md.md

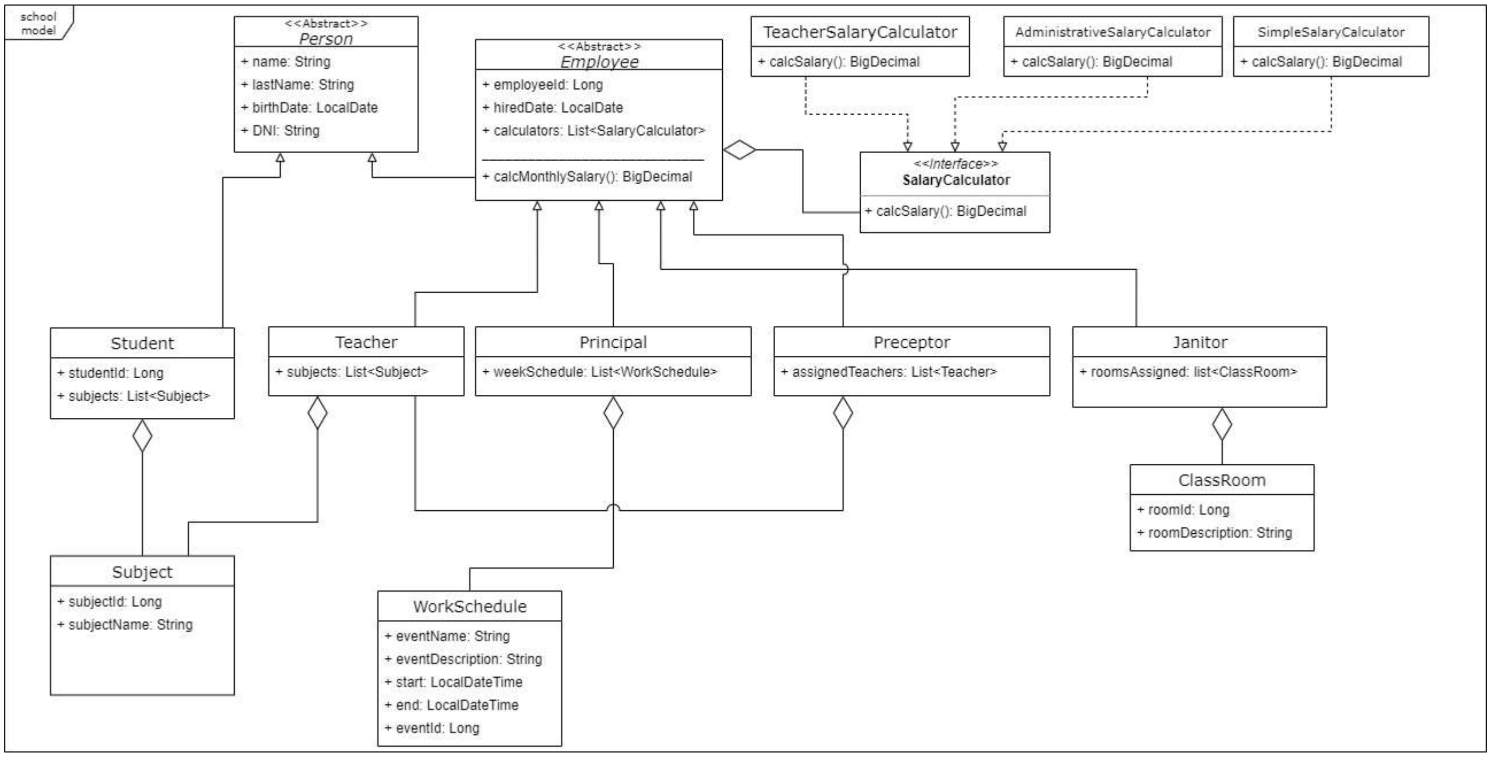
Developer Hiring Exam

Joaquin Rodriguez

Excercise 1

A) "Create the diagram that describes the domain model and relationships between entities."

