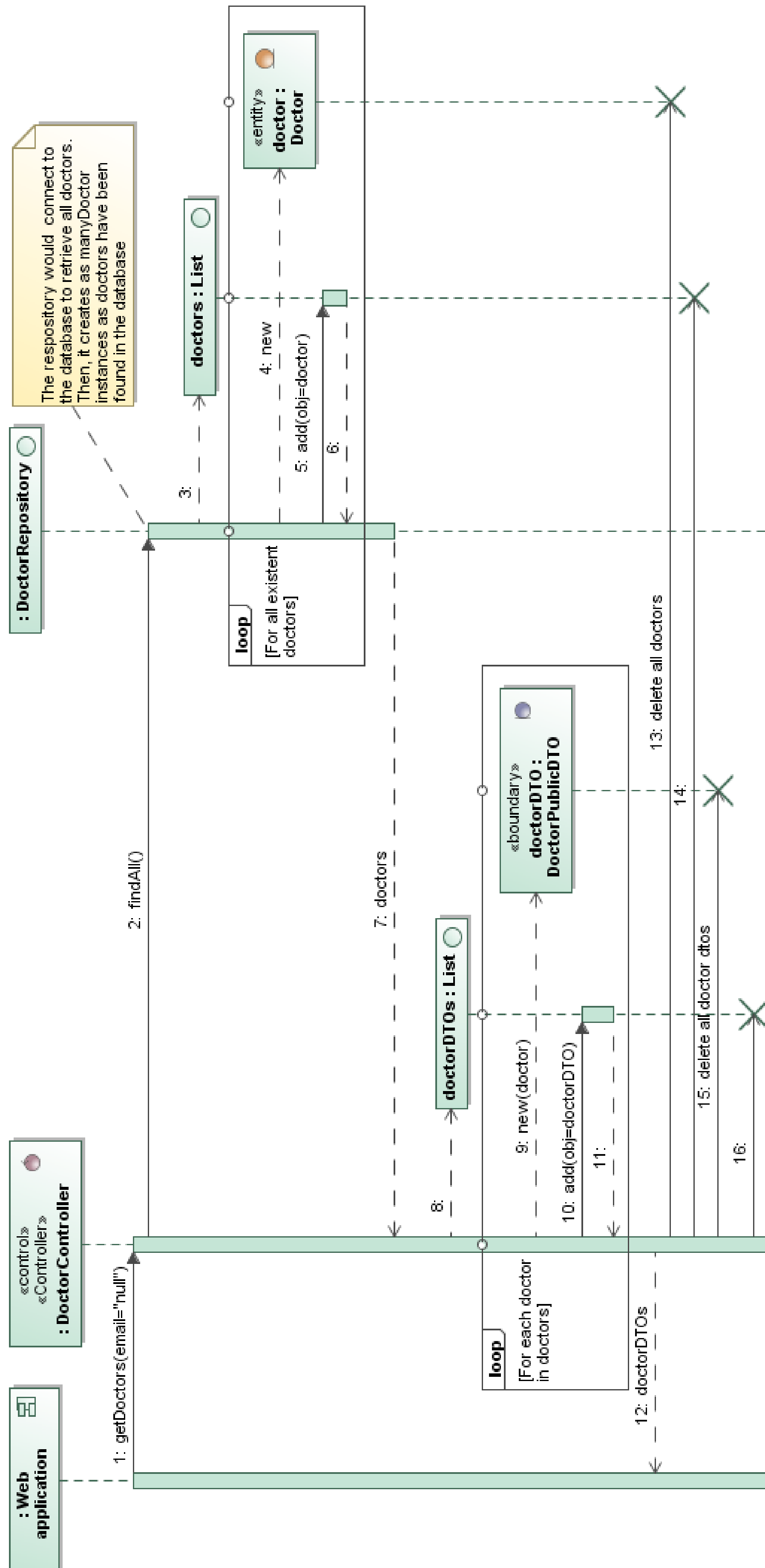


Figure 32: REST Service - DoctorController.getDoctors sequence

Another sequence of messages worth showing is the generation of a schedule. We will present the complete diagram, and then divide it into two different figures (so that it can be read without zooming in).

The first part of the diagram (messages 1 to 11) corresponds to the Web Application's request, and the second part (messages 11 to 31) is the generation of the schedule. Note the schedule generation will be asynchronous, and will not lock the web application.

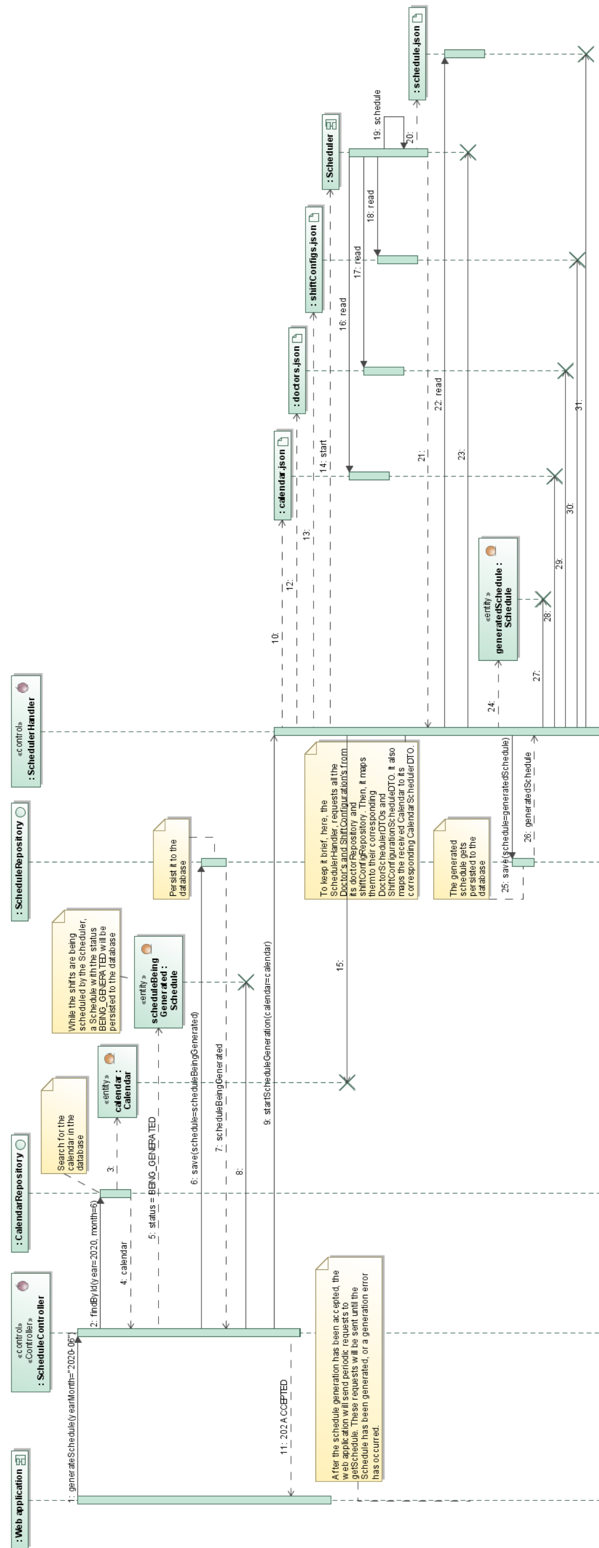


Figure 33: REST service - ScheduleController.generateSchedule sequence - Overview

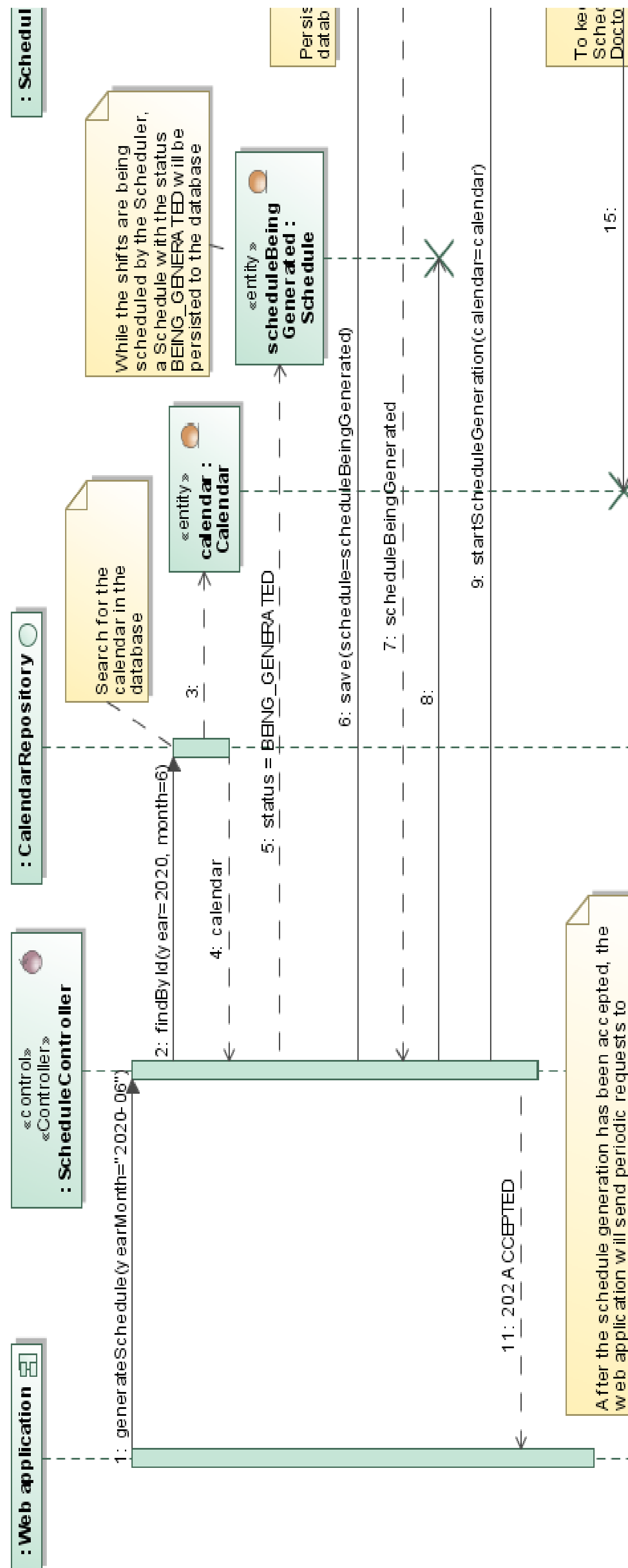


Figure 34: REST Service - ScheduleController.generateSchedule sequence - Upper part

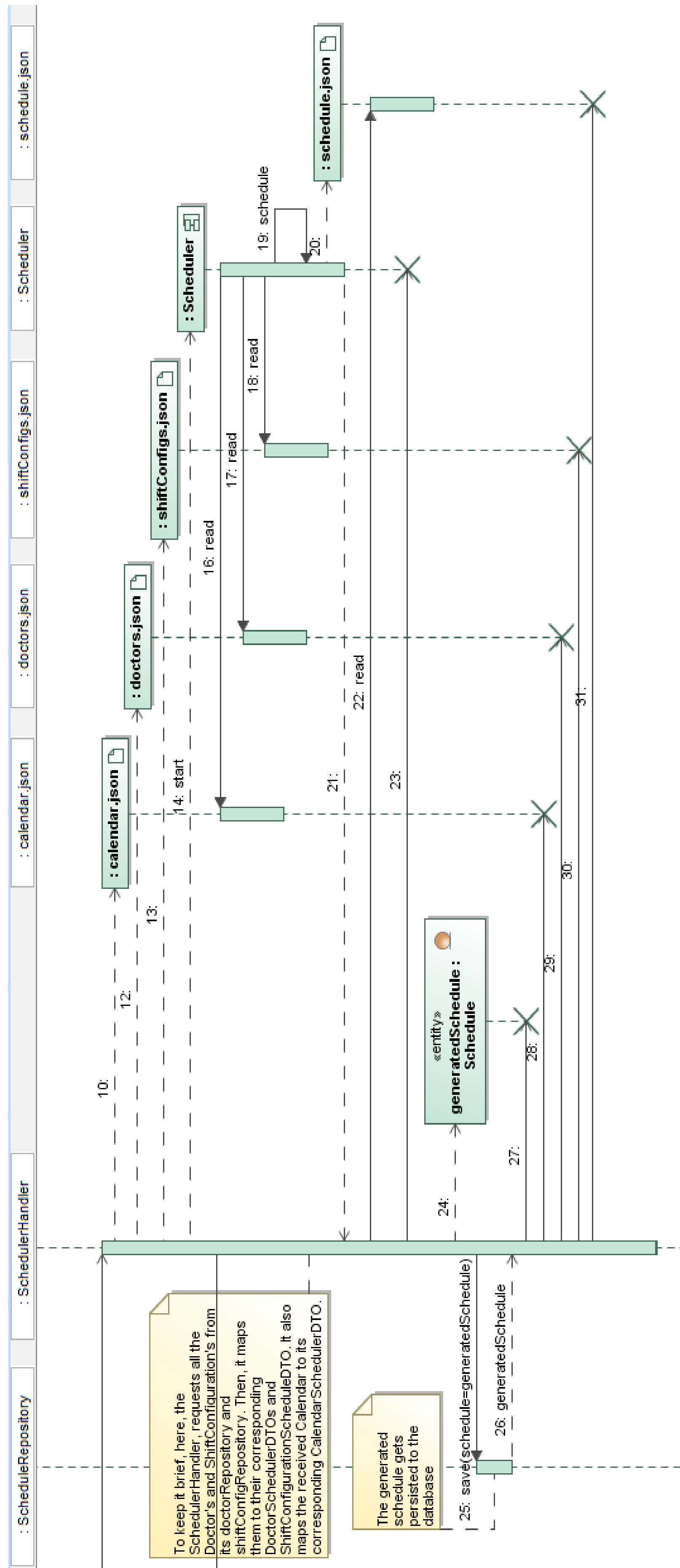


Figure 35: REST Service - ScheduleController.generateSchedule sequence - Bottom part

4.4.4. Communication with the Scheduler

The Scheduler does not need to be a daemon process, it will only be needed from time to time. For this reason, the REST service will be the responsible for starting the Scheduler every time it is needed. Specifically, the process will be started when a POST request is sent to the corresponding IR-03 - Schedule URI. This section will describe the communication process between the REST service and the Scheduler.

To understand this communication process we need to know the information the Scheduler needs: all the D*octors* in the system[vii], their S*hiftConfigurations* and the *Calendar* of the corresponding month (The specific information needed is shown in 4.5 Scheduler). Then, with this information, the Scheduler will execute its algorithm and will generate the desired *Schedule*.

Now, the way we have decided to implement this communication is through files. Specifically, the REST service will pass on four arguments to the Scheduler process. These will be:

- Path to a file containing the list of *DoctorSchedulerDTOs* as JSON.
- Path to a file containing the list of *ShiftConfigurationSchedulerDTOs* as JSON.
- Path to a file containing the *CalendarSchedulerDTO* as JSON.
- Path to a file that will contain the resulting *ScheduleSchedulerDTO* as JSON.

Then, the Scheduler will read the first three files to extract the information it needs. Afterwards, when its algorithm finishes, it will write the generated schedule to the last file. If the schedule could not be generated correctly, the *ScheduleSchedulerDTO* will be in the GENERATION_ERROR status. Lastly, the REST service will try to read this file and persist the schedule.

This process might be easier to understand with a simple activity diagram:

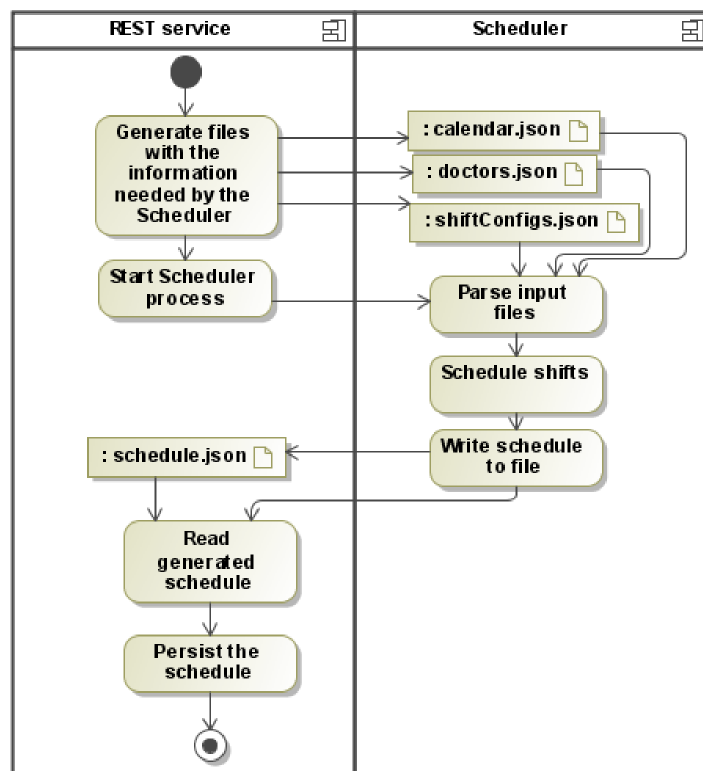


Figure 36: REST service - Communication with the Scheduler

vii Only the AVAILABLE doctors should be provided.

This communication process has three main problems:

- Concurrency: If the same file paths are used every time (as they currently do), and multiple requests to schedule shifts are received at the same time, the behaviour of the system would not be defined. It would depend on the order of the file reads/writes. However, this risk can be assumed for now, as we are not expecting for multiple schedules to be generated simultaneously.

- Insecure: As the information will be written into plain text in files, any process with permissions to read them could get this data. Still, the impact of this issue has been minimized by removing any non necessary information from the serialized data (the doctor's names and emails).

- Required R/W permissions: As both the scheduler and REST service need to read/write into these files, they need to have the appropriate permissions.

Other alternatives we have also considered are exchanging the information through raw sockets, encapsulating it on some application like HTTP, or even have the Scheduler access the database on its own. However, keeping in mind the current goal is to present a functioning prototype of the system, these alternatives would take longer to develop.

