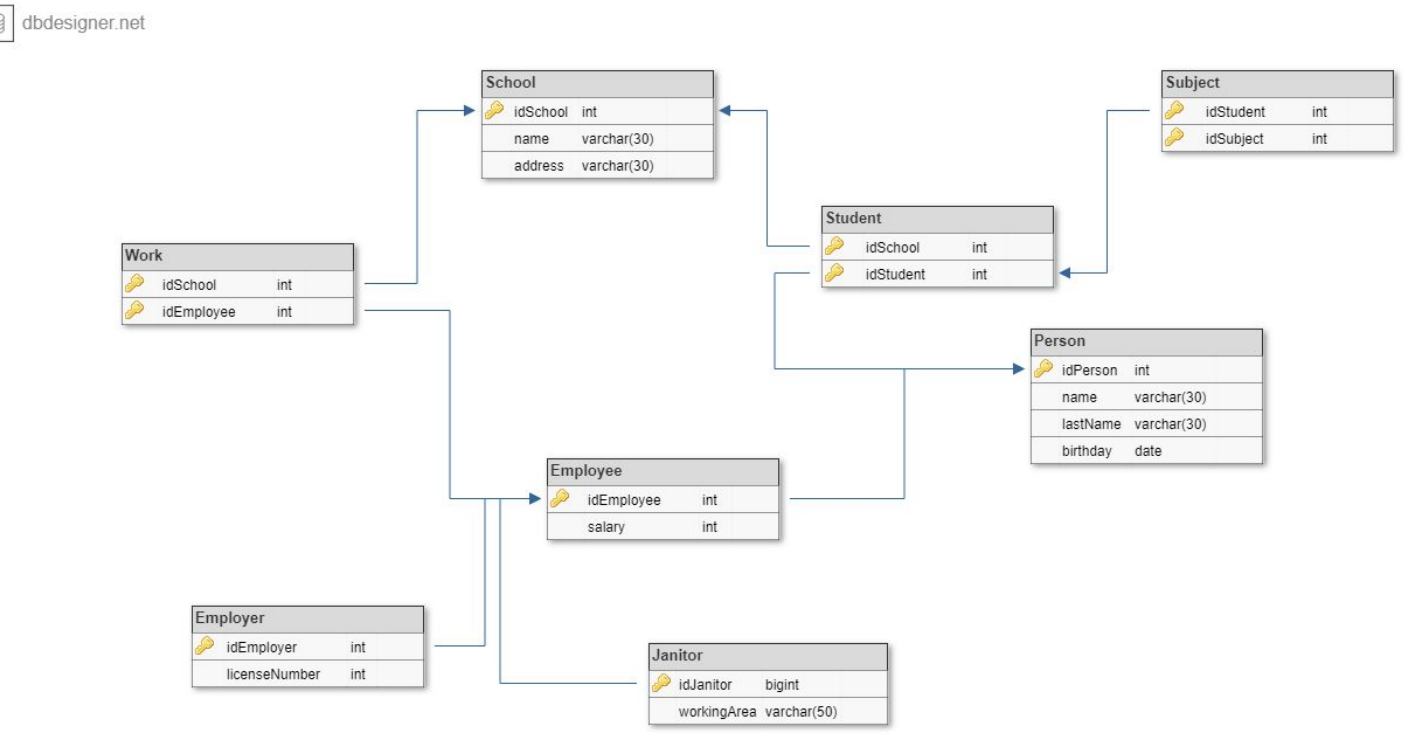
Ejercicio 1

Punto A:
Adjunto imagen (Diagrama-Ejercicio-1.png) del diagrama de clases UML del modelo implementado para el ejercicio uno.

Punto B:
En la carpeta JavaTeracode está el ejercicio implementado en java.