Diagram file on "school-diagram-domain-model.png"


school model
<<Abstract>> Person
+ name: String
+ lastName: String
+ birthDate: LocalDate
+ DNI: String
<<Abstract>> Employee
+ employeeId: Long
+ hiredDate: LocalDate
+ calculators: List<SalaryCalculator>
+ calcMonthlySalary(): BigDecimal
TeacherSalaryCalculator
+ calcSalary(): BigDecimal
AdministrativeSalaryCalculator
+ calcSalary(): BigDecimal
SimpleSalaryCalculator
+ calcSalary(): BigDecimal
<<Interface>> SalaryCalculator
+ calcSalary(): BigDecimal
Student
+ studentId: Long
+ subjects: List<Subject>
Teacher
+ subjects: List<Subject>
Principal
+ weekSchedule: List<WorkSchedule>
Preceptor
+ assignedTeachers: List<Teacher>
Janitor
+ roomsAssigned: list<ClassRoom>
ClassRoom
+ roomId: Long
+ roomDescription: String
Subject
+ subjectId: Long
+ subjectName: String
WorkSchedule
+ eventName: String
+ eventDescription: String
+ start: LocalDateTime
+ end: LocalDateTime
+ eventId: Long

B) "Returns all the students grouped by the first letter of last name"

Method on file "StudentService line 21"

```
public Map<Character, List<Student>>getStudentsGrupedByLastName(){
        return this.students
        .findAll()
        .stream()
        .collect(Collectors.groupingBy(s -> s.getLastName().charAt(0), Collectors.toList()));
}
```

C) "Return all the students taking specific subject"

Method on file "StudentService line 28"

```
public List<Student> getStudentsInSubject(Long subjectId) {
        return this.students
        .findSubjectById(subjectId)
        .map( subj ->
                this.students
                .findAll()
                .stream()
                .filter( s -> s.getSubjects().contains(subj))
                .collect(Collectors.toList())
        ).orElse(new ArrayList<Student>());
}
```

Duplicates are checked at saving by method “addSubjectToStudent” on file “StudentService”

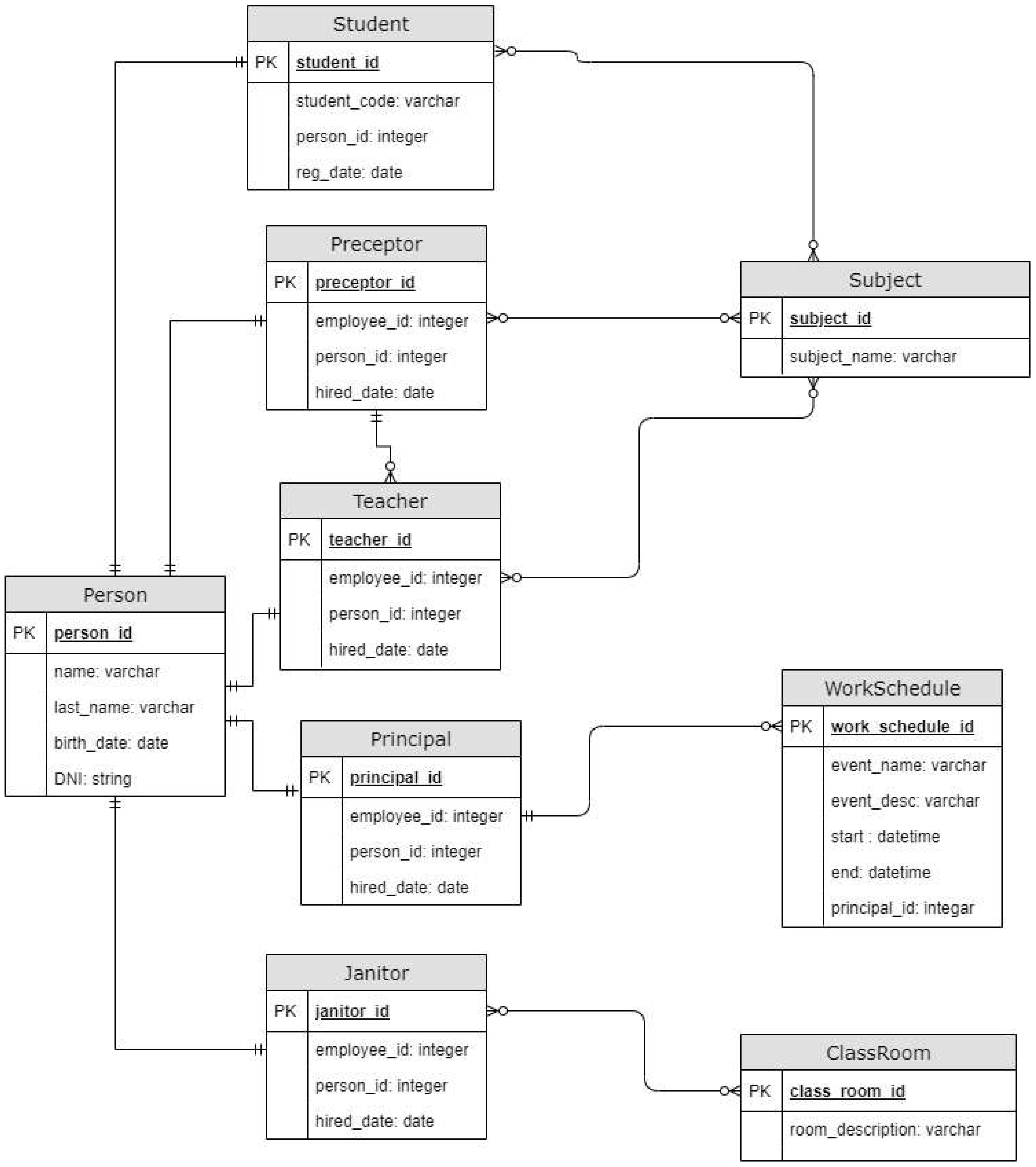
```
public void addSubjectToStudent(Long subjectId, Long studentId) {
        //Ideally, this should be a restriction in the database.
        Optional<Student> student = this.students.findById(studentId);

        student.map(s -> s.getSubjects()
                        .stream()
                        .filter(sub -> sub.getId().equals(subjectId))
                        .findFirst().get())
        .ifPresent(s2 -> {
                throw new UserAlreadyWithSubjectException(s2.getName() + "-" + subjectId);
        });
        this.students.addSubjectToStudent(student.get(), subjectId);
}
```