Still, in the future, these problems should be amended. For this reason, both the design of the REST service and the Scheduler take this problem into account. Specifically, they aim to make changing the communication process as simple as possible:

- REST service: The whole communication process currently takes place in the *SchedulerHandler* class. If this process was to be changed, we would only need to change this class, or create a new one and inject it to the *ScheduleController*. See 2.8.1.5 Dependency injection.

- Scheduler: All the scheduling logic has been separated from the communication process. This way, if it was to be changed, only the main method would have to be modified (The main method is only responsible for configuration and the communication process).

4.5. Scheduler

This system will be responsible for scheduling the shifts of a certain month, according to the requirements[viii]. To do this, the scheduler will only need to know the shift configuration of all doctors, the doctors' id, start date and status, and the calendar whose schedule is to be generated.

To begin with, we will introduce the algorithm used to calculate cyclic-shifts, and then the linear programming problem modelling the restrictions to schedule non-cyclic-shifts. Lastly, we will show how the scheduler has been designed.

4.5.1. Cyclic-shift scheduling algorithm

This section will explain how CSs will be calculated. To calculate the shifts a doctor will have a certain month, all we need is a reference date and a the rate at which CSs are repeated (BR-SCH-02 - CS rate):

$$RefDate, FirstDayMonth, R \in \mathbb{N}$$

Where:

- *RefDate* is the number of days elapsed since the 1st of January of 1970[ix] until the reference date of a doctor.

[viii] Note that doctor's absences will not be taken into account in this first version of the scheduler. They will be left as future improvements to the system.

[ix] Note the 1st of January of 1970 has been selected arbitrarily. The algorithm would work with any other date, as long as the same one is used to calculate both *RefDate* and *FirstDayMonth*.

- *FirstDayMonth* is the number of days elapsed since the 1st of January of 1970 until the first day of the month whose shifts are to be calculated.

- *R* is the rate at which CS take place.

With these three values, we can calculate the number of days from the *FirstDayMonth* until the next CS of the doctor:

$$NumDays = R - \left[\left(FirstDayMonth - RefDate \right) Mod\, R \right]$$

This way, the doctor will have a CS the days $FirstDayMonth + NumDays$, $FirstDayMonth + 2\, NumDays$, $FirstDayMonth + 3\, NumDays$ ...

If *NumDays* equals *R*, then the doctor will also have a CS on *FirstDayMonth*.

By applying the previous formulas to all the doctors, we will have all the CSs of the desired month.

4.5.2. Non-cyclic-shift linear programming problem

This section describes how the restrictions to assign NCSs have been modelled as an integer programming problem with boolean variables.

4.5.2.1. Definitions

First, we will define seven constants:

$$N, M \in \mathbb{N}$$

N is the number of available doctors in the system[x].

M is the number of working days in the month whose shifts are to be scheduled (BR-SCH-03 - NCS allowed days).

$$W_{wanted\,shift}, W_{unwanted\,shift}, W_{wanted\,consultation}, W_{shift}, W_{consultation} \in \{0,1,2,\ldots,100\}$$ [xi]

W wanted shift: The weight given to assigning a regular-shift to a doctor who wants it.

W unwanted shift: The weight given to assigning a regular-shift to a doctor who does not want it.

W wanted consultation: The weight given to assigning a consultation to a doctor who wants it.

W shift: The weight given to assigning any regular-shift.

W consultation: The weight given to assigning any consultation.

Note: An explanation on how to choose the values for these weights is given at 4.5.3 Scheduler's design.

We will also define 3 x N x M constants that will represent whether a doctor has a certain shift preference:

$$w_{wanted\,shift,ij} \in \{0, W_{wanted\,shift}\} \forall\ i=1,2,..,N, j=1,2,\ldots,M$$

$$w_{unwanted\,shift,ij} \in \{0, W_{unwanted\,shift}\} \forall\ i=1,2,..,N, j=1,2,\ldots,M$$

[x] A doctor is available if they are not in the deleted status.

[xi] The weights could theoretically have any real value greater than or equal to zero. However, just to bound their values, we have chosen to restrict them to the set {0,1,…,100}.

$$w_{wanted\,consultation,ij} \in \{0, W_{wanted\,consultation}\} \forall\ i=1,2,..,N, j=1,2,...,M$$

These will represent the whether a doctor wants/does not want a regular-shift/consultation. Note that it only makes sense to define these values for working days (j=1,…,M), as there cannot be any NCSs assigned on a non working day.

For example, $w_{wanted\ shift,\ 1,1} = 0$ represents the doctor 1 does not have a request to get a shift assigned on the first working day of the month. However, if $w_{wanted\ shift,\ 1,2} = W_{wanted\ shift}$, that represents the doctor 1 would like to have a shift the second working day of the month.

Now, we will define two sets:

$$A \subset \mathbb{N}^2$$

This set will contain the pair i, j with $i=1,2,...,N, j=1,2,...,M$ if the *i-th* doctor has a cyclic-shift the *j-th* working day of the month.

$$B \subset \mathbb{N}$$

This set will contain the values i with $i=1,2,...,N$ if the *i-th* doctor can have NCS on days they do not have a CS (BR-SCH-11 - NCS only when CS).

We will also define the values that regard the shift configuration of a doctor and the configuration of each working day:

$$minS_i, maxS_i, numC_i \in \mathbb{N}\ \forall\ i=1,2,...,N$$

Where $minS_i$, $maxS_i$ and $numC_i$ represent the minimum number of regular-shifts, maximum number of NCS and number of consultations of the *i-th* doctor (BR-SCH-04 - Minimum regular-shifts per doctor, BR-SCH-05 - Maximum NCS per doctor, BR-SCH-06 – Consultations per doctor).

$$numS_j, numC_j \in \mathbb{N}\ \forall\ j=1,2,...,M$$

When $numS_i$ and $numC_i$ represent the minimum number of regular-shifts and consultations that have to be scheduled the *j-th* working day of the month (BR-SCH-08 - Minimum number of regular-shifts and consultations per day).

Lastly, we will define the variables of the problem:

$$s_{ij} \in \{0,1\} \forall\ i=1,2,..,N, j=1,2,...,M$$

These variables will represent whether the *i-th* doctor has a regular-shift on the *j-th* working day of the desired month. E.g. if $s_{1,2}=1$, the doctor 1 has a regular-shift the second working day of the month.

$$c_{ij} \in \{0,1\} \forall\ i=1,2,..,N, j=1,2,...,M$$

These variables will represent whether the *i-th* doctor has a consultation on the *j-th* working day of the desired month.

4.5.2.2. The linear programming problem

Now that the needed definitions are made, we will present the whole problem. Afterwards, we will describe each one of its parts:

$$Max \sum_{i=1}^{N} \sum_{j=1}^{M} \left(w_{wanted\,shift,ij} - w_{unwanted\,shift,ij} - W_{shift}\right) s_{ij} + \left(w_{wanted\,consultation,ij} + W_{consultation}\right) c_{ij} \quad (1)$$

$$s_{ij} = 1 \; \forall \; i,j \in A \quad (2)$$

$$s_{ij} + c_{ij} \leqslant 1 \; \forall \; i=1,2,\ldots,N, j=1,2,\ldots,M \quad (3)$$

$$\sum_{j=1}^{M} s_{ij} \geqslant minS_i \; \forall \; i \in B \quad (4)$$

$$\sum_{j=1}^{M} s_{ij} + c_{ij} \leqslant maxS_i \; \forall \; i \in B \quad (5)$$

$$\sum_{j=1}^{M} c_{ij} \leqslant numC_i \; \forall \; i \in B \quad (6)$$

$$\sum_{i=1}^{N} s_{ij} \geqslant minS_j \; \forall \; j=1,2,\ldots,M \quad (7)$$

$$\sum_{i=1}^{N} c_{ij} \geqslant minC_j \; \forall \; j=1,2,\ldots,M \quad (8)$$

4.5.2.3. The objective function

The objective function (1) has five different contributions:

1. $w_{wanted\,shift,ij} s_{ij}$ This contribution will try to maximize the number of regular-shifts assigned to doctors who wanted them.
2. $-w_{unwanted\,shift,ij} s_{ij}$ This contribution will try to minimize the number of regular-shifts assigned to doctors who did not want them.
3. $-W_{shift} s_{ij}$ This contribution will try to minimize the number of regular-shifts assigned overall.
4. $w_{wanted\,consultation,ij} c_{ij}$ This contribution will try to maximize the number of consultations assigned to doctors who wanted them.
5. $W_{consultation} c_{ij}$ This contribution will try to maximize the number of consultations assigned overall.

The importance of these five contributions are not their actual values but their relative differences. For example, if $W_{wanted\,shift} - W_{unwanted\,shift} < 0$ then, meeting unwanted regular-shift requests will have a higher priority than meeting wanted-shift requests. This is further explained in 4.5.3 Scheduler's design.

4.5.2.4. CS implies a NCS

The restriction (2) corresponds to the requirement BR-SCH-07 - A CS implies a regular-shift.

4.5.2.5. Only one shift per day

The restriction (3) will make sure a doctor cannot be assigned a regular-shift and a consultation the same day. This is because, as mentioned in the 1.1 Description of the problem, both regular-shifts and consultations take place at the same hour.

4.5.2.6. Maximums and minimums per doctor

The restrictions (4), (5) and (6) correspond to requirements BR-SCH-04 - Minimum regular-shifts per doctor, BR-SCH-05 - Maximum NCS per doctor and BR-SCH-06 – Consultations per doctor respectively.

4.5.2.7. Minimums per day

The restrictions (7) and (8) correspond to requirement BR-SCH-08 - Minimum number of regular-shifts and consultations per day.

4.5.3. Scheduler's design

The scheduler has been divided into two functions. Each with different responsibilities:

- Main function: Responsible for the communication process with the REST service (reading and writing to the corresponding files), and responsible for loading the configuration (reading from the configuration file).

- Schedule function: Responsible for applying the algorithm to calculate cyclic-shifts, and responsible for solving the linear programming problem.

We can find a simple activity diagram on how the scheduler works in Figure 36: REST service - Communication with the Scheduler (page 86). In that diagram, we can see there are four JSON files involved: doctors.json, shiftConfigs.json, calendar.json and schedule.json. There is also another JSON file used by the scheduler: schedulerConfig.json. We will describe de content of these files one by one:

- **doctors.json**:

  This file will contain the information related to all AVAILABLE doctors in the system. Specifically, the scheduler only needs the their id, their startDate and their absence. For example:

```
[
 {
   "id": 1,
   "absence": null,
   "startDate": "2020-05-01"
 },
 {
   "id": 2,
   "absence": {
     “start”: “2020-07-15”,
     “end”: “2020-07-25”
   },
   "startDate": "2020-05-01"
 },
...
]
```

- **shiftConfigs.json**:

  This file has to contain the shift configuration of all AVAILABLE doctors in the system. For example:

```
[
 {
   "doctorId": 1,
   "maxShifts": 0,
   "minShifts": 0,
   "numConsultations": 0,
   "doesCycleShifts": true,
   "hasShiftsOnlyWhenCycleShifts": false,
   "unwantedShifts": [],
   "wantedShifts": [
     {
       "id": 2,
       "shift": "Tuesday"
     },
     {
       "id": 3,
       "shift": "Wednesday"
     }
   ],
   "wantedConsultations": []
 },
 {
   "doctorId": 2,
   "maxShifts": 5,
   "minShifts": 3,
   "numConsultations": 0,
   "doesCycleShifts": true,
   "hasShiftsOnlyWhenCycleShifts": true,
   "unwantedShifts": [
     {
       "id": 3,
       "shift": "Wednesday"
     }
   ],
   "wantedShifts": [],
   "wantedConsultations": []
 },
...
]
```

- **calendar.json**:

  This file has to contain the configuration of the month whose shifts will be scheduled. For example:

```
{
 "month": 5,
 "year": 2020,
 "dayConfigurations": [
   {
     "day": 1,
     "isWorkingDay": true,
     "numShifts": 2,
     "numConsultations": 0,
     "unwantedShifts": [{"id": 1}, {"id": 4}],
     "wantedShifts": []
   },
   {
     "day": 2,
     "isWorkingDay": false,
```

```
      "numShifts": 2,
      "numConsultations": 0,
      "unwantedShifts": [],
      "wantedShifts": [{"id": 3}]
    },
  ...
  ]
```

The wanted (unwanted) shifts represent a doctor who wants (does not want) to have a regular-shift that day. E.g. the doctors with id 1 and 4 do not want to have a regular-shift the 1st of May 2020.

- **schedule.json**:

This will be the output file of the scheduler, and will contain the generated schedule:

```
  {
  "month": 5,
  "year": 2020,
  "status": "PENDING_CONFIRMATION",
  "days": [
    {
      "day": 1,
      "cycle": [{"id": 1},{"id": 2}],
      "shifts": [{"id": 2},{"id": 19}],
      "consultations": [{"id": 13},{"id": 17},{"id": 20}]
    },
    {
      "day": 2,
      "cycle": [{"id": 3},{"id": 4}],
      "shifts": [],  # Not a working day
      "consultations": []
    },
  ...
  ]
```

However, if there has been any error while trying to generate the schedule, the status will be “GENERATION_ERROR”, and the list “days” will be empty.

- **schedulerConfig.json**:

This file will contain the information needed by the scheduler that has not been supplied by the REST service. This is, the rate at which cyclic-shifts are repeated (BR-SCH-02 - CS rate), and the weights used in the objective function:

$W_{wanted\,shift}, W_{unwanted\,shift}, W_{wanted\,consultation}, W_{shift}, W_{consultation}$

The content of this files will be as follows:

```
  {
  "description":
          "This is the configuration file of the scheduler. The
          description of each configuration is self explanatory.
          However, there is one thing to take into account: the
          ABSOLUTE VALUES of the weights are NOT IMPORTANT by
          themselves. What actually makes the behaviour of the
          scheduler change is the difference in the RELATIVE
          VALUES. This is, if wantedShiftWeight=100 and
          unwantedShiftWeight=100, the scheduler will give the
          same priority to both of them. However, if
          wantedShiftWeight=2 and unwantedShiftWeight=1, the
```

```
        scheduler will give twice as much priority to fulfilling
        a wanted shift request than an unwanted shift one",
  "cycleShiftRate": {
    "value": 10,
    "description":
          "This value will represent the rate at which
          doctors have cycle shifts. For example, If the
          value is 10, doctors will have cycle-shifts every
          10 days. A value greater than zero is expected"
   },
  "wantedShiftWeight": {
    "value": 30,
    "description":
          "This value represents the weight given to allowing
          a doctor to take one of their wanted shifts. A
          higher value means a higher priority to assign
          wanted shifts to the doctors. A value greater than
          or equal to zero is expected."
  },
  "unwantedShiftWeight": {
    "value": 30,
    "description":
          "This value represents the weight given to allowing
          a doctor to not take one of their unwanted shifts.
          A higher value means a higher priority to not
          assign unwanted shifts to the doctors. A value
          greater than or equal to zero is expected."
},
  "wantedConsultationWeight": {
    "value": 30,
    "description":
          "This value represents the weight given to allowing
          a doctor to take one of their wanted consultations.
          A higher value means a higher priority to assign
          wanted consultations to the doctors. A value
          greater than or equal to zero is expected."
},
  "allShiftWeight": {
    "value": 10,
    "description":
          "This value represents the weight given to each
          shift that is scheduled. The contribution of shifts
          to the objective function will be negative, meaning
          that a higher value of this field will make the
          scheduler try to assign as few shifts as possible.
          A value greater than zero is expected"
},
  "consultationWeight": {
    "value": 10,
    "description":
          "This value represents the weight given to each
          consultation that is scheduled. The contribution of
          consultations to the objective function will be
          positive, meaning that a higher value of this field
          will make the scheduler try to assign as much
          consultations as possible. A value greater than
          zero is expected"
  }
}
```

The “description” fields contain explanations on how the values they refer to affect the scheduling problem. This will make it easier to configure it without the need to look into the code if the requirements were to change. For example, if we wanted to give to wanted shift requests a higher preference than to unwanted shift requests, we would have to make “wantedShiftWeight.value” greater than “unwantedShiftWeight.value”.

As explained in 4.4.4 Communication with the Scheduler, the paths to doctors.json, shiftConfigs.json, calendar.json and schedule.json will be arguments of the scheduler program specified by the REST service. However, the path to the scheduleConfig.json will be defined as constant in the program. Still, the scheduler accepts an optional argument “`--configDir=<path/to/config/directory>`” that can be used to indicate a different directory containing the scheduler’s configuration files. For example.: “`--configDir=/etc/guardians/scheduler/`”.

Note the exact procedure followed by the schedule function will not be explained. It just applies the concepts explained in the 2 State of technology chapter to solve a linear programming problem using the Google ORTools library. However, the code can be found on its public repository [73].

4.6. Web application

The Web application will be responsible for presenting the user (A-02 – Shift manager) a simple interface to interact with the system.

The design of this web application will be divided into two sections:

- Structural design
- Behavioural design

4.6.1. Structural design

The class diagram of the application can be found below on Figure 37: Web application - Class diagram.

Note that, the model classes will have the same attributes and relations as in 4.4 REST service. For this reason, they are not shown in the diagram. For further detail, refer to 4.4.2.1 Model classes.

Also, note that the communication process with the REST service has been separated into service classes. This way, the controllers are only responsible for parsing user input, and selecting the template to be shown to the user (The *String* values returned by the controllers are the path to the templates relative to `resources/templates`).