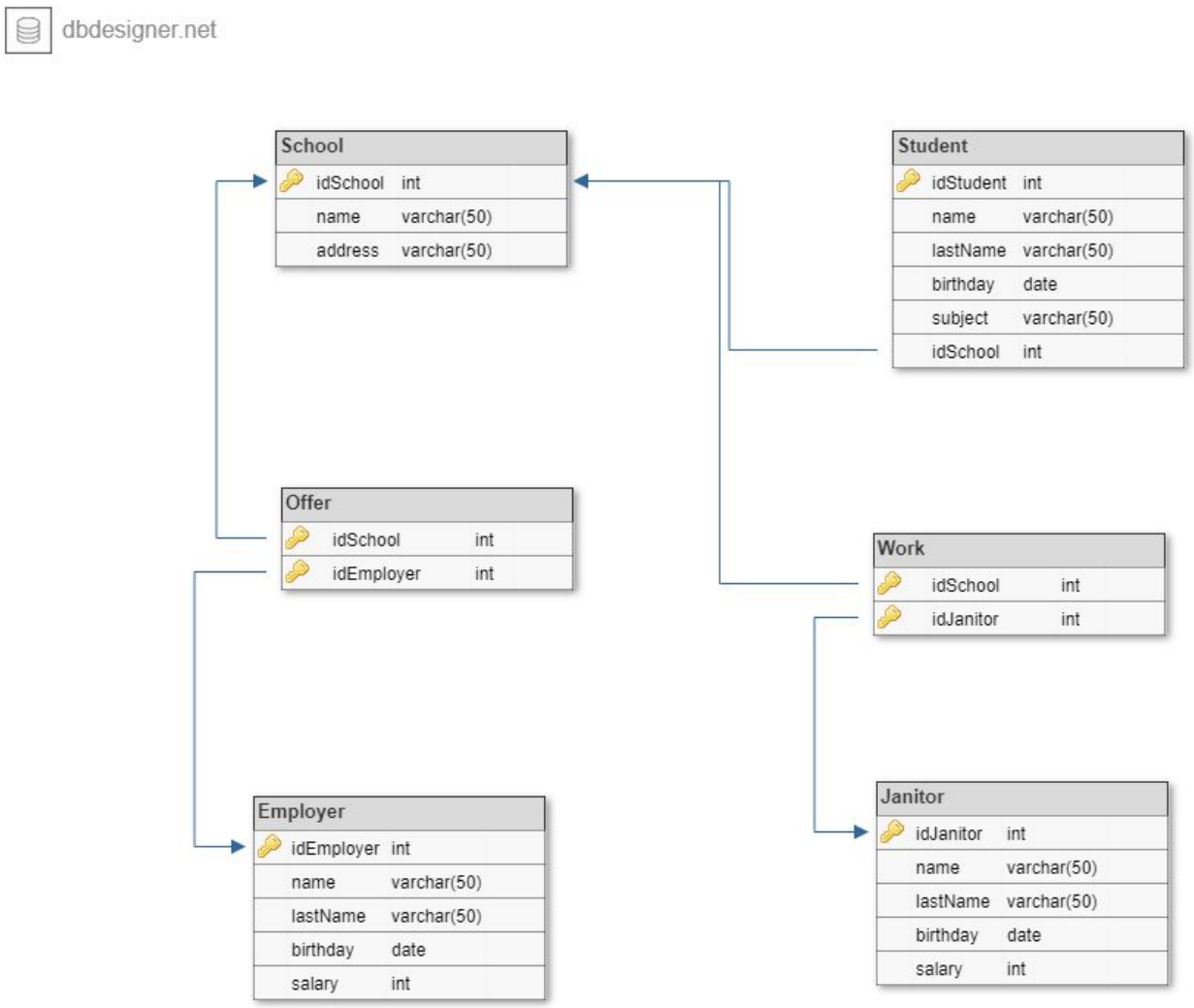
Punto C:
En la carpeta JavaTeracode está el ejercicio implementado en java. Se puede ejecutar la clase main para ver la demostración de ambos puntos.

Punto D:
En el **diagrama 1**, se muestra el esquema de tablas para la base de datos del ejercicio 1. Este esquema tiene como punto fuerte la correcta distribución de los datos, donde las tablas están normalizadas para evitar la redundancia de datos. Al no tener redundancia de datos, se pueden mantener los datos actualizados fácilmente. Cada tabla cuenta con un índice primario de búsqueda. El punto débil es que hay que hacer joins entre las tablas para obtener los datos deseados. Mientras más compleja es la consulta, más habrá que optimizar la consulta para que sean eficientes.