D) “Create two different database table structure diagrams”

First diagram:

Diagram file on “db-model-1.png”



Pros:

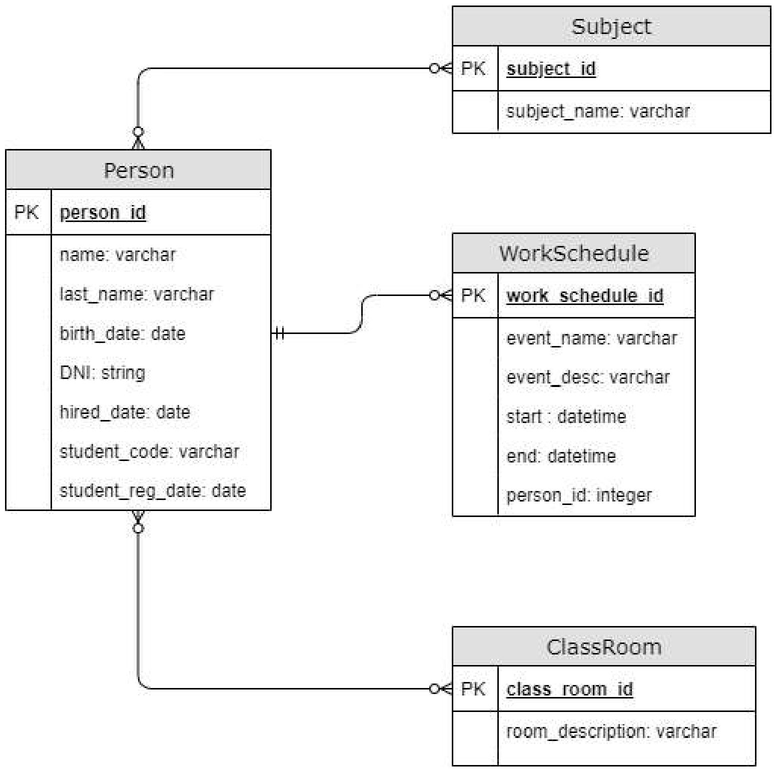
- Maintains the same structure than the class diagram, modifications are easier to map to the code
- Easier to understand and validate relations and restrictions
- In the future, modifications that involves differences between entities are easier to implement

Cons:

- Joins with multiple tables will degrade performance
- Insertions of rows involves multiples tables

Second diagram:

Diagram file on “db-model-2.png”



Pros:

- Query simplification: less joins means better performance and easier to write queries
- Inserting a new row involves less tables

Cons:

- It’s difficult to maintain restrictions and data consistency
- Extra steps may be needed in order to map data structure to project entity classes

E)

The first change you can do to the query is remove the first join since the data needed is on the “person” table, then you can limit the fields fetched to first_name and last_name instead of all fields (*). If not all rows are needed at once, it’s always a good idea to do pagination using limit and offset clauses to fetch less rows. Also you can add an index to the field used in the ‘where’ clause for better performance.

F)

A solution for this scenario is the creation of a materialized view from that query to gain performance since it's not updated frequently, this is a common solution in scenarios like historic summaries or reports where info it’s not only updated with low frequency but it’s also not very important to have the latest information. Other solutions mentioned on exercise E are also valid (indexes since tables are not updated frequently, or paginated queries to return partial information). Also, a non-sql related solution might be the usage of an external cache service (memcache, redis, etc.)