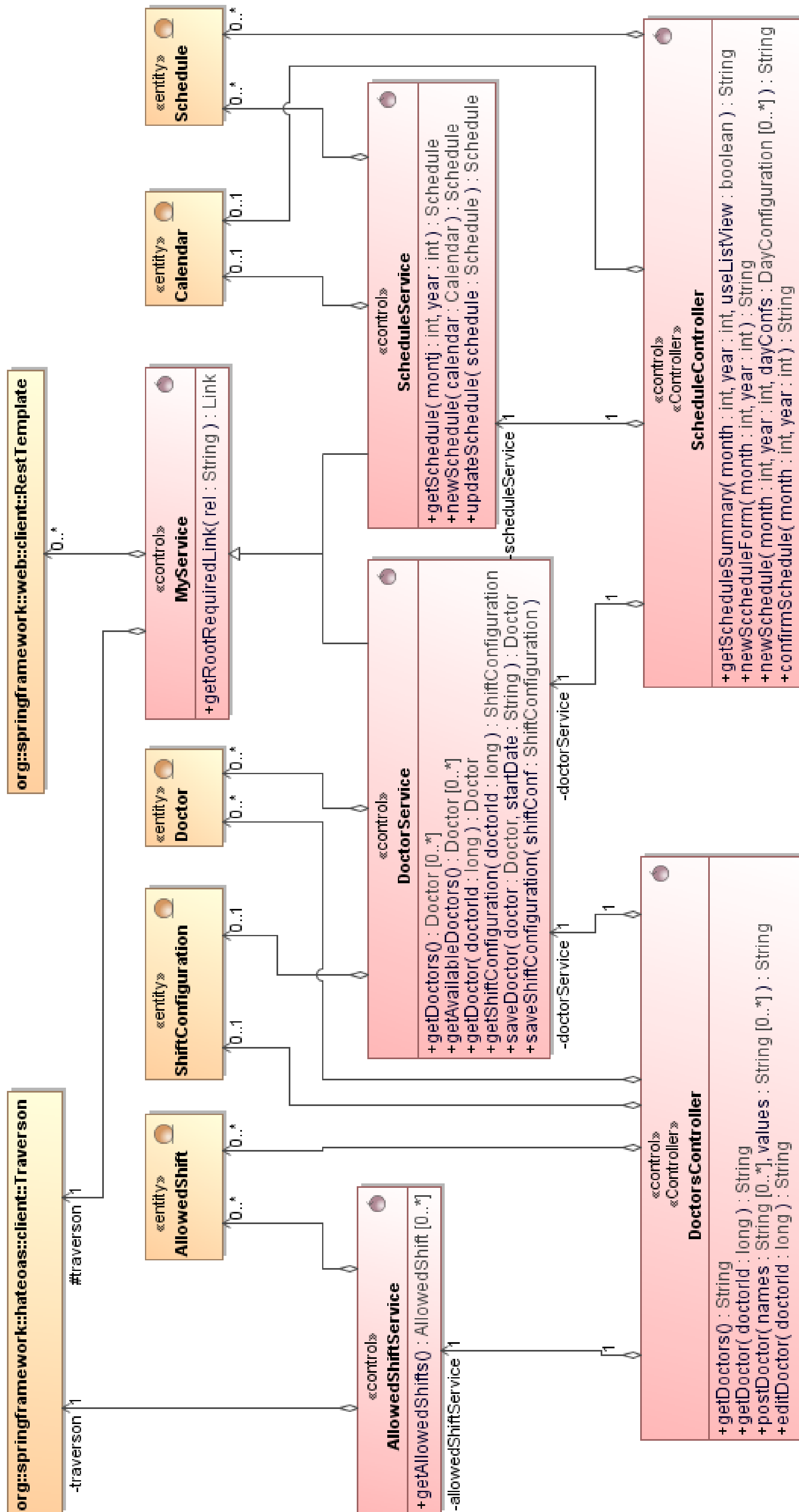


Figure 37: Web application - Class diagram

4.6.2. Behavioural design

To begin with, we will start with the basic scenario in which the user wants to see a list of all existent doctors in the system:

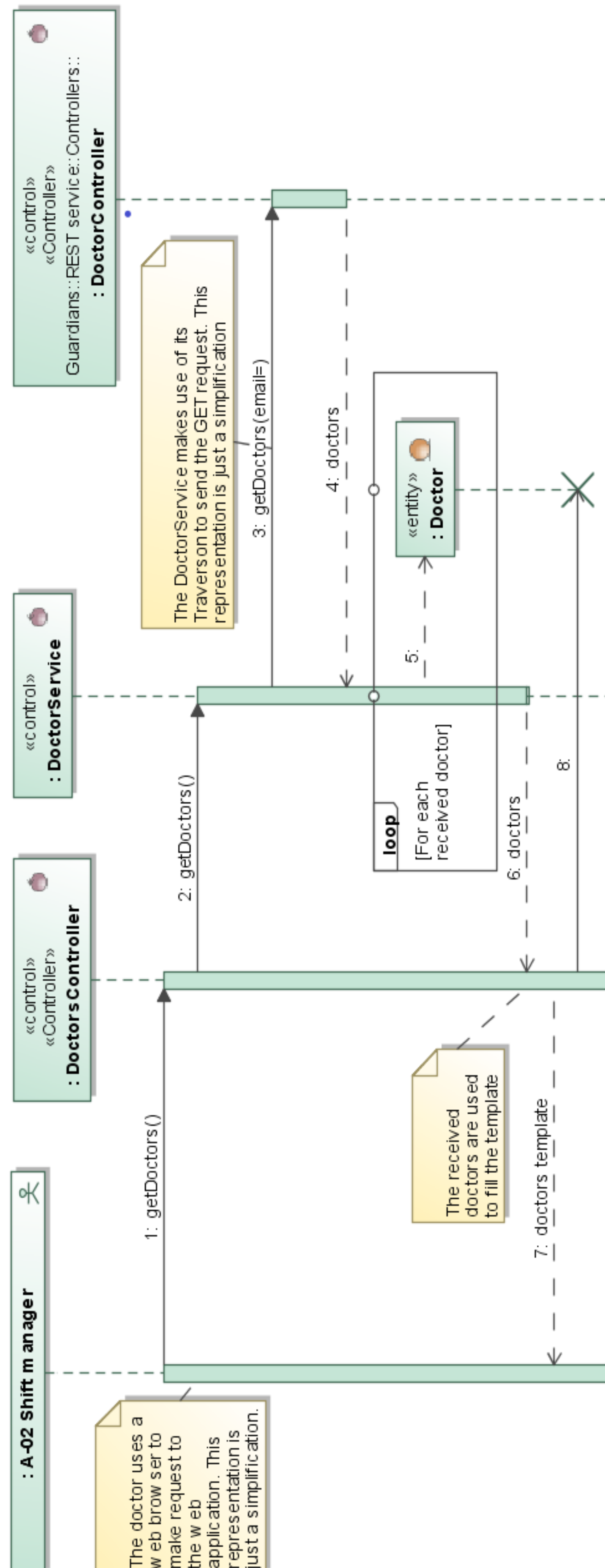


Figure 38: Web application - DoctorsController.getDoctors sequence

Now, let's say the user would like to create/edit a doctor. Then, they first have to request the edit-doctor form:

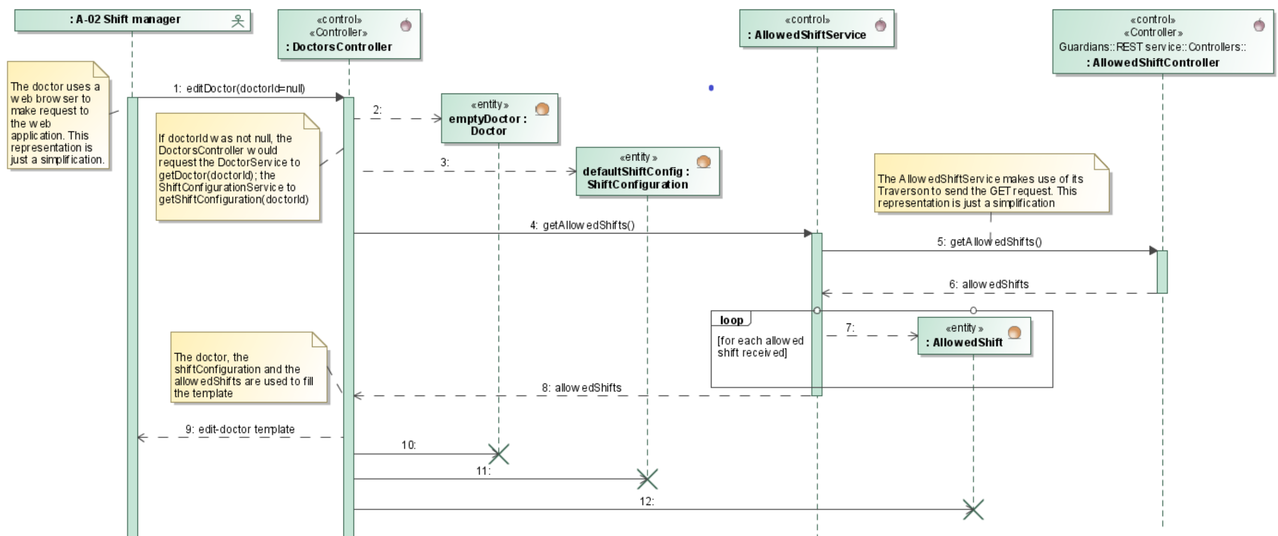


Figure 39: Web application - DoctorsController.newDoctorForm sequence

As the image can only be read correctly by zooming in, we will divide into into two halves and present them separately:

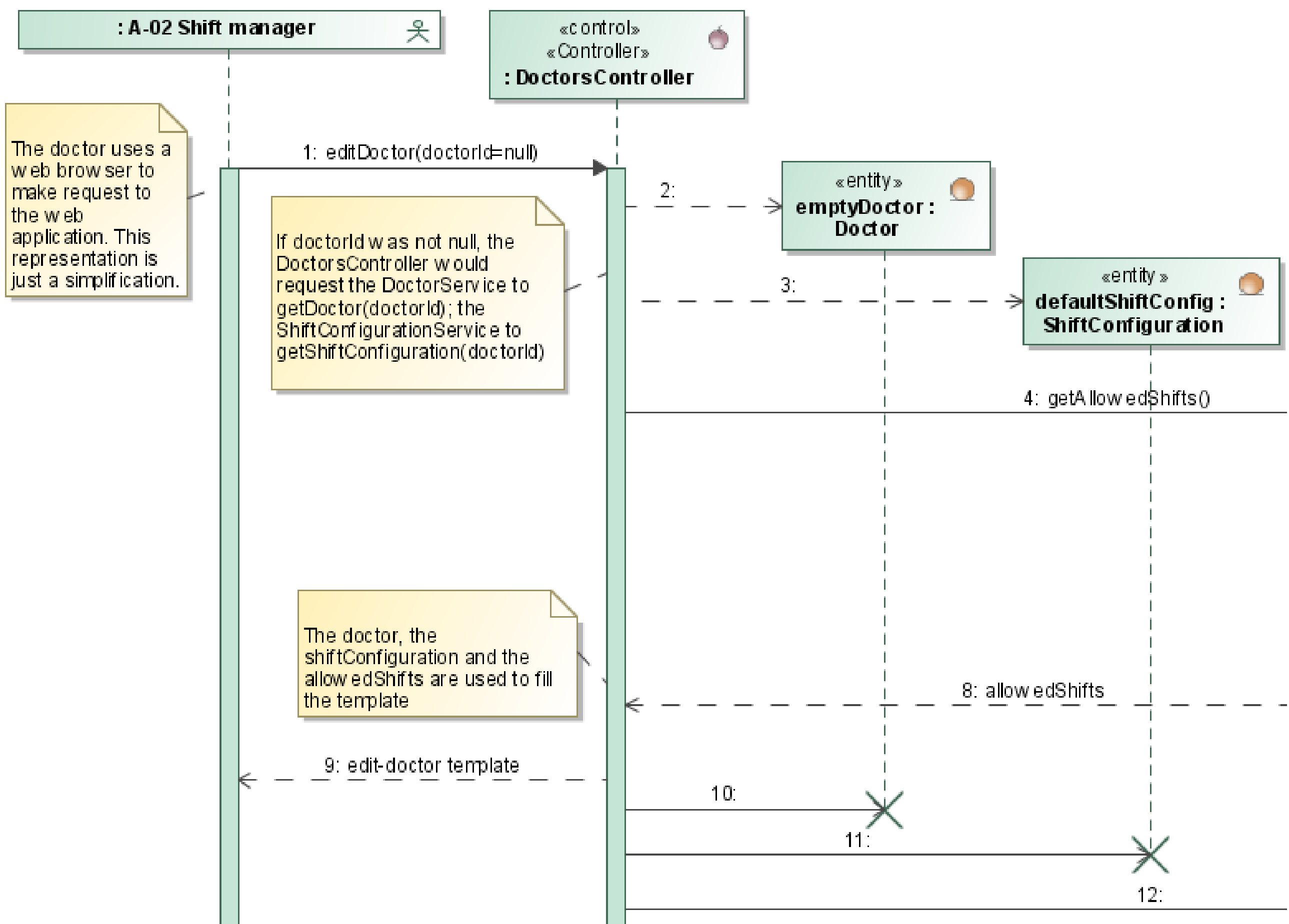


Figure 40: Web application - DoctorsContrller.newDoctorForm sequence - left part

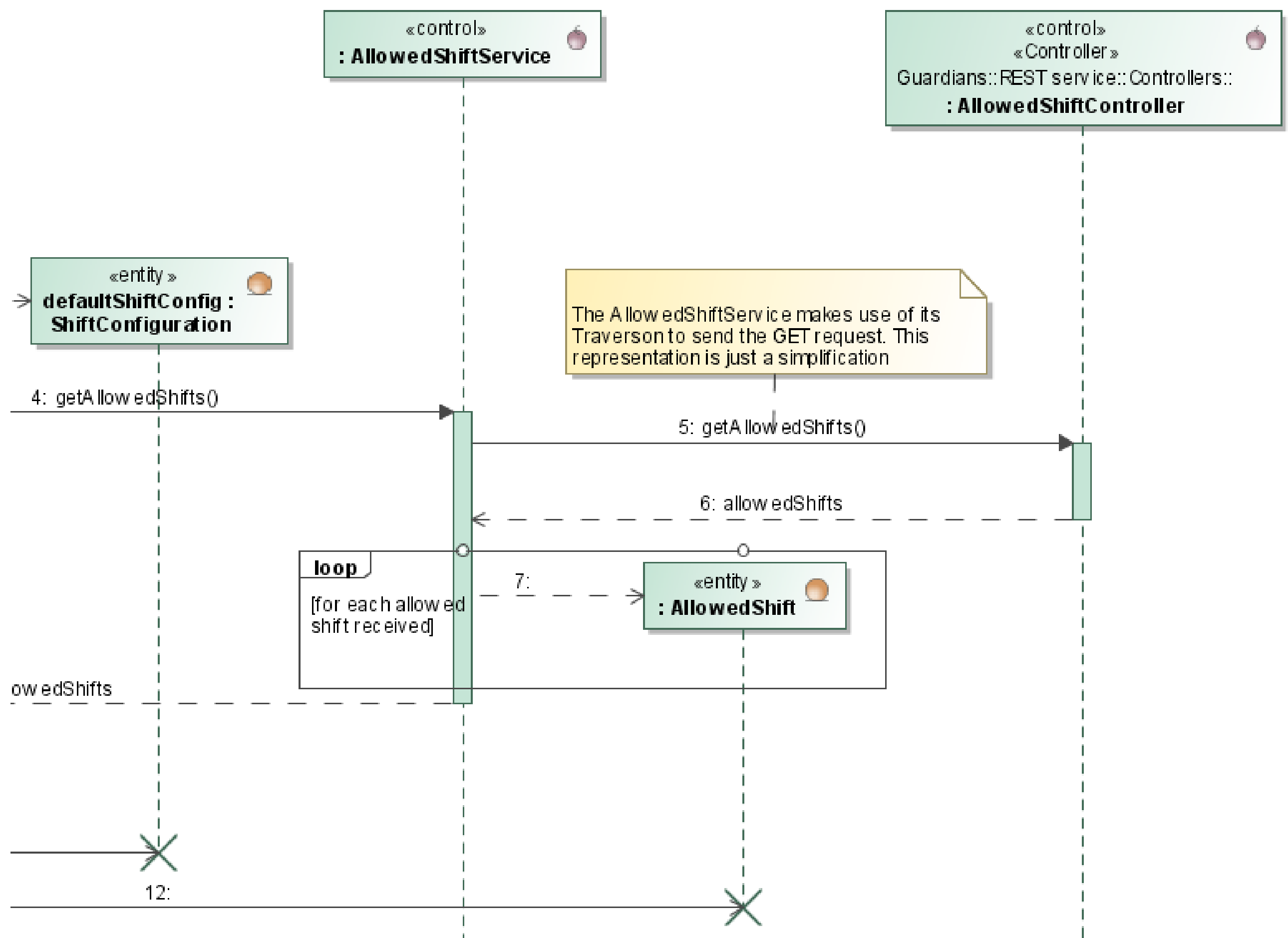


Figure 41: Web application - DoctorsController.newDoctorForm sequence - right part

After the edit-doctor template is sent, and the user submits the form, the following sequence of messages will occur (We will also show them separately, as the image below is only readable by zooming in):

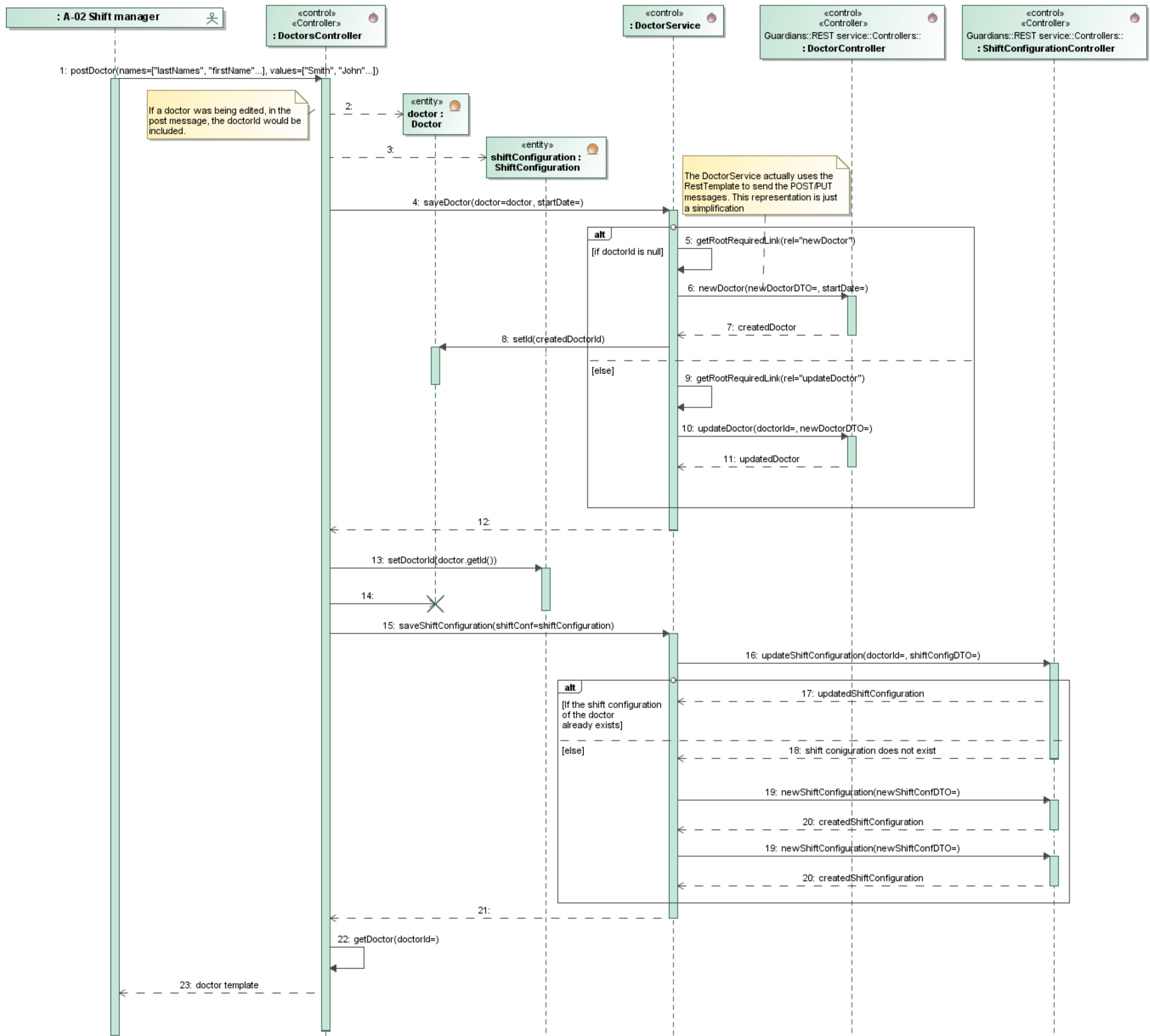


Figure 42: Web application - DoctorsController.newDoctor sequence

The controller creates a *Doctor* and a *ShiftConfiguration* instance using the sent information:

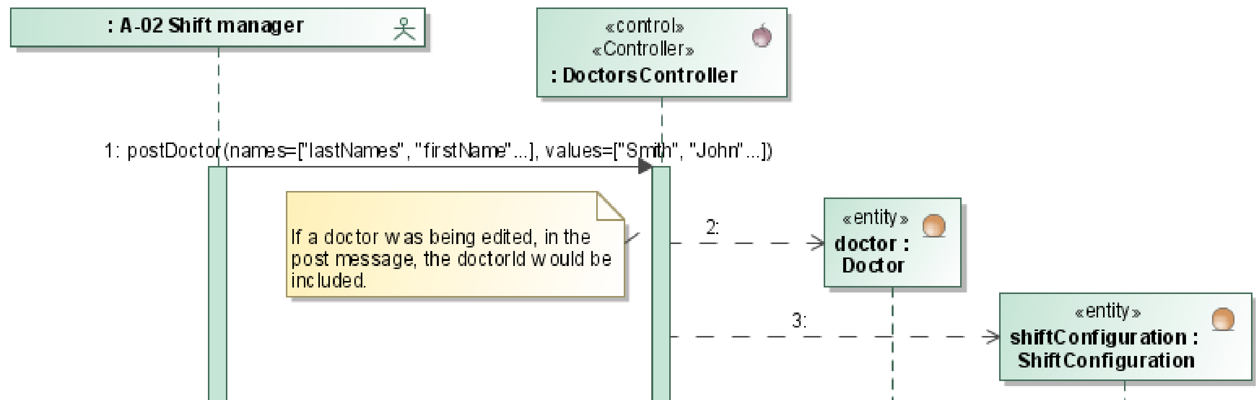


Figure 43: Web application - DoctorsController.newDoctor sequence - Submit form

Then, the *DoctorsController* requests the *DoctorService* to persist the *Doctor*:

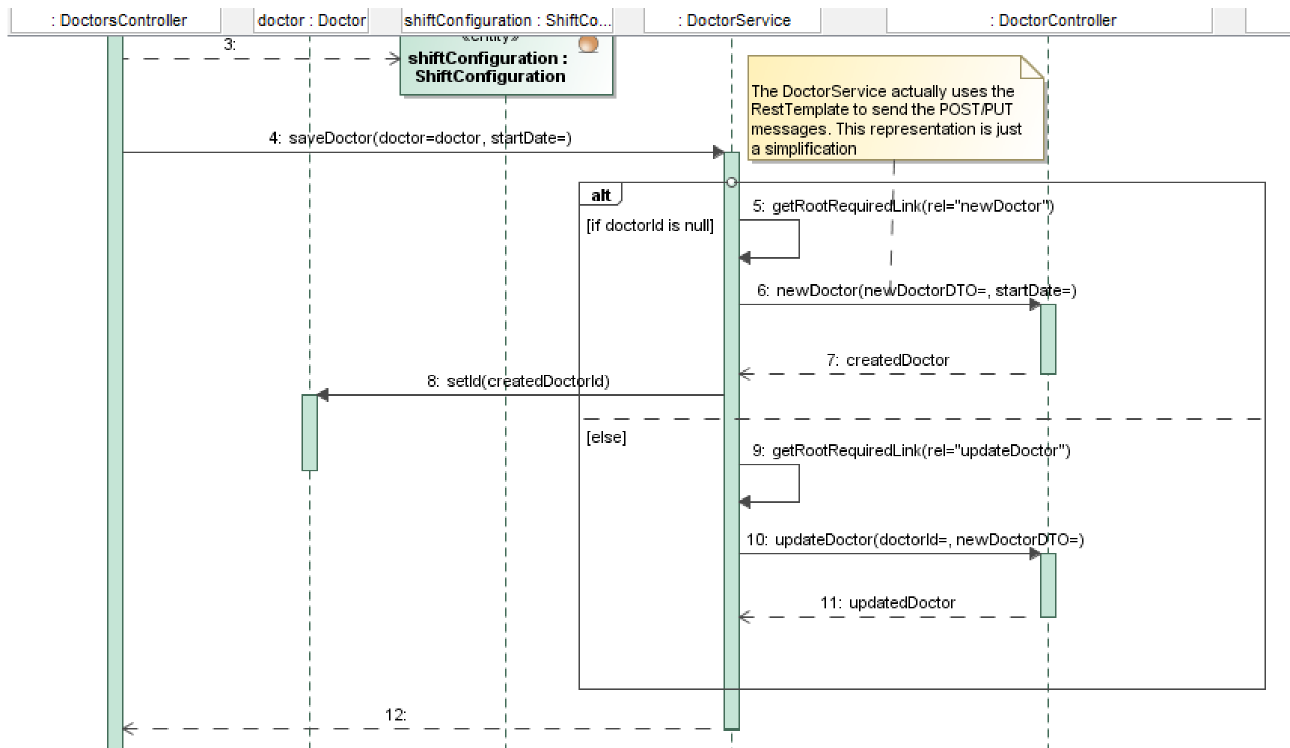


Figure 44: Web application - DoctorsController.newDoctor sequence - Persist doctor

Then, the *ShiftConfiguration* has to be persisted as well:

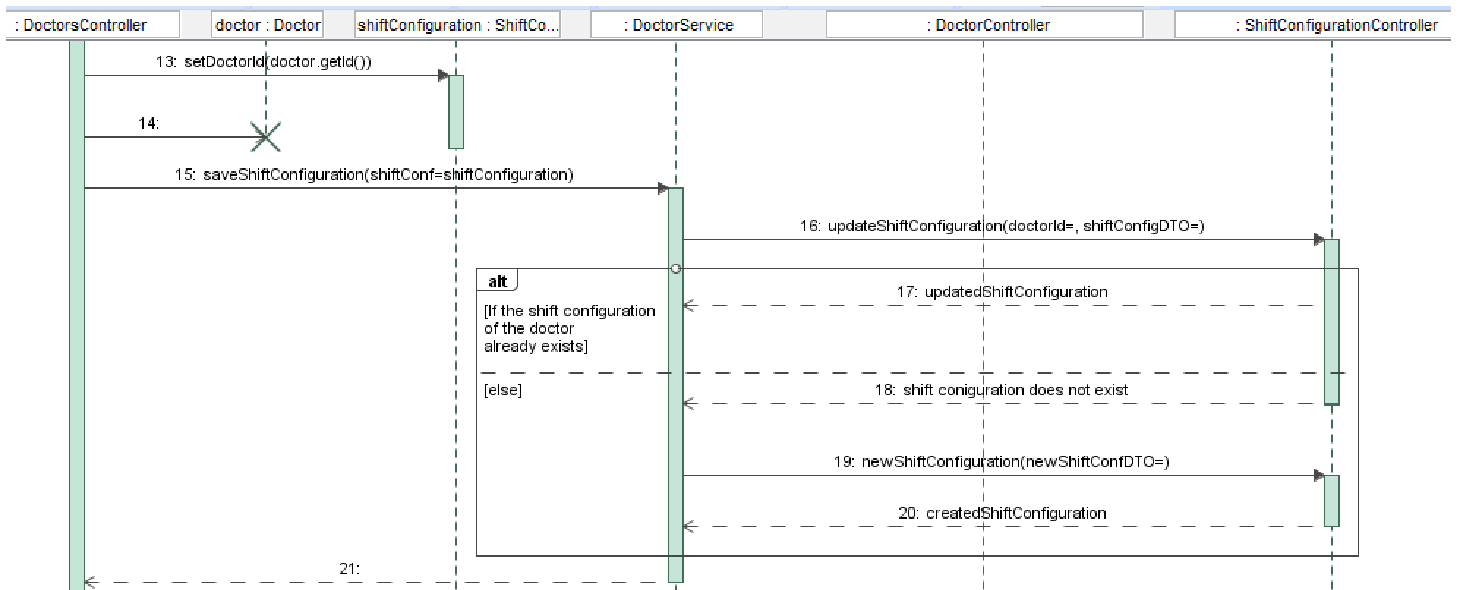


Figure 45: Web application - DoctorsController.newDoctor sequence - Persist shift configuration

Lastly, the user is presented with the doctor template:

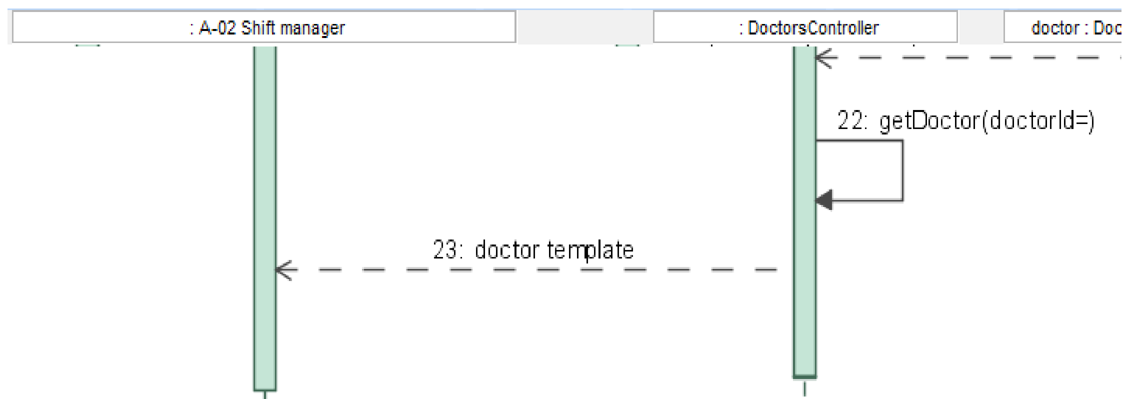


Figure 46: Web application - DoctorsController.newDoctor sequence - Response

The sequences of messages exchanged by the *ScheduleController* to get a list of all schedules, the information of a specific schedule or the form to generate a schedule are fairly similar to the previous diagrams. For this reason, these diagrams will not be shown.

Now, we will present the sequence of messages exchanged to generate a new schedule:

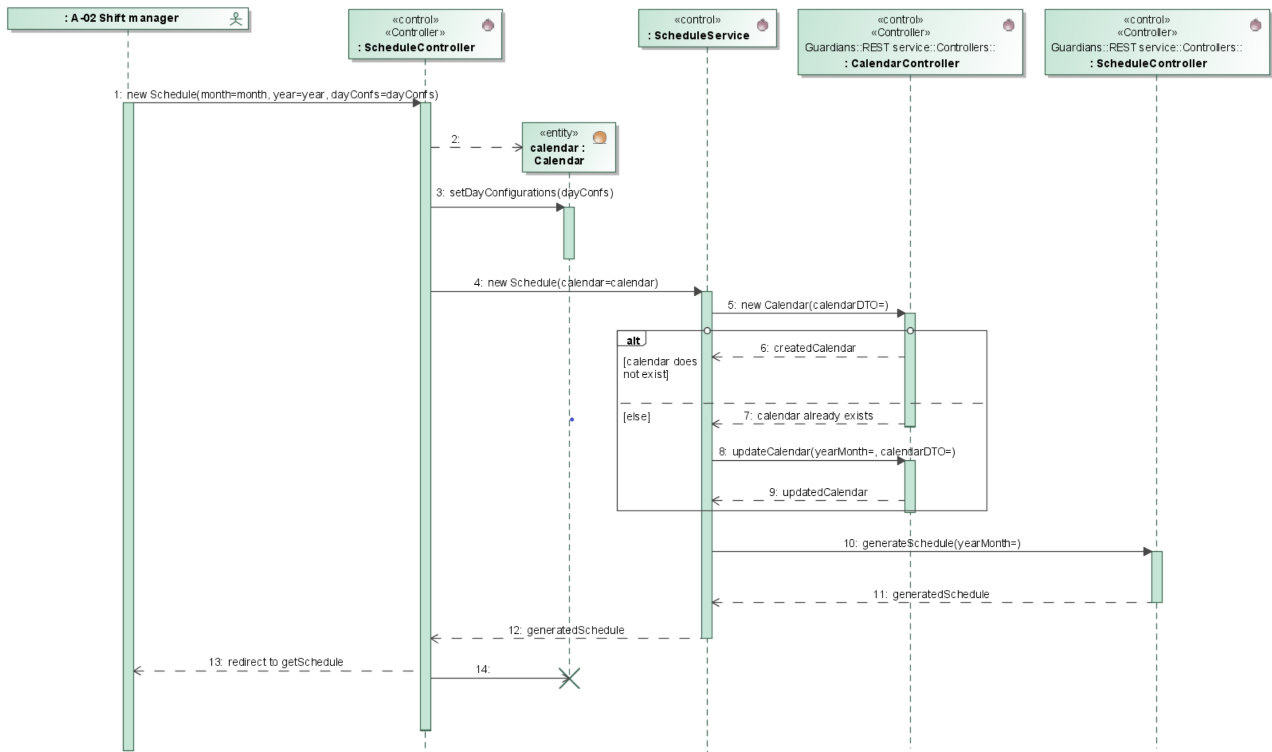


Figure 47: Web application - ScheduleController.newSchedule sequence

First of all, the *ScheduleController* creates a *Calendar* with the information sent in the POST request:

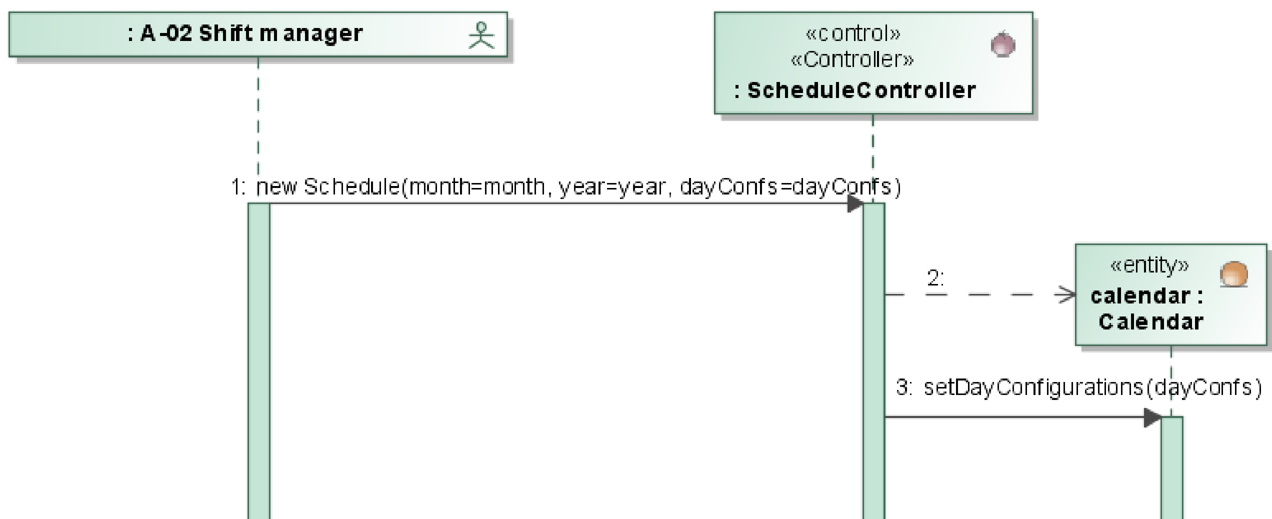


Figure 48: Web application - ScheduleController.newSchedule sequence - User submits form

Then, the *ScheduleController* requests the *ScheduleService* to generate a Schedule:

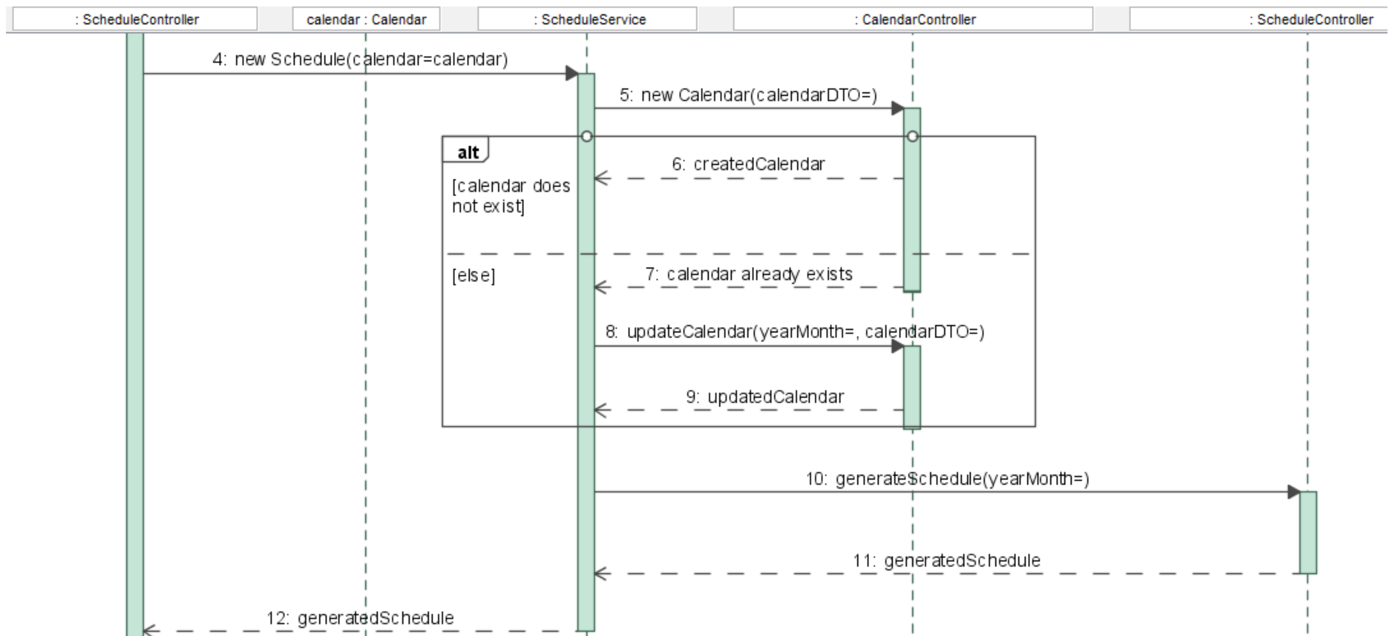


Figure 49: Web application - ScheduleController.newSchedule sequence - Request schedule generation

Lastly, the user is redirected to get the schedule information:

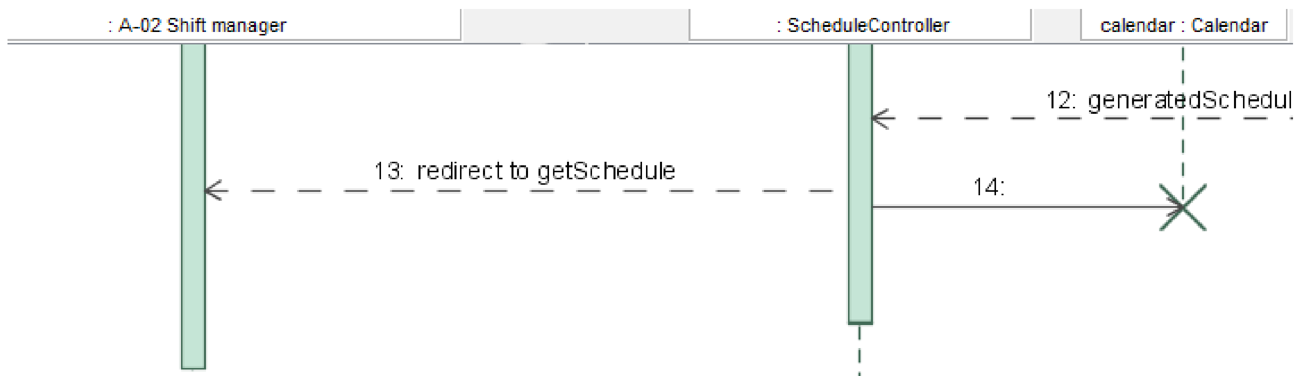


Figure 50: Web application - ScheduleController.newSchedule sequence - Redirect user

5. CONCLUSIONS AND FUTURE WORK

Looking back at the project objectives, we can say we have successfully completed them:

The first objective of the project was to automate and reduce the amount of time needed to schedule the doctor's shifts. With the developed application, in the simplest scenario, we can have a schedule generated in less than a minute. The simplest scenario would be not wanting to change the doctor's shift configurations nor the default generation parameters (working days, minimum number of regular-shifts per day…).

Another project's objective was having the system store the scheduling configuration, and allowing changing it. This is what we have defined as the doctor's shift configuration. Hence, we can say we have also achieved this goal.

Being able to retrieve previous schedules was also one of this project's purposes. This is the responsibility of the web application's "Schedules" page: it allows the end user to query the REST service for a specific schedule and displays it in a readable format. Therefore, we can say we have completed this objective.

We have also tried to make the architecture flexible enough so that new features and changes will be developed with more ease. For example, let's say we developed a new system that allows doctors to change shifts. Then, to integrate it with the current application, it would just have to consume the REST API (we may also need to add new resources that would allow keeping track of shift changes).

Lastly, and most importantly, the main goal of this project was to develop a first functioning prototype that will allow us to validate the system. This is, we can now show system to the end user, get feedback on the features that have to be improved, and suggestions on the functionality that is yet to be added.

All in all, we could argue this project has been a success.

5.1. Future work

As the current system is just a prototype, there are many requirements that are yet to be met. For this reason, this section will describe future lines of work to improve and complete this application. Moreover, as there are three different problems to be solved (Scheduling problem, Pager assignment and Management problem), we will describe possible lines of work on each of them.

5.1.1. Scheduling problem

Regarding the scheduling problem, the current requirements have been met. However, there are additional features that can be added to improve it (some of which, we have already started to develop). These will be explained in the following sections.

5.1.1.1. Edit a schedule

A use case that will have to be implemented before the management problem can be solved is allowing the user to edit a schedule. This is, after a schedule is generated, the shift manager (Table 2: A-02 – Shift manager) might want to change it.

From Figure 11: Desired scheduling procedure, the edit schedule use case would correspond to:

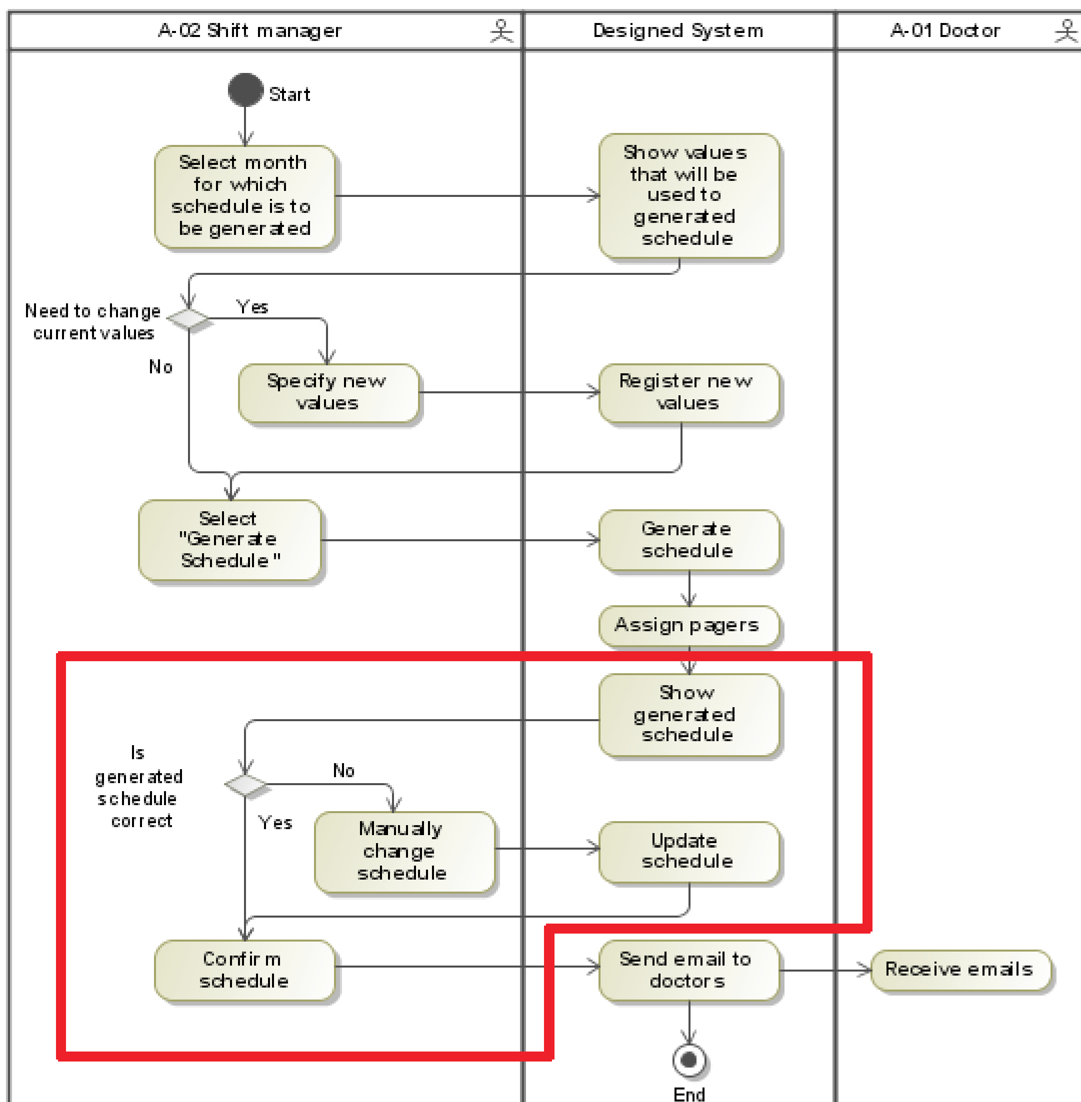


Figure 51: Desired scheduling procedure - Edit schedule

Note that, although sending an email to the doctors would already be part of the management problem, the system has to register the schedule has been confirmed. This is to not let the schedule be updated manually any more after it has been sent to the doctors.

Although it was not in the scope of this project, we have started implementing this use case. In this section, we will describe the approach we have decided and how its development can be continued:

The current shift manager (Table 2: A-02 – Shift manager) wills to keep using Excel to edit schedules. This is, the shift manager would provide the application an Excel file, and the application would have to extract the edited schedule from it. However, Excel has nothing to do with the current application. Therefore, to reduce its usage throughout the system, we will give the responsibility of handling Excel files to the Web Application. This way, the REST service does not need to be aware of the existence of Excel files.

On 1.2 Current situation we showed a sample of one of the Excel files used to schedule doctor's shifts:

16	A	B	A #	C #	
17	D	E	F #	E #	
18	G	H	I #	J #	
19	I	K	I #	K #	L #
20	M	N	M #	O #	
21	P	Q			
22	F	R			

Figure 52: Excel file example

However, we did not explain the meaning of the colour scheme. Now, to be able to understand how to parse an Excel file into the representation of a schedule (Table 40: Schedule resource), we do have to know how this file is created and the meaning of the different colours:

- Each row represents the shifts of a certain day. The day number is indicated by the first column. E.g. on the image above, the first row corresponds to the shifts that took place the 16th of a certain month.

- Each non-empty cell of a given row represents a shift on a certain day. E.g. on day 17 there were four shifts scheduled, and on day 21 there were only two.

- Doctors are identified in the Excel by their surnames. This is, A, B, C, D… would be the surnames of the corresponding doctors.

- Non working days are represented on blue (The whole row has to be coloured). The cells of a non working day represent the CSs that took place that day. E.g. days 21 and 22 were non working days. Doctors P and Q had a CS the 21st and doctors F and R had a CS the 22nd.

- Working days are not coloured in blue. Specifically, CS are represented with light grey, regular shift with orange (light and dark), and consultations with salmon.

- On NCSs, the sign '#' is added after the doctors surname. E.g. if doctor 'A' has a non cycle shift, we would write "A #" in the corresponding cell.

- Regular shifts should be coloured on **light orange** if the corresponding doctor **does not have** a CS that day, or coloured in **dark orange** if the doctor **does have** CS.

- On working days, the order on which shifts are represented is always the same: first, CS; then, regular shifts; and lastly, consultations.

Now, we will explain how this use case will be implemented:

To isolate the handling of Excel files from the rest of the web application, we have created a new service class:

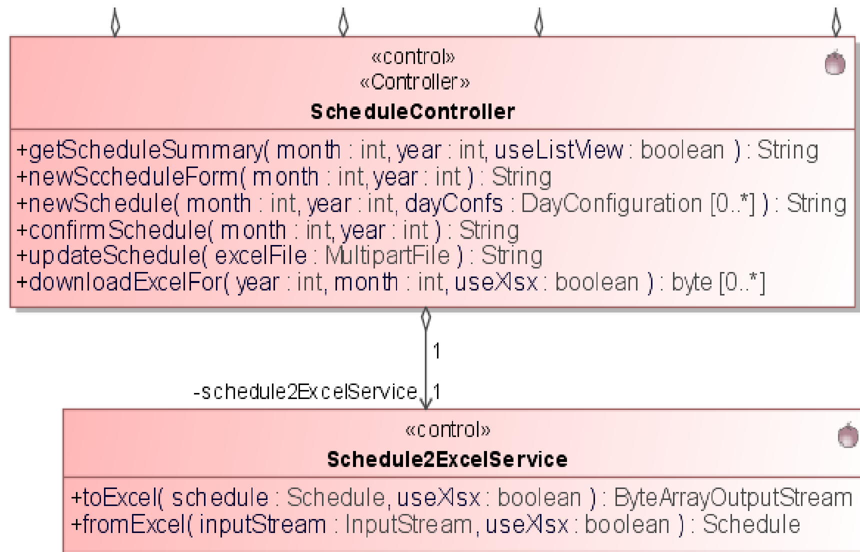


Figure 53: Schedule2ExcelService class

Note the *ScheduleController* now has two new methods: One to handle request to convert a Schedule to an Excel file, and another one to update a Schedule given an Excel file. The behaviour of the system on both of these cases will be as shown in the following diagrams.

Note that, to read and create Excel files, we will be using Apache POI [74]. We can find a simple tutorial on how to use this library at [75].

The development on the edit schedule feature has already been started. Specifically, the user can already download the Excel representing a specific schedule, and can upload an Excel file to the web application. However, the *ScheduleService.updateSchedule* and *Schedule2ExcelService.fromExcel* methods still have to be developed. The *ScheduleController.updateSchedule* already uses the previous methods, but they do not do anything yet.

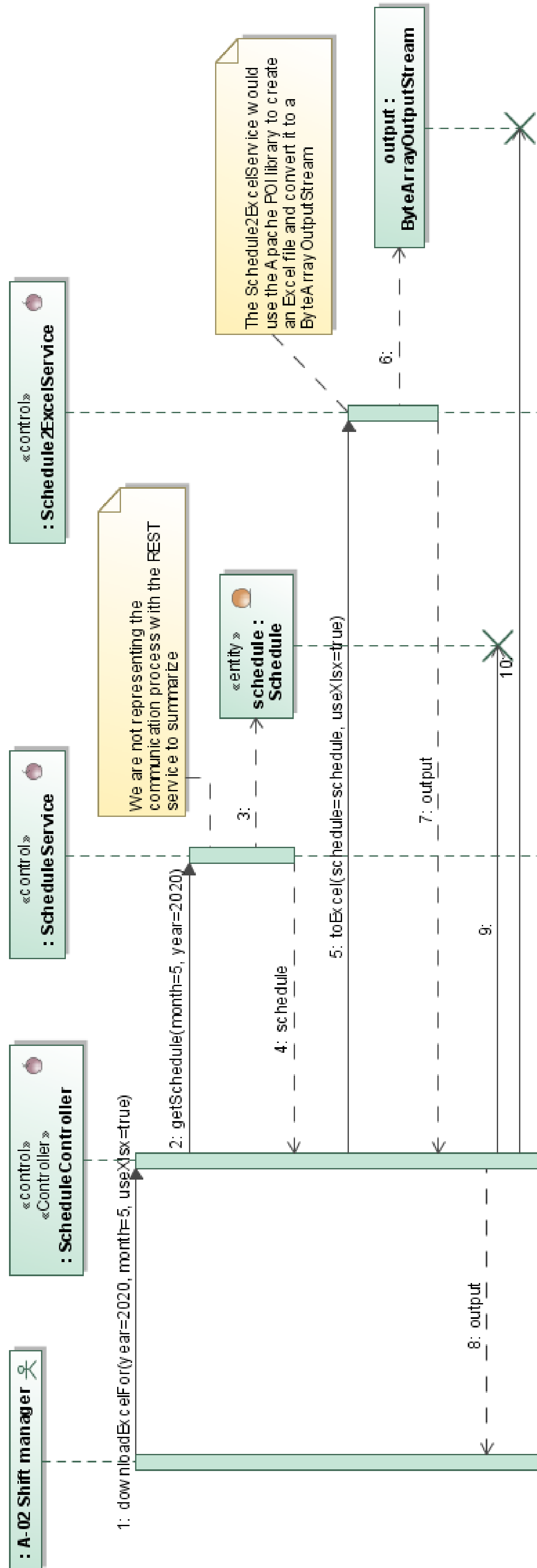


Figure 54: ScheduleController.downloadExcelFor

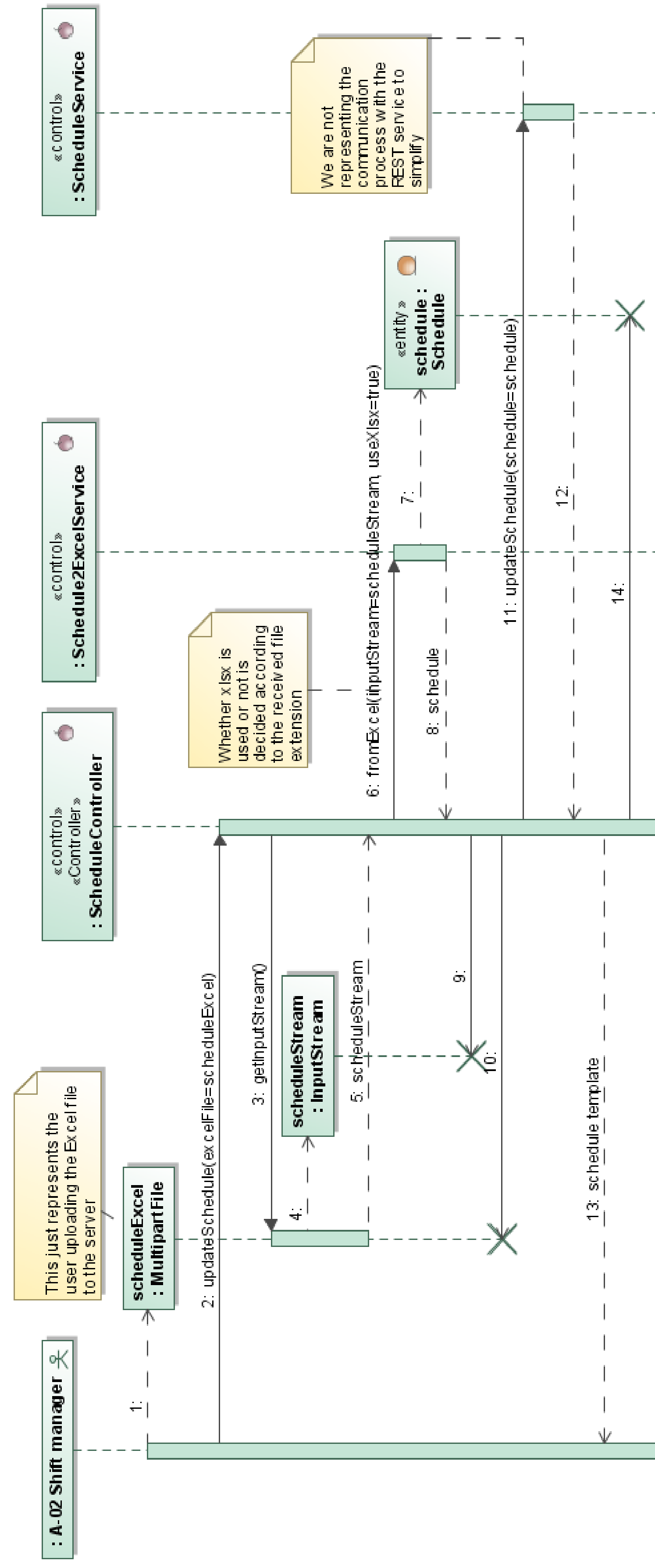


Figure 55: ScheduleController.updateSchedule

5.1.1.2. Doctor's absences

A doctor's absence is a period of time in which the doctor will not be working. This time could be due to holidays or to a different reason such as time off sick.