G)

To calculate the age of student it can be useful to use an already created function in the sql engine, for example:

```
SELECT *
FROM student s
WHERE EXTRACT ( year FROM age(current_date , s.date_of_birth) ) BETWEEN 19 AND 21;
```

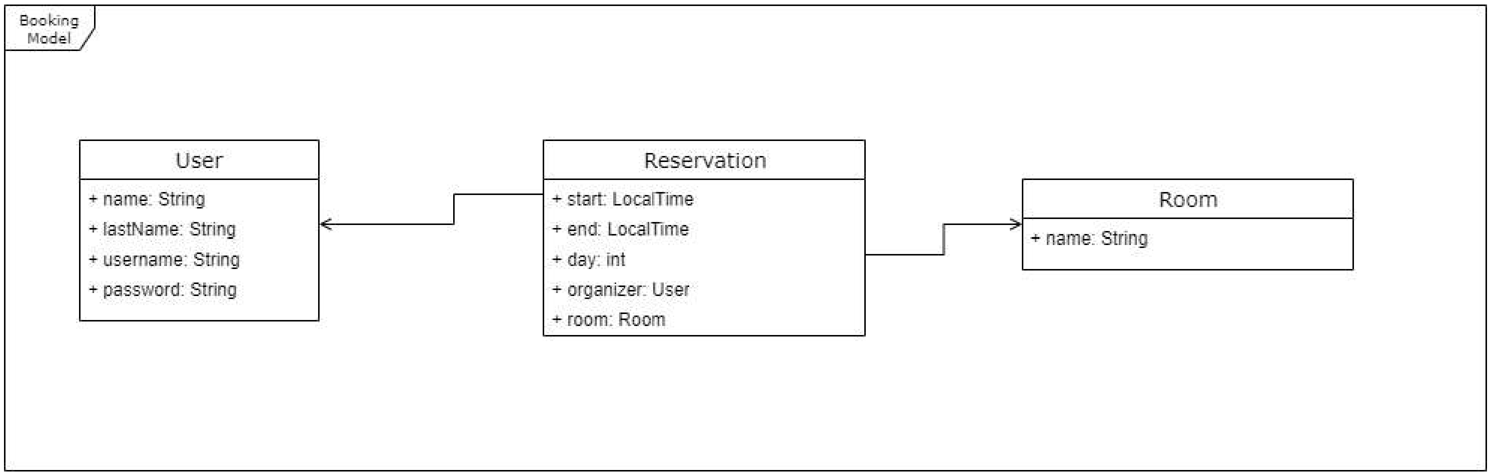
One way to optimize the query would be to add an index on the date_of_birth field and calculate the max and min dates of birth in the moment of performing the query to avoid performing extra calculations on each student row.

H) Business logic in the database is achieved with the use of stored procedures, triggers and integrity restrictions with the main advantage of gaining performance ,centralizing the core business logic to avoid replication/maintenance in multiple applications and provide security with sensible operations, at the cost of added complexity when you have requirements that can be difficult to solve with sql/plsql and in some case maintenance, testing or debugging can be frustrating.

Excercise 2

A)