The REST service currently takes into account absences. This is, a *doctor* resource (Table 33: Doctor resource) can have an absence associated to it. However, doctors may have several absence periods within one month, and the current model does not take that into account.

Moreover, the neither the web application nor the scheduler take absences into account. This is, the web application should allow the user to add or delete doctor's absences. Then, the scheduler algorithm should also take into account that a doctor cannot have any shift during an absence period.

Furthermore, if a doctor is absent, another doctor should do their shifts. This, again, is not currently taken into account by the system and is left as a future improvement.

5.1.1.3. Mandatory / Unavailable shifts

Another feature that can be added to make the system more flexible would be creating a new type of shift preference (Table 14: BR-SCH-09 - Shift preferences). Mandatory and unavailable shifts would be similar to the current wanted or unwanted shifts, however, they would have a different meaning: A mandatory shift would represent the corresponding doctor <u>has to have</u> a regular-shift the specified day, whereas an unavailable shift would represent the doctor <u>must not have</u> a regular-shift the given day.

This new feature would make the system more flexible by allowing the user to choose certain shifts that have to (or must not) be done by certain doctors.

5.1.1.4. Consultation preferences

Currently, by the examples we have been given on shift preferences, we have only implemented wanted consultations. However, we could also add unwanted, mandatory or unavailable consultations. This would give the user more control on how consultations are scheduled.

5.1.2. Pager assignment problem

The pager assignment problem can be solved after the scheduler has finished scheduling shifts. The scheduler could assign the pagers based on the generated schedule: we can already identify the doctors that can have the pager (Table 19: BR-PG-03 – Pager allowed doctors), and on which days it has to be assigned (Table 17: BR-PG-01 – Pager allowed days). To know which doctors should be assigned the pager twice (Table 22: BR-PG-06 - Second pager assignment order), we could keep track of the last time each doctor had the it assigned twice (for example, by adding a new property to the doctor's shift configuration). Lastly, the REST service could update the dates on which doctors had to have the pager assigned twice (based on the assigned pagers).

Note this is just a suggestion, and it would have to be thought thoroughly before implementing it. For example, we might also want to add pager preferences similar to Table 14: BR-SCH-09 - Shift preferences.

5.1.3. Management problem

The management problem, we could divide it into two separate parts:

- Notifying the doctors of their schedule
- Allowing shift changes

5.1.3.1. Notifying doctors of their schedule

Notifying doctors of their schedule consists on both, sending them an email whenever the schedule of a month is confirmed, and allowing them to have the most updated version of each schedule (after shift changes have been applied).

For the first part, sending the doctors an email, we could have an email server to send it. Its services would be requested after a schedule is confirmed, as shown on Figure 51: Desired scheduling procedure - Edit schedule. Another idea would be using the services of an email provider. For example, Google has an available REST API that allows sending mails from a gmail account. See [76].

Regarding the second part, we could make use of a CalDAV [77] server to provide access to the schedules. Then, the doctors would need a CalDAV client. This way, to allow the doctors to have the most updated version of the schedule, we would only have to update the CalDAV server whenever a schedule changes.

For example, while we were studying this project, one of the alternatives we came up with was the Open Source Baikal CalDAV project. Its home page can be found at [78]. However, as finding a reliable CalDAV client or implementing one could be a costly or lengthy task. Another alternative would be using a calendar provider. For example, Google Calendar has a REST API that allows managing calendars. See [79]. They also provide libraries for different languages such a Python or Java to consume their API (they can be found on the previous reference).

5.1.3.2. Allowing shift changes

Allowing shift changes is a whole new addition to the application, as it introduces a new actor into the system: Table 1: A-01 - Doctor.

One of the solutions we came up with, and that can be explored when designing a solution to this problem, would be using a chatbot. The doctors would talk to this chatbot to request a shift change. For example, they could send the bot a command such as: "change regular-shift 2020-06-15"[xii]. Then, the bot could notify the other doctors there is a person willing to change one of their shifts. Afterwards, if a doctor is willing to change the proposed shift, they could send the bot a command such as "accept-change regular-shift 2020-06-15". Finally, the bot would have to communicate with the main application to report the change.

For example, an option we considered was using the API to develop bots [80] provided by Telegram [81]. Moreover, there are different libraries that already provide a simple wrapper around this API. For example, python-telegram-bot [82] is a simple-to-use Python module that supports the main operations of the Telegram's Bots API.

[xii] In these illustrative examples, we will assume the communication between the doctors and the chatbot would be through commands. We could also allow the doctors to use natural language but then, we would have to use Natural Language Processing (NLP) techniques. On this simple discussion, we will assume the communication will be through commands, but NLP would have to be considered as an option if this feature was to be implemented.

REFERENCES

[1] Bradley, S.; Hax, A; and Magnanti, T. 1992. *Applied Mathematical Programming*. Reading, Mass: Addison-Wesley. Chapters 1 and 9

[2] I. Restrepo, María; Rousseau, Louis-Martin; Vallée, Jonathan. 2019. *Home healthcare integrated staffing and scheduling*. Omega

[3] Ceric, Arnela. 2015. *Bringing together evaluation and management of ICT value: A systems theory approach*. Electronic Journal of IS Evaluation. 18.

[4] Omg.org. 2020. *About The Unified Modeling Language Specification Version 2.5*. [online] Available at: <https://www.omg.org/spec/UML/2.5/About-UML/> [Accessed 27 May 2020].

[5] Sommerville, I. 2002. *Ingeniería de software* ([2a ed. en español]). Méxic. Pearson Educación.

[6] Docs.spring.io. 2020. 17. Web MVC Framework. [online] Available at: <https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html> [Accessed 3 June 2020].

[7] Baeldung. 2020. *The DAO Pattern In Java | Baeldung*. [online] Available at: <https://www.baeldung.com/java-dao-pattern> [Accessed 3 June 2020].

[8] Martinfowler.com. 2020. *P Of EAA: Data Transfer Object*. [online] Available at: <https://martinfowler.com/eaaCatalog/dataTransferObject.html> [Accessed 3 June 2020].

[9] Wiki.python.org. 2020. *Beginnersguide/Overview - Python Wiki*. [online] Available at: <https://wiki.python.org/moin/BeginnersGuide/Overview> [Accessed 22 May 2020].

[10] Docs.python.org. 2020. *3.7.7 Documentation*. [online] Available at: <https://docs.python.org/3.7/> [Accessed 23 May 2020].

[11] Docs.python.org. 2020. 3. An Informal Introduction To Python — Python 3.8.3 Documentation. [online] Available at: <https://docs.python.org/3/tutorial/introduction.html#lists> [Accessed 11 June 2020].

[12] Docs.python.org. 2020. 5. Data Structures — Python 3.8.3 Documentation. [online] Available at: <https://docs.python.org/3/tutorial/datastructures.html#dictionaries> [Accessed 11 June 2020].

[13] Docs.python.org. 2020. What'S New In Python 2.0 — Python 3.8.3 Documentation. [online] Available at: <https://docs.python.org/3/whatsnew/2.0.html#list-comprehensions> [Accessed 11 June 2020].

[14] Python.org. 2020. PEP 274 -- Dict Comprehensions. [online] Available at: <https://www.python.org/dev/peps/pep-0274/> [Accessed 11 June 2020].

[15] Docs.python.org. 2020. Json — JSON Encoder And Decoder — Python 3.8.3 Documentation. [online] Available at: <https://docs.python.org/3/library/json.html> [Accessed 11 June 2020].

[16] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. [online] Available at: <https://tools.ietf.org/html/rfc7159> [Accessed 28 May 2020].

[17] Google Developers. 2020. *OR-Tools | Google Developers*. [online] Available at: <https://developers.google.com/optimization> [Accessed 23 May 2020].

[18] Google Developers. 2020. Python Reference: CP-SAT | OR-Tools | Google Developers. [online] Available at: <https://developers.google.com/optimization/reference/python/sat/python/cp_model> [Accessed 11 June 2020].

[19] Docs.python.org. 2020. What'S New In Python 3.6 — Python 3.8.3 Documentation. [online] Available at: <https://docs.python.org/3/whatsnew/3.6.html#pep-498-formatted-string-literals> [Accessed 11 June 2020].

[20] GitHub. 2020. Google/Or-Tools. [online] Available at:
<https://github.com/google/or-tools/blob/master/examples/python/shift_scheduling_sat.py> [Accessed 11 June 2020].
[21] W3schools.com. 2020. *Introduction To Java*. [online] Available at:
<https://www.w3schools.com/java/java_intro.asp> [Accessed 23 May 2020].
[22] Docs.oracle.com. 2020. *Java Platform SE 8*. [online] Available at:
<https://docs.oracle.com/javase/8/docs/api/> [Accessed 23 May 2020].
[23] Docs.oracle.com. 2020. Anonymous Classes (The Java™ Tutorials > Learning The Java Language > Classes And Objects). [online] Available at:
<https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html> [Accessed 25 June 2020].
[24] Docs.oracle.com. 2020. Lambda Expressions (The Java™ Tutorials > Learning The Java Language > Classes And Objects). [online] Available at:
<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html> [Accessed 25 June 2020].
[25] Cgi.csc.liv.ac.uk. 2020. 2Cs24 Declarative. [online] Available at:
<https://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html> [Accessed 11 June 2020].
[26] Docs.oracle.com. 2020. Java.Util.Stream (Java Platform SE 8). [online] Available at:
<https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html> [Accessed 25 June 2020].
[27] Web.mit.edu. 2020. Reading 25: Map, Filter, Reduce. [online] Available at:
<https://web.mit.edu/6.005/www/fa15/classes/25-map-filter-reduce/> [Accessed 11 June 2020].
[28] Docs.oracle.com. 2020. Lesson: Annotations (The Java™ Tutorials > Learning The Java Language). [online] Available at:
<https://docs.oracle.com/javase/tutorial/java/annotations/> [Accessed 23 June 2020].
[29] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral Dissertation. University of California, Irvine. September 2000. Chapter 5. [online] Available at:
<http://roy.gbiv.com/pubs/dissertatio.n/top.htm> [Accessed 22 May 2020]
[30] Fielding, R.; J. Reschke; *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. [online] Available at:
<https://tools.ietf.org/html/rfc7231 > [Accessed 22 May 2020].
[31] W3.org. 2020. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. [online] Available at:
<https://www.w3.org/TR/xmlschema11-1/> [Accessed 23 June 2020].
[32] M. Kelly, *JSON Hypertext Application Language*. Draft-kelly-json-hal-08. [online] Available at:
<https://tools.ietf.org/html/draft-kelly-json-hal-08 > [Accessed 28 May 2020].
[33] Spring. 2020. *Spring Makes Java Simple*. [online] Available at:
<https://spring.io/> [Accessed 23 May 2020].
[34] Spring Initializr. 2020. *Spring Initializr*. [online] Available at:
<https://start.spring.io/> [Accessed 2 June 2020].
[35] Porter, B., Zyl, J. and Lamy, O., 2020. Maven – Welcome To Apache Maven. [online] Maven.apache.org. Available at:
<https://maven.apache.org/> [Accessed 12 June 2020].
[36] Spring.io. 2020. Building REST Services With Spring. [online] Available at:
<https://spring.io/guides/tutorials/bookmarks/> [Accessed 17 June 2020].
[37] Youtube. *REST Beyond the Obvious – API Design for Ever-Evolving SystemsI.* [online] Available at:
<https://www.youtube.com/watch?v=WDBUlu_lYas > [Accessed 17 June 2020].
[38] Spring.io. 2020. Accessing Data With Mysql. [online] Available at:
<https://spring.io/guides/gs/accessing-data-mysql/> [Accessed 17 June 2020].
[39] Baeldung. 2020. How To Do @Async In Spring | Baeldung. [online] Available at:
<https://www.baeldung.com/spring-async> [Accessed 18 June 2020].
[40] Baeldung. 2020. Guide To Internationalization In Spring Boot | Baeldung. [online] Available at:
<https://www.baeldung.com/spring-boot-internationalization> [Accessed 18 June 2020].
[41] Baeldung. 2020. Spring Boot: Customize Whitelabel Error Page | Baeldung. [online] Available at:
<https://www.baeldung.com/spring-boot-custom-error-page> [Accessed 18 June 2020].
[42] Docs.spring.io. 2020. Web On Servlet Stack. [online] Available at:
<https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html> [Accessed 12 June

2020].
[43] Stackify. 2020. How Spring MVC Really Works. [online] Available at: <https://stackify.com/spring-mvc/> [Accessed 12 June 2020].
[44] Project, A., 2020. Apache Tomcat® - Welcome!. [online] Tomcat.apache.org. Available at: <https://tomcat.apache.org/index.html> [Accessed 12 June 2020].
[45] Spring.io. 2020. Spring Boot. [online] Available at: <https://spring.io/projects/spring-boot> [Accessed 12 June 2020].
[46] Mit.edu. 2020. Curl Man Page. [online] Available at: <https://www.mit.edu/afs.new/sipb/user/ssen/src/curl-7.11.1/docs/curl.html> [Accessed 12 June 2020].
[47] Baeldung. 2020. Spring Dependency Injection | Baeldung. [online] Available at: <https://www.baeldung.com/spring-dependency-injection> [Accessed 12 June 2020].
[48] Docs.spring.io. 2020. Core Technologies. [online] Available at: <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-collaborators> [Accessed 12 June 2020].
[49] Projectlombok.org. 2020. Project Lombok. [online] Available at: <https://projectlombok.org/> [Accessed 12 June 2020].
[50] Slf4j.org. 2020. Logger (SLF4J 2.0.0-Alpha0 API). [online] Available at: <http://www.slf4j.org/apidocs/org/slf4j/Logger.html> [Accessed 12 June 2020].
[51] Spring.io. 2020. *Spring Data JPA*. [online] Available at: <https://spring.io/projects/spring-data-jpa> [Accessed 2 June 2020].
[52] Docs.oracle.com. 2020. Javax.Persistence (Java(TM) EE 7 Specification Apis). [online] Available at: <https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html> [Accessed 17 June 2020].
[53] Docs.spring.io. 2020. Jparepository (Spring Data JPA 2.3.1.RELEASE API). [online] Available at: <https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html> [Accessed 17 June 2020].
[54] Docs.oracle.com. 2020. Validator (Java(TM) EE 7 Specification Apis). [online] Available at: <https://docs.oracle.com/javaee/7/api/javax/validation/Validator.html> [Accessed 17 June 2020].
[55] Hardy Ferentschik, G., 2020. Hibernate Validator 6.1.5.Final - Jakarta Bean Validation Reference Implementation: Reference Guide. [online] Docs.jboss.org. Available at: <https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/?v=6.1#section-validating-bean-constraints> [Accessed 17 June 2020].
[56] Docs.oracle.com. 2020. Valid (Java(TM) EE 7 Specification Apis). [online] Available at: <https://docs.oracle.com/javaee/7/api/javax/validation/Valid.html> [Accessed 17 June 2020].
[57] Javaee.github.io. 2020. Javax.Validation.Constraints (Java(TM) EE 8 Specification Apis). [online] Available at: <https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-frame.html> [Accessed 17 June 2020].
[58] Hardy Ferentschik, G., 2020. Hibernate Validator 6.1.5.Final - Jakarta Bean Validation Reference Implementation: Reference Guide. [online] Docs.jboss.org. Available at: <https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/?v=6.1#validator-customconstraints-simple> [Accessed 17 June 2020].
[59] Docs.jboss.org. 2020. Chapter 2. Mapping Entities. [online] Available at: <https://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html> [Accessed 17 June 2020].
[60] Balasubramaniam, V., 2020. Composite Primary Keys In JPA | Baeldung. [online] Baeldung. Available at: <https://www.baeldung.com/jpa-composite-primary-keys> [Accessed 17 June 2020].
[61] Spring.io. 2020. *Spring HATEOAS*. [online] Available at: <https://spring.io/projects/spring-hateoas> [Accessed 2 June 2020].
[62] Mode, D., 2020. HATEOAS Driven REST Apis – REST API Tutorial. [online] Restfulapi.net. Available at: <https://restfulapi.net/hateoas/> [Accessed 18 June 2020].
[63] Docs.spring.io. 2020. Entitymodel (Spring HATEOAS 1.1.0.RELEASE API). [online] Available at: <https://docs.spring.io/spring-hateoas/docs/current/api/org/springframework/hateoas/EntityModel.html> [Accessed 18 June 2020].