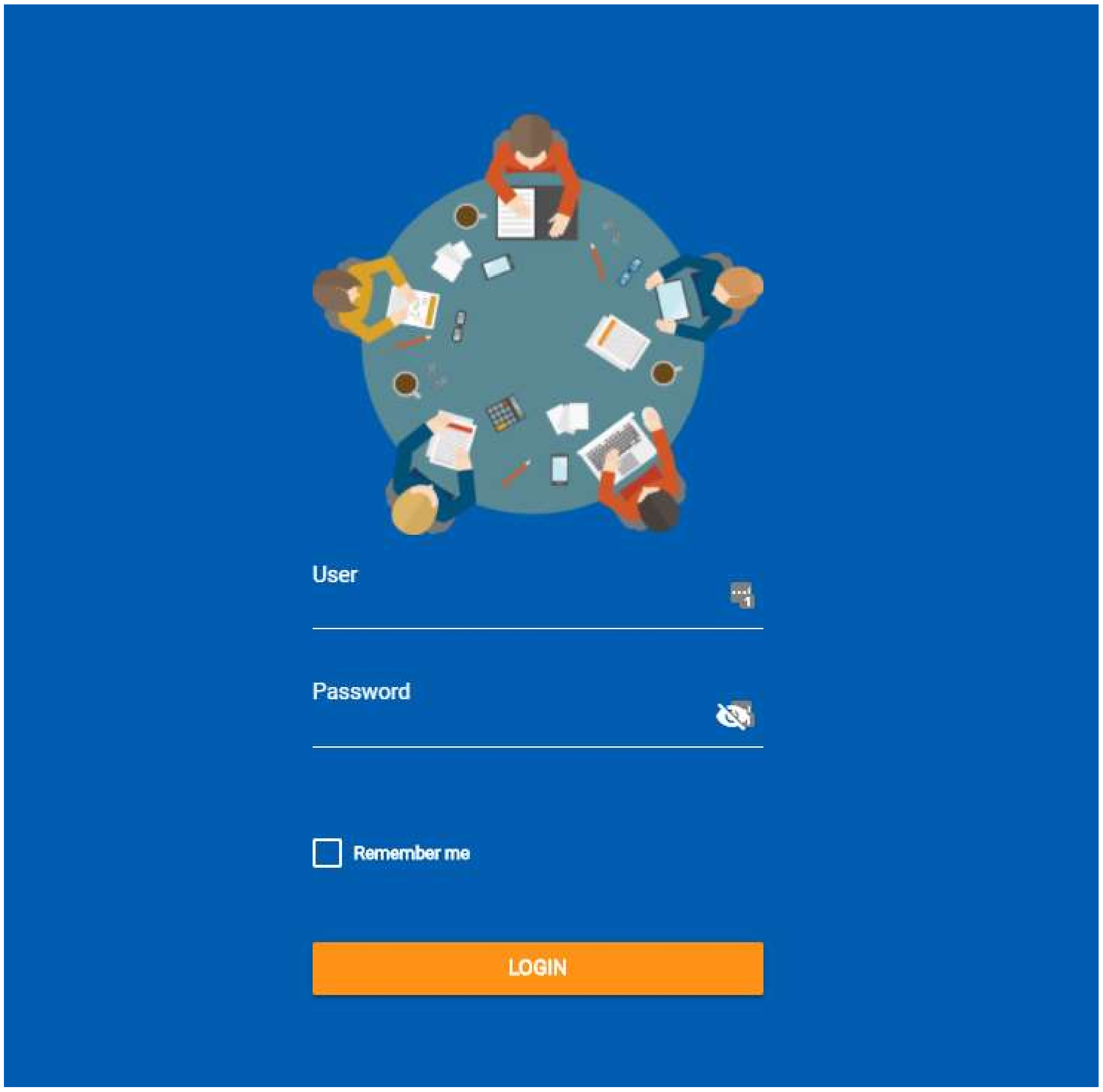
Diagram file on “booking-domain-model.png”



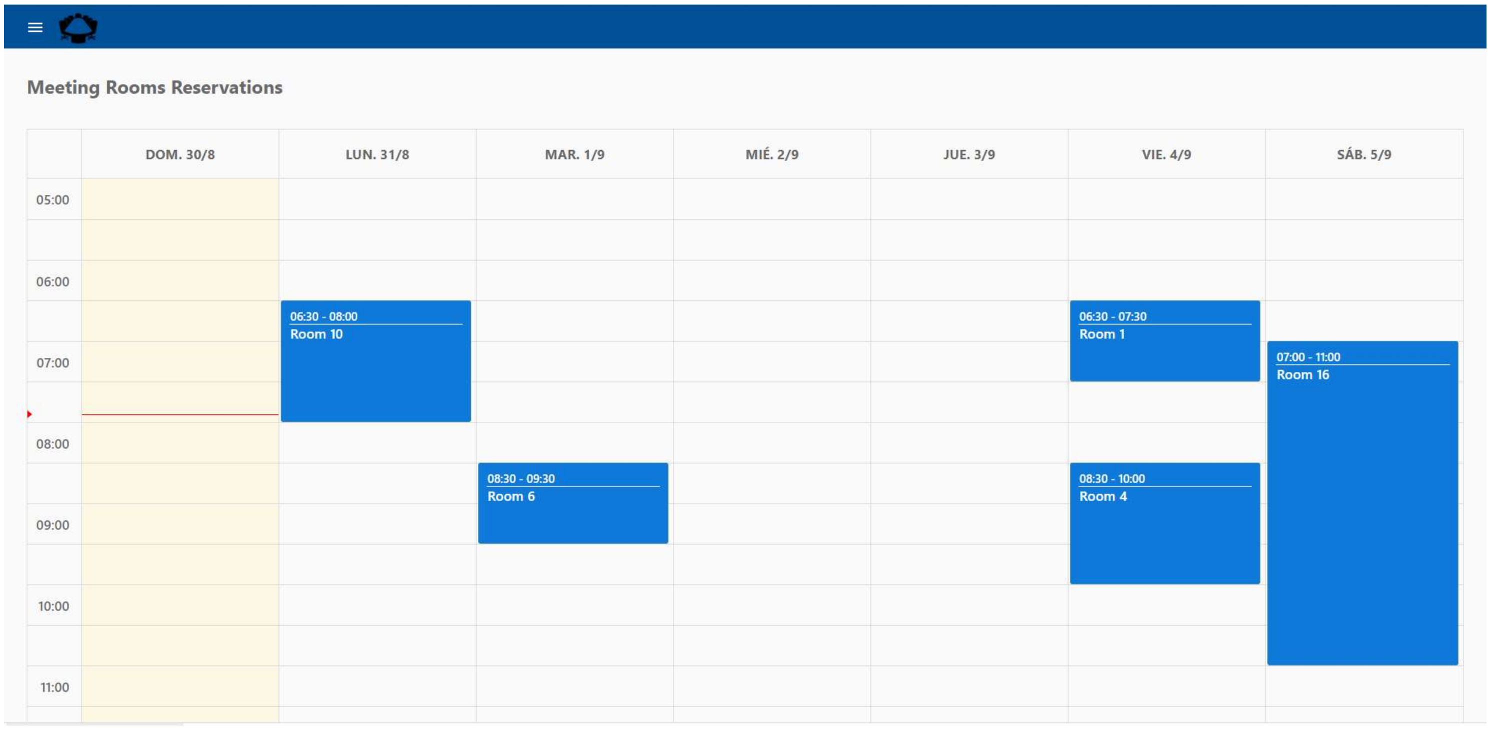
B)

User interface for booking application:

Login screen

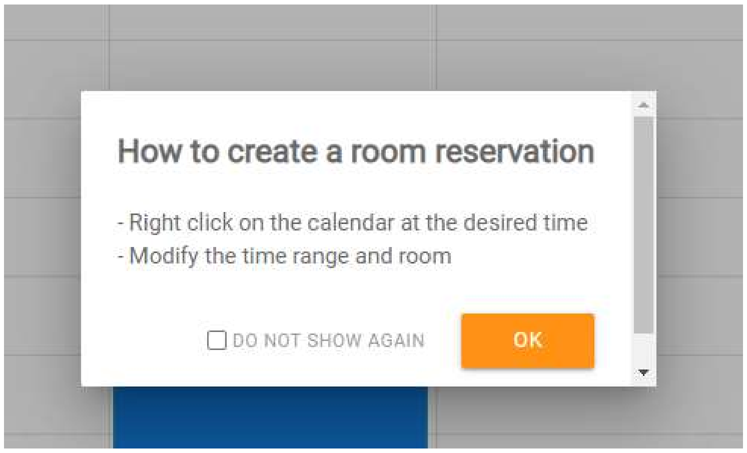


Home view



	DOM. 30/8	LUN. 31/8	MAR. 1/9	MIÉ. 2/9	JUE. 3/9	VIE. 4/9	SÁB. 5/9
05:00							
06:00		06:30 - 08:00 Room 10				06:30 - 07:30 Room 1	
07:00							07:00 - 11:00 Room 16
08:00			08:30 - 09:30 Room 6			08:30 - 10:00 Room 4	
09:00							
10:00							
11:00							