[64] Docs.spring.io. 2020. Collectionmodel (Spring HATEOAS 1.1.0.RELEASE API). [online] Available at: <https://docs.spring.io/spring-hateoas/docs/current/api/org/springframework/hateoas/CollectionModel.html> [Accessed 18 June 2020].

[65] Docs.spring.io. 2020. Representationmodelassembler (Spring HATEOAS 1.1.0.RELEASE API). [online] Available at: <https://docs.spring.io/spring-hateoas/docs/current/api/org/springframework/hateoas/server/RepresentationModelAssembler.html> [Accessed 18 June 2020].

[66] Docs.spring.io. 2020. Traverson (Spring HATEOAS 1.1.0.RELEASE API). [online] Available at: <https://docs.spring.io/spring-hateoas/docs/current/api/org/springframework/hateoas/client/Traverson.html> [Accessed 18 June 2020].

[67] Oliver Gierke, J., 2020. Spring HATEOAS - Reference Documentation. [online] Docs.spring.io. Available at: <https://docs.spring.io/spring-hateoas/docs/current/reference/html/> [Accessed 18 June 2020].

[68] Docs.spring.io. 2020. Resttemplate (Spring Framework 5.2.7.RELEASE API). [online] Available at: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html> [Accessed 18 June 2020].

[69] Baeldung. 2020. A Guide To The Resttemplate | Baeldung. [online] Available at: <https://www.baeldung.com/rest-template> [Accessed 18 June 2020].

[70] Thymeleaf.org. 2020. Tutorial: Using Thymeleaf. [online] Available at: <https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html> [Accessed 18 June 2020].

[71] Mysql.com. 2020. Mysql. [online] Available at: <https://www.mysql.com/> [Accessed 3 June 2020].

[72] Docs.oracle.com. 2020. Supplier (Java Platform SE 8). [online] Available at: <https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html> [Accessed 17 June 2020].

[73] GitHub. 2020. Miggoncan/Guardiansscheduler. [online] Available at: <https://github.com/miggoncan/guardiansScheduler> [Accessed 10 June 2020].

[74] Poi.apache.org. 2020. Apache POI - The Java API For Microsoft Documents. [online] Available at: <https://poi.apache.org/> [Accessed 4 July 2020].

[75] Baeldung. 2020. Working With Microsoft Excel In Java | Baeldung. [online] Available at: <https://www.baeldung.com/java-microsoft-excel#apache-poi> [Accessed 4 July 2020].

[76] Google Developers. 2020. Gmail API | Google Developers. [online] Available at: <https://developers.google.com/gmail/api> [Accessed 4 July 2020].

[77] C. Daboo. *Calendaring Extensions to WebDAV (CalDAV)*. RFC 4791. [online] Available at: <https://tools.ietf.org/html/rfc4791> [Accessed 4 July 2020].

[78] Sabre.io. 2020. Baïkal - Baikal. [online] Available at: <https://sabre.io/baikal/> [Accessed 4 July 2020].

[79] Google Developers. 2020. Calendar API | Google Developers. [online] Available at: <https://developers.google.com/calendar> [Accessed 4 July 2020].

[80] Core.telegram.org. 2020. Bots: An Introduction For Developers. [online] Available at: <https://core.telegram.org/bots> [Accessed 4 July 2020].

[81] Telegram. 2020. Telegram – A New Era Of Messaging. [online] Available at: <https://telegram.org/> [Accessed 4 July 2020].

[82] Python-telegram-bot.org. 2020. Python-Telegram-Bot. [online] Available at: <https://python-telegram-bot.org/> [Accessed 4 July 2020].

APPENDIX A – CODE

All of the code developed in this project can be found on these three GitHub repositories:

- RESTful service:

  https://github.com/miggoncan/guardiansRESTinterface [Accessed 10 June 2020]

- Scheduler:

  https://github.com/miggoncan/guardiansScheduler [Accessed 10 June 2020]

- Web application:

  https://github.com/miggoncan/guardiansWebapp [Accessed 10 June 2020]

- Deployment scripts and instructions:

  https://github.com/miggoncan/guardiansDeployment [Accessed 1 July 2020]

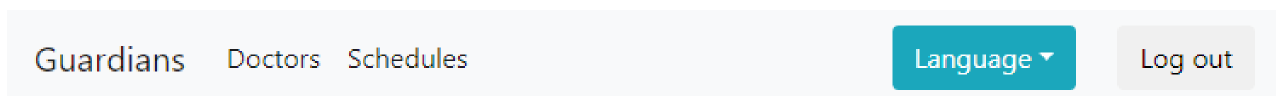
Note: The README.md file on the RESTful service repository contains a guide to set up the project for development. It also contains links to the JavaDocs of the project.

Appendix B – Web application usage

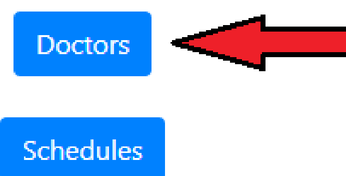
Note: As this project is being developed in English but the web application's users speak Spanish, the application's language can be easily changed at runtime between these two languages. See the "Language" drop-down button at the top right of the following image. Still, as this document is being written in English, this usage guide will be in the same language (A usage guide in Spanish will also be developed, but will not be included as part of this project).

1. Create a new doctor

1. From the homepage, click on 'Doctors':

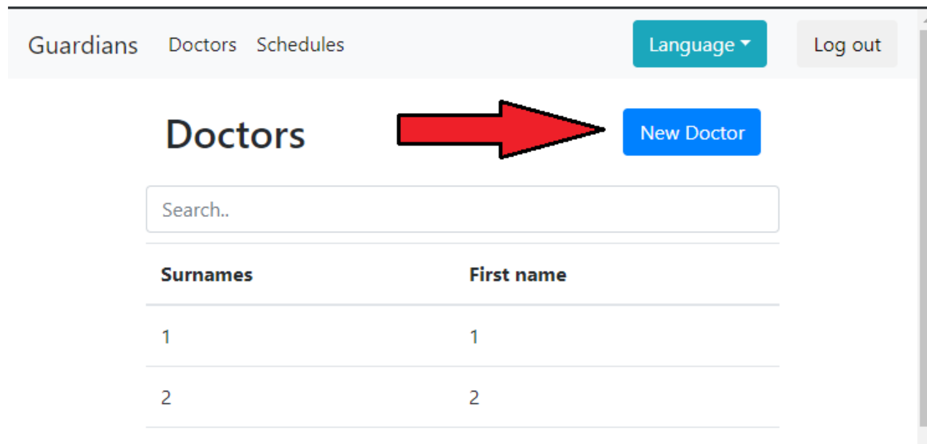


Welcome!

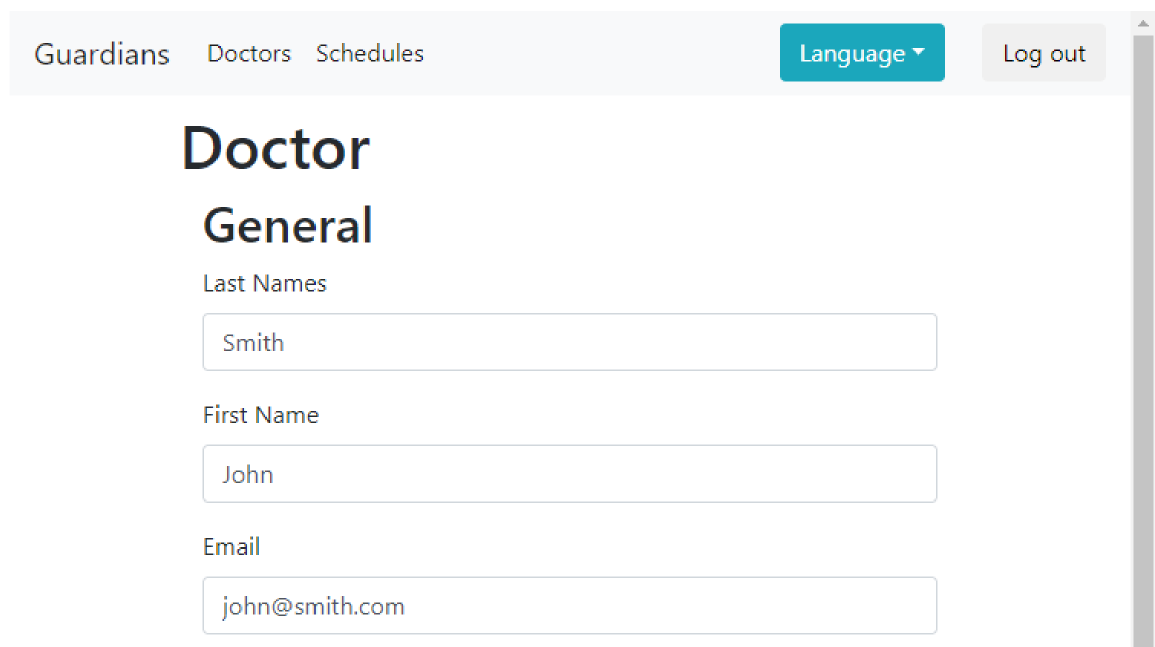


Schedules

2. Click on ‘New Doctor’ on the top right corner:



3. Fill in the general fields:



4. Select the first cyclic-shift of the doctor (if the doctor does not do cyclic-shifts, select the current date):

Start date

2020-06-20

« June 2020 »

Mo	Tu	We	Th	Fr	Sa	Su
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

ifts

Max

5. Select the shift preferences of the doctor and click the submit button:

Preferences

☑ Cyclic-shifts

☑ Shifts

☐ Shifts only the Cyclic-shifts

Min.

3

Max.

4

☑ Consultations

Number

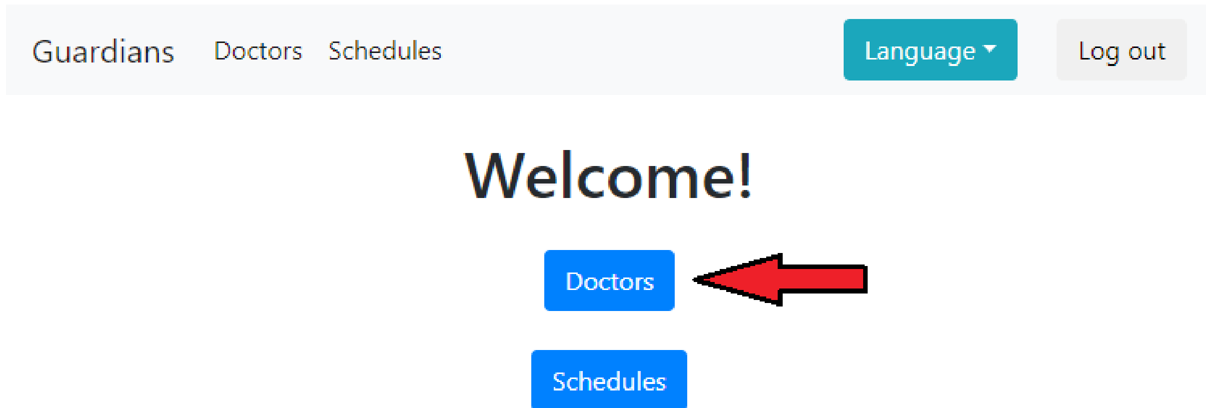
2

Show More

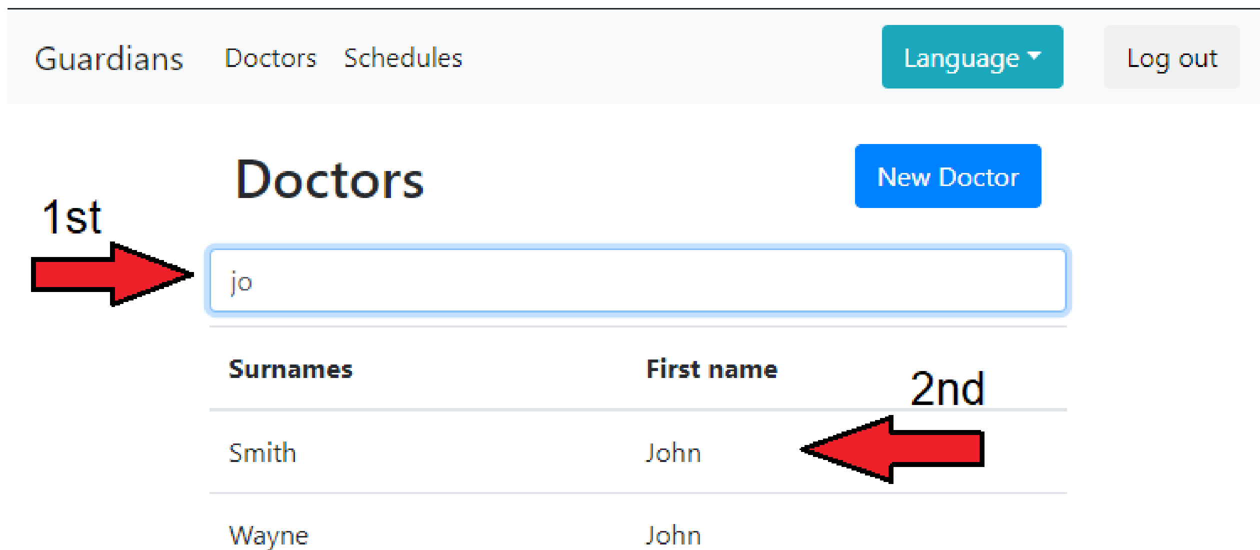
Cancel Submit

2. Edit a doctor

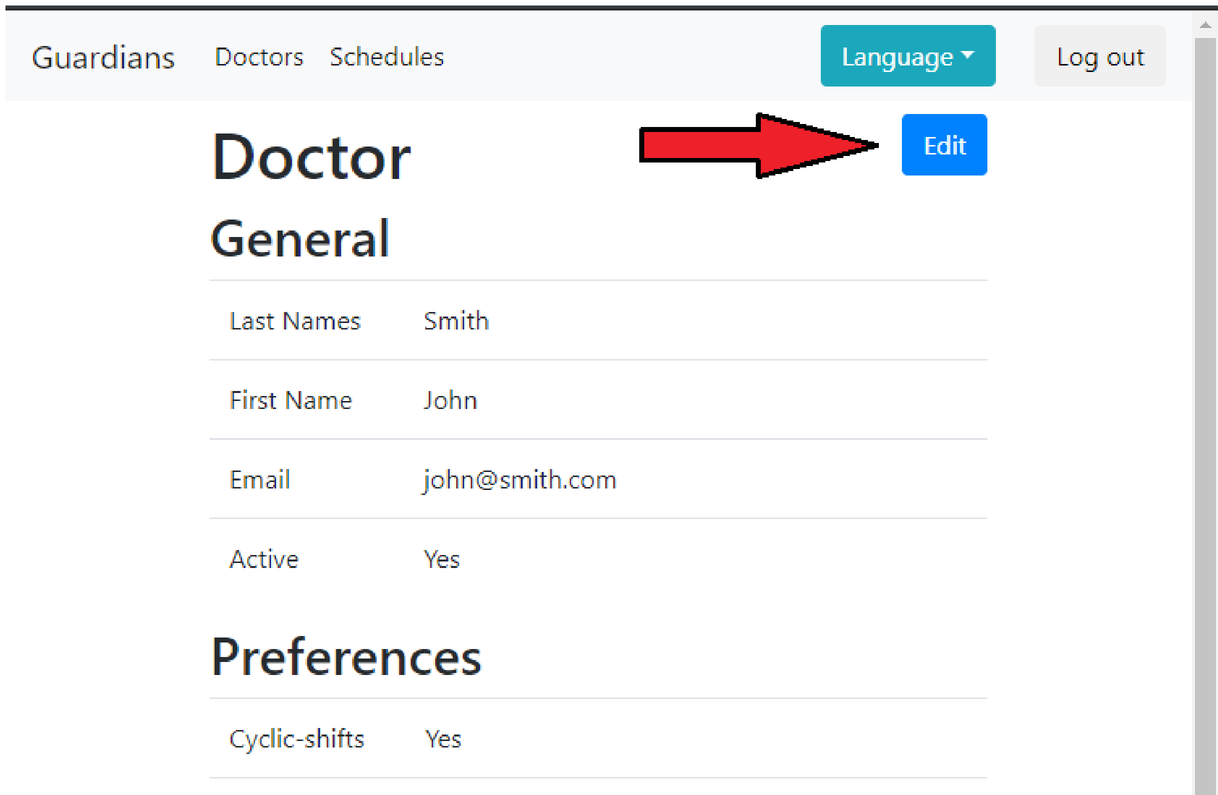
1. Click on 'Doctors' from the homepage:



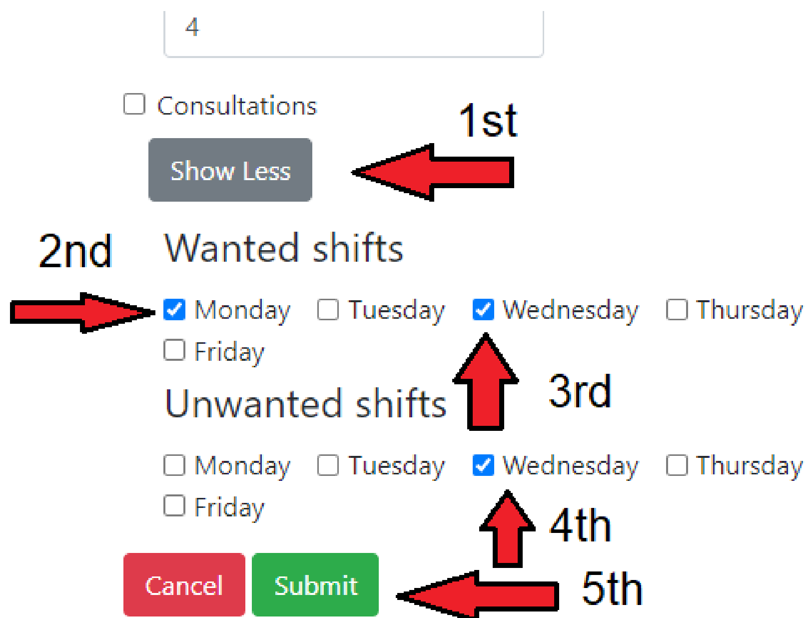
2. Search for the doctor to be edited and click on it:



3. Click on 'Edit':

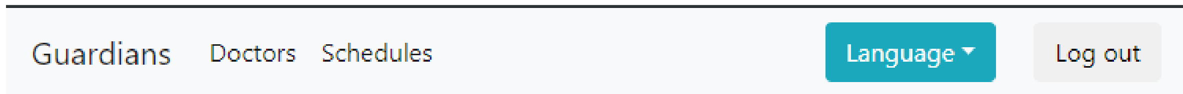


4. Edit the desired fields and submit the changes (E.g. let's add some wanted and unwanted shifts):

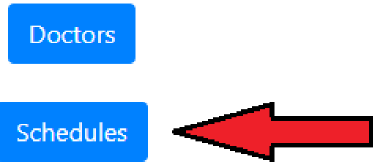


3. Generate a new schedule

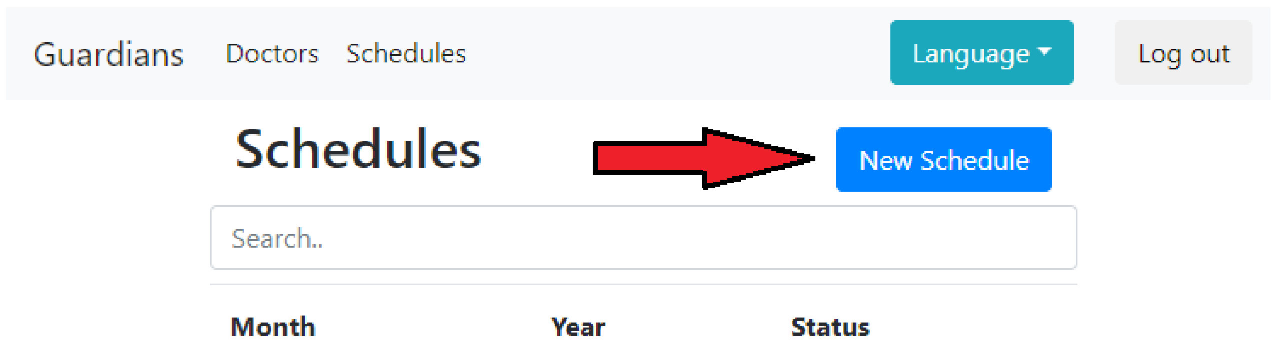
1. From the home page, click on 'Schedules':



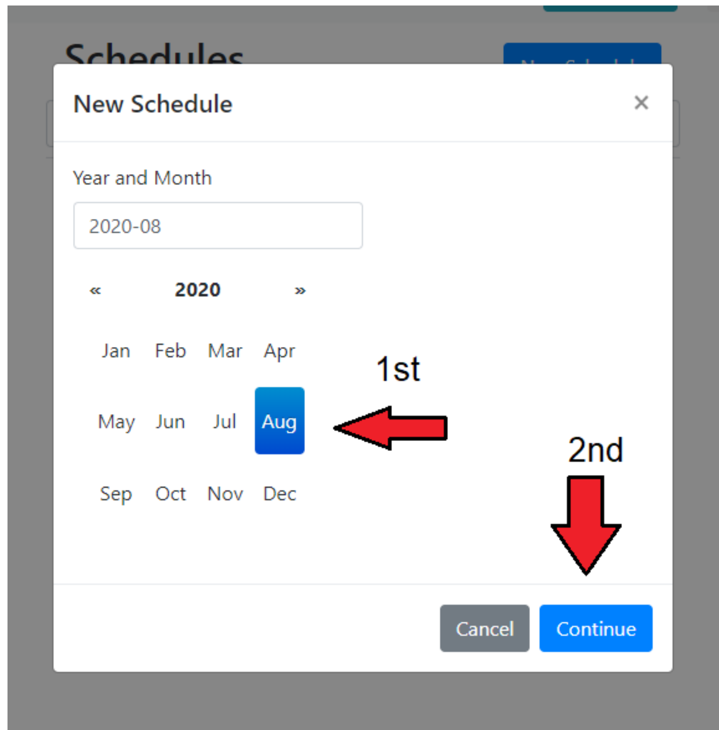
Welcome!



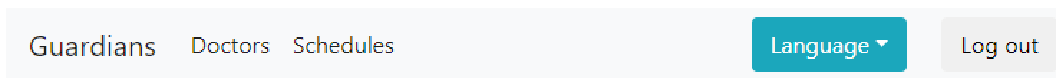
2. Click on 'New Schedule':



3. Select the month whose schedule is to be generated (for example, August 2020) and click on 'Continue':



4. The following screen will appear:



New Schedule

Select non working days with a click

More options on right-click/hold

5. We can set a day to be a non-working-day by clicking on it (E.g. 14th of August):

Select non working days with a click

More options on right-click/hold

August 2020

Cancel Submit

6. We can change the configuration of a particular day by right clicking on it (E.g. let's say doctor John Smith wants a shift the 17th of August):

    1. Right click on the desired day:

New Schedule

Select non working days with a click

More options on right-click/hold

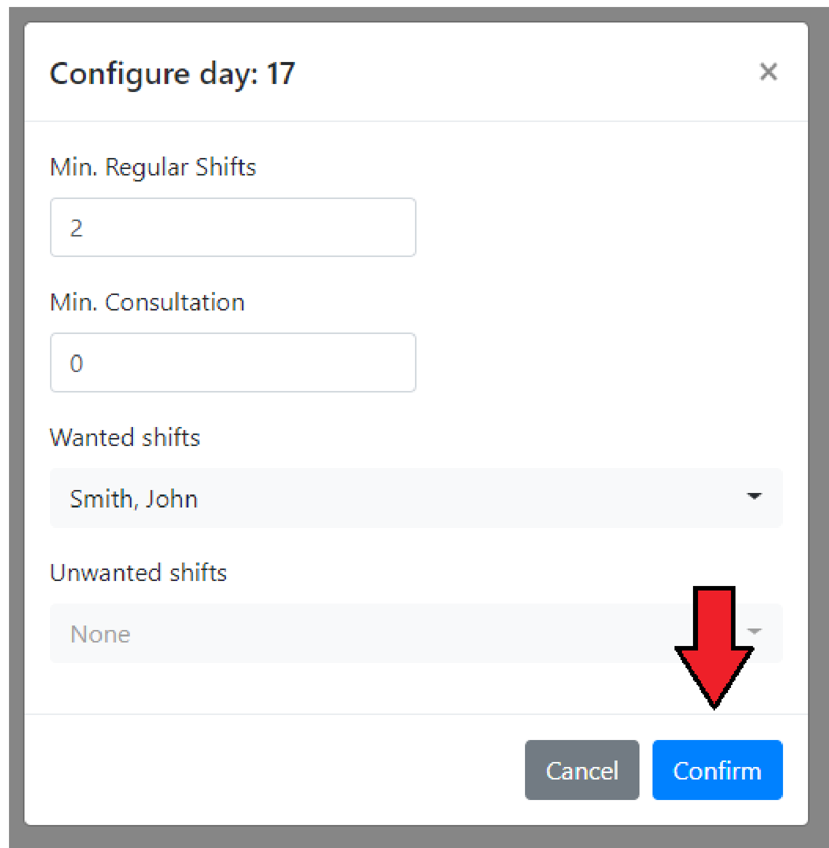
August 2020

Cancel Submit

2. Click on the wanted shifts field:

3. Search for the desired doctor and click on it:

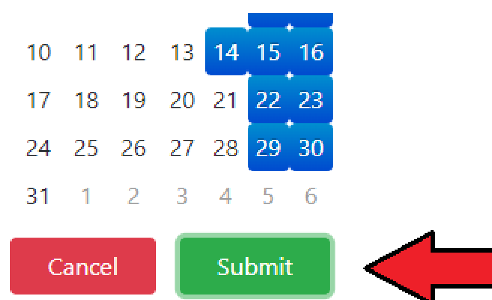
4. Click on confirm:



If no configuration is applied, the default configuration is:

- Min. Regular-shifts: 2
- Min. Consultations: 0
- No wanted shifts
- No unwanted shifts

7. After applying the desired configuration (if any), we can generate the schedule by clicking on 'Submit':



8. The generated schedule will be displayed after generation has finished (if may take a while):

Guardians Doctors Schedules Language Log out

Schedule

Calendar List

2020-08

To be Confirmed

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
27	28	29	30	31	1 Cyclic-shifts 6, 6 5, 5	2 Cyclic-shifts 7, 7 Wayne, John 8, 8
3 Cyclic-shifts 10, 10 9, 9 Shifts 10, 10	4 Cyclic-shifts 12, 12 11, 11 Shifts 9, 9	5 Cyclic-shifts 14, 14 13, 13 Shifts 14, 14	6 Cyclic-shifts 15, 15 Smith, John 16, 16 Shifts	7 Cyclic-shifts 17, 17 18, 18 Shifts 17, 17	8 Cyclic-shifts 19, 19 20, 20	9 Cyclic-shifts 2, 2 1, 1

9. We can change how the schedule is displayed by clicking on the top right buttons:

Guardians Doctors Schedules Language Log out

Schedule

Calendar List

2020-08

To be Confirmed

1
Cyclic-shifts
6, 6
5, 5

2
Cyclic-shifts
7, 7
Wayne, John
8, 8

3
Cyclic-shifts
10, 10
9, 9

Shifts
10, 10
21, 21
9, 9

APPENDIX C – DEPLOYMENT

This section will describe the deployment that will be made. As mentioned in 4Solution designed, the four components of the application (Web server, REST service, Scheduler and DBMS) will be deployed into a single machine.

To make the deployment easier, it has been automated. The scripts and instructions needed to deploy it can be found at the deployment repository. See Appendix A – Code.

Now, we will explain the procedure followed by these scripts. Note the deployment has been tested on Ubuntu 18.04 LTS

First of all, we need to know the different files that will have to be deployed for each of the services:

- REST service: As mentioned in 2.8 Spring, using Maven, we can build the whole Spring application into a single self-contained jar file. This way, for the REST service to be deployed, we only need three different files:
  - The service's jar file
  - The application.properties file, to configure the service.
  - The public-private key, for TLS encryption.
- Web server: As the web server is just another Spring application built with Maven, it only needs three different files:
  - The server's jar file, configure the server.
  - The application.properties file, to configure the server
  - The public-private key, for TLS encryption.
- Scheduler: The scheduler is divided in two different directories:
  - Configuration directory: Contains the scheduler configuration file mentioned at 4.5.3 Scheduler's design, and a file to configure logging.

  - Source directory: Contains the main script and the script responsible for scheduling the shifts.

Now, we need to know the project's dependencies:

- Python 3.7
- OpenJDK-1.8
- MySQL server

For the automated deployment, we also need:

- Git
- MySQL command line client

Now, deployment consists basically on four steps:

1. Modifying the application's configuration files
2. Moving the different files to their corresponding locations
3. Configuring the database
4. Configuring permissions on the application's files

These four steps will be explained in the following sections.

1. Modifying the application's configuration files

First of all, we have to clone the desired version of each of the services (Web server, REST service and Scheduler). Their repositories can be found at Appendix A – Code.

Afterwards, we need to generate four different passwords. One for the REST service to authenticate to the database, another one for the web server to authenticate to the REST service, and two more for the web server and the REST service to decrypt their private-public key (as they will be stored in a PKCS12 keystore). This is the responsibility of the `generatePassword.py` script. It takes a single argument (the desired password length), and prints the generated password to its standard output stream.

Then, we can generate the self-signed certificates used by the REST service and the Web server using `keytool`, a command line tool included in the JRE to generate certificates. This is the responsibility of the `generatePkcs12Key.sh` script. It takes one positional argument and two optional arguments. The positional argument is the password used to encrypt the generated key. The optional arguments are:

- `--alias=<value>`: Where `value` is the alias of the generated key. It defaults to *guardians*.
- `--file=<value>`: Where `value` is the path where the generated PKCS12 file will be located. It defaults to *./guardians.p12*.

Lastly, we have to insert the generated passwords into their respective application.properties file. For example, the password used by the Web server to authenticate to the REST service will the value of the property `auth.rest.password`.

To make this process simpler, we have changed the values of these properties to a known token. For example:

```
auth.rest.password = REST_SERVICE_PASSWORD
```

Then, using the `replace.py` script, we can easily look for these tokens and substitute them by their corresponding values. For example, if the generated password was *abcdefghijk*, we would call the script as:

```
python3 replace.py path/to/application.properties \
  REST_SERVICE_PASSWORD=abcdefghijk \
  SOME_OTHER_TOKEN=someValue \
  ...
```

Note that some other values, besides the passwords, will also have to be updated on the configuration files. However, we will not discuss them in this section to keep it brief. Refer to the mentioned repository to find all of these values. Still, to give an example, the ExecStart option of the Systemd service files have to call the correct version of the jar file and at the desired installation path. E.g from the REST service *.service* file:

```
ExecStart=/usr/lib/jvm/java-8-openjdk-amd64/bin/java -jar \
    PATH_TO_GUARDIANS_JAR
```

Where `PATH_TO_GUARDIANS_JAR` has to be the path to the REST service's jar file. E.g.: `/usr/bin/guardians/guardians-v1.0.0.jar`.

2. Moving the different files to their corresponding locations

As moving a file from a location to a different one can be as simple a using the `mv` command, this section will show the default locations of each of the application's files (note they can be changed on the `config.sh` file):

```
/etc/
 |- guardians/ # The REST service's configuration directory
 |    |- application.properties
 |    |- keystore/
 |    |    |- guardiansRest.p12
 |    |- scheduler/ # The Scheduler's configuration directory
 |    |    |- scheduler.json
 |    |    |- logging.json
 |- guardiansWebapp/ # The Web server's configuration directory
 |    |- application.properties
 |    |- keystore/
 |    |    |- guardiansWebapp.p12
 |- systemd/system/
 |    |- guardians.service        # Systemd service files for both
 |    |- guardiansWebapp.service  # the REST service and Web server
/var/log/
 |- guardians/ # The REST service's and scheduler's logging dir
 |    |- guardians.log
 |    |- scheduler.log
 |- guardiansWebapp/ # The Web server's logging directory
 |    |- guardiansWebapp.log
/usr/lib/
 |- guardians/
```

```
|   |- guardians-vXXX.jar # The REST service's jar file
|   |- scheduler/ # The scheduler's source directory
|   |    |- main.py
|   |    |- scheduler.py
|- guardiansWebapp/
|   |- guardiansWebapp-vXXX.jar # The Web server's jar file
```

3. Configuring the database

This section will show the SQL statement that will be used to configure and populate the database with initial data. Note all of these statements can be found at the 'sql' directory in the release branch of the REST service repository.

First of all, we have to create database used by the REST service (the following commands should be run on a `mysql` prompt):

```
> CREATE DATABASE db_guardians;
```

Then, we have to create the application's user:

```
> CREATE USER 'guardiansUser'@'%'
        IDENTIFIED BY 'GENERATED_PASSWORD';
```

Now, we grant only the required privileges on this user:

```
> GRANT SELECT, INSERT, DELETE, UPDATE ON db_guardians.*
  TO 'guardiansUser';
```

Lastly, we have to create the tables needed by our application. For example, the doctor table will be created a

```
> CREATE TABLE doctor
   (
     id          BIGINT NOT NULL,
     email       VARCHAR(255) NOT NULL,
     first_name  VARCHAR(255) NOT NULL,
     last_names  VARCHAR(255) NOT NULL,
     start_date  date NOT NULL,
     status      INTEGER NOT NULL,
     PRIMARY KEY (id)
   )
  engine=InnoDB;
```

Note the create statements have been automatically generated with JPA. To do this, we just need to add the following into the application.properties (and then, execute the build and execute the application):

```
  spring.jpa.properties.javax.persistence.schema-generation \
    .scripts.action = create # Only genereate the 'create'
statements
  spring.jpa.properties.javax.persistence.schema-generation \
    .scripts.create-target = create.sql # Name of the generated file
  spring.jpa.properties.javax.persistence.schema-generation \
    .scripts.create-source = metadata # Generate SQL from @Entity
```

Then, we will also preload the allowed shifts. For example:

```
> INSERT INTO allowed_shift VALUES (1, 'Monday');
```

Lastly, the `configure.sh` has an option (`PRELOAD_DOCTORS_AND_SHIFT_CONFIGS=1`) that allows choosing whether initial doctors and their shift configurations should be preloaded. This initial data has been provided by the current shift manager at HUVM. For example:

```
> INSERT INTO doctor
           (id, first_name, last_names, email, start_date, status)
  VALUES   (1, '1', '1', '1@guardians.com', '2020-05-01', 0);

> INSERT INTO shift_configuration
           (doctor_id, min_shifts, max_shifts, num_consultations,
            does_cycle_shifts, has_shifts_only_when_cycle_shifts)
  VALUES   (1, 0, 0, 0, TRUE, FALSE);
```

4. Configuring permissions on the application's files

This section will explain the permissions that will be granted on the application's files. Note that we will also create two new users in the system *guardiansUser* and *guardiansWebappUser* that will own their respective files.

The configuration directories will just need permissions to read and list files. For example:

```
chmod 550 /etc/guardians
```

Then, the configuration files will just need read permissions:

```
chmod 440 /etc/guardians/application.properties
```

The log directories will need permissions to read, list (both for the log rotation) and create files (to create the logs). For example:

```
chmod 770 /var/log/guardians
```

The directories containing the jar files and the python scripts need permissions to read, list and create (the jars will generate files like `.classpath`, and the python interpreter will generate the `__pycache__` directory). For example:

```
chmod 770 /usr/lib/guardians
```

Lastly, the python scripts and the jar files will only need read privileges. For example:

```
chmod 440 /usr/lib/guardians/scheduler/main.py
```