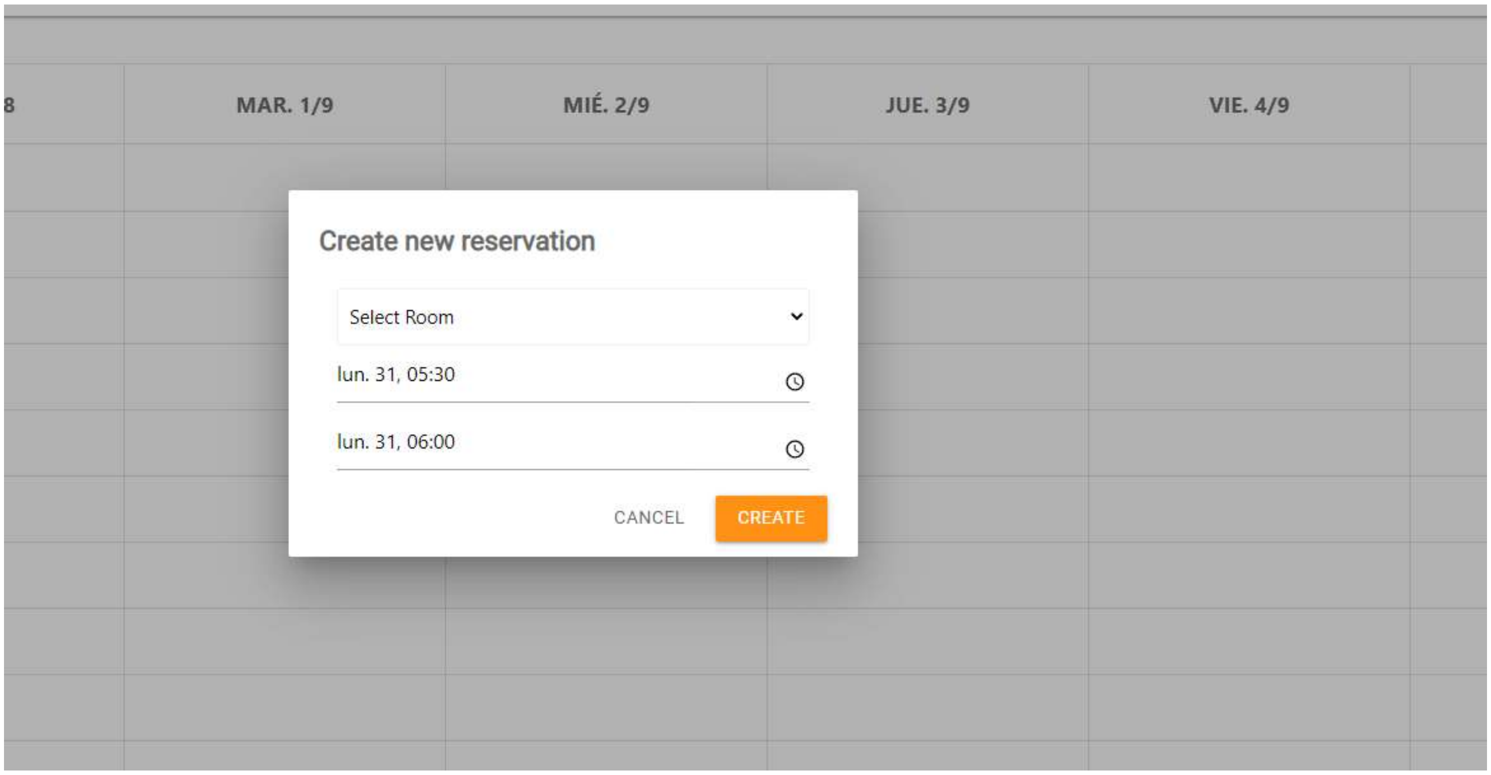
Welcome popup with indications



How to create a room reservation

- Right click on the calendar at the desired time
- Modify the time range and room

Creating new reservation



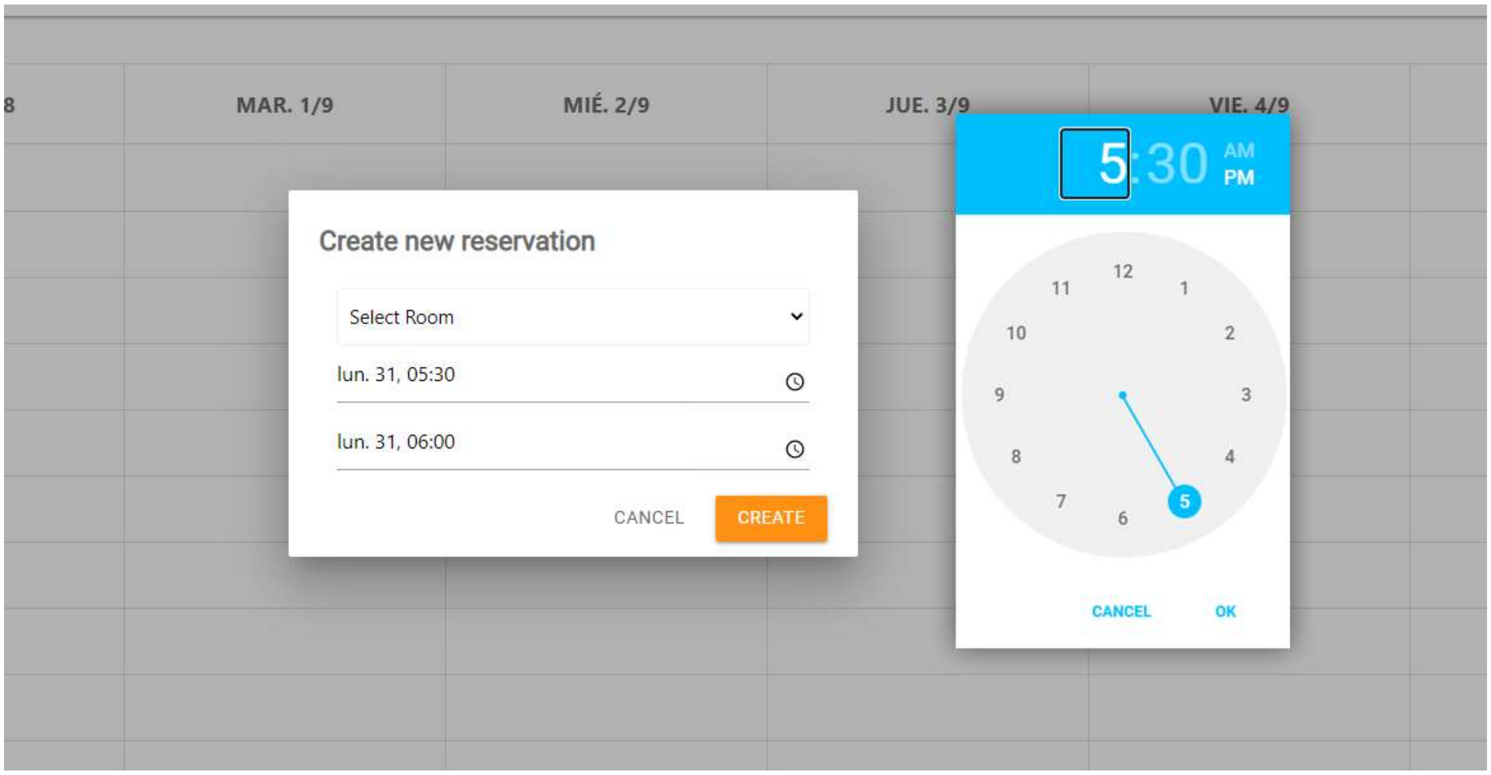
Create new reservation

Select Room

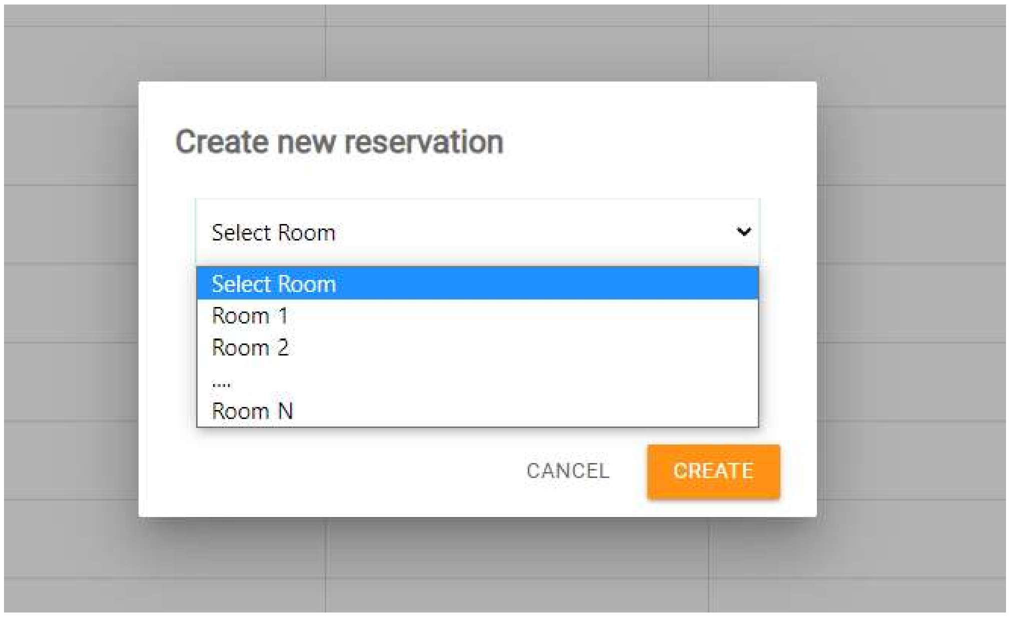
lun. 31, 05:30

lun. 31, 06:00

Selecting time



Selecting room



C)

```
public void verifyReservationBeforeSave(LocalTime start,LocalTime end,Long roomId) {
        //Time restriction check

        //Not less than 15 min
        //Not more than 180 min (3hs)
        long minutes = ChronoUnit.MINUTES.between(start,end);

        if(minutes < MIN_ALLOWED || minutes > MAX_ALLOWED)
                throw new InvalidDurationException();

        //check reservations that overlaps in time for the same room
        //query the database to get all reservations in a room
        reservations.findReservationsByRoomId(roomId)
        .stream()
        .filter(r -> {
                return r.getStart().isBefore(end) && r.getEnd().isAfter(start);
                //this can be avoided by adding the time conditions directly to the first query
                //and just checking if a matching record exists on database
        })
        .findFirst()
        .ifPresent(r -> {
                throw new OverlapException(" reservation id:"+r.getId()+" for room id:"+roomId);
        });
}
```