(diagrama 1)

En el **Diagrama 2**, vemos una base de datos mal normalizada, donde hay redundancia de datos para las tablas Student, Employer (Principal) y Janitor. Al tener redundancia de datos, el tamaño de la base de datos es mucho mayor. Otra cosa que tiene como contra es que al momento de actualizar los datos, es una tarea costosa dado que hay que actualizar todas las tuplas que toquen ese dato.



(diagrama 2)

Punto E: Esta consulta se puede optimizar de varias maneras. Una es armando un índice por los campos que se debe retornar, para encontrarlos más rápido. Aun así, se podría optimizar más, tratando de no hacer tantos joins. En el caso original, los tres joins juntos no son necesarios dado que el nombre y apellido del Janitor se encuentran en la tabla persona. Por esto, la tabla que se debe recorrer es persona y son innecesarios a ese punto los joins con EMPLOYEE y JANITOR. Por esto mismo, podríamos enviar el join restante a una subconsulta. De esta forma no hacemos tantas uniones de tablas al principio, por lo que la cantidad de datos a utilizar es menor. SQL evalúa primero el FROM, luego el WHERE y por último el SELECT. Dentro del WHERE, hacemos una subconsulta, buscando los JANITOR que tengan como workingArea la palabra Hallway. Por último, el SELECT * trae todos los campos de la/las tabla/s en cuestión. Por lo que en la consulta del ejercicio, se estaban retornando todos los datos de los joins. Si hacemos el SELECT solo con los datos que se necesitan, también ahorramos mucho tiempo en la respuesta de la consulta.

```
SELECT P.name, P.lastName
FROM PERSON P
WHERE EXISTS (
                        SELECT 1
                        FROM EMPLOYEE E INNER JOIN JANITOR J ON E.id = J.id
                        WHERE J.workingArea = "Hallway"
                          AND P.id = J.id
                )
```

Punto F:

Primero intentaría reestructurar la consulta de forma eficiente, tratando de acceder a la menor cantidad de joins posibles. Utilizando las herramientas que nos provee el explain de SQL para ver cómo se comportan las consultas a bajo nivel.

Luego, crearía un índice por los campos de búsqueda, dado que tenemos poca tasa de actualización. De esta forma, accederemos más rápido a los datos solicitados. Al tener poca tasa de actualización, el índice se tendrá que reorganizar muy pocas veces.

En caso de que los valores se puedan dar por rangos, se podría utilizar un índice de árbol a+, dado que son muy eficientes para devolver datos por rangos, y en caso de que la consulta retorne solo un valor se podría utilizar un índice por hash.

Punto G:

Enviando la tabla STUDENT como subconsulta dentro del where, ya estamos eliminado el join de tablas. A su vez, al estar dentro de un EXISTS, SQL donde encuentra un match TRUE por la condición WHERE de la subconsulta, no sigue buscando matcheos para la row de PERSON procesada.

```
SELECT name,lastName
FROM PERSON P
WHERE EXISTS (
            SELECT 1
            FROM STUDENT S
            WHERE S.id = P.id
           )
  AND DATEDIFF(DAY,P.birthday, GETDATE())/365 BETWEEN 19 AND 24
```

Punto H:

Se podría hacer metiendo varios stores procedures que manejan la lógica de negocio. Se tendrían que agregar triggers y checks que controlen las reglas para agregar/actualizar/borrar registros de las tablas.

Como siguiente paso, se tendría que hacer un relevo de cantidad de datos de cada tabla, para saber que tablas son las más críticas para hacer actualizaciones costosas sobre índices (tablas que tengan muchos insert o updates).

Sobre las que no tengan alta tasa de actualización de datos, se podrían implementar índices para disminuir el tiempo de acceso a los datos.

Esta solución no tiene muchos pro dado que es muy complejo administrar un sistema completamente sobre bases de datos.

A las bases hay que delegarle la responsabilidad de ejecutar transacciones lo más sencillas posibles para no saturar el servidor.

Aunque técnicamente no es recomendable tener el core de la api a nivel de base de datos, existen herramientas o motores muy potentes como por ejemplo SQL Server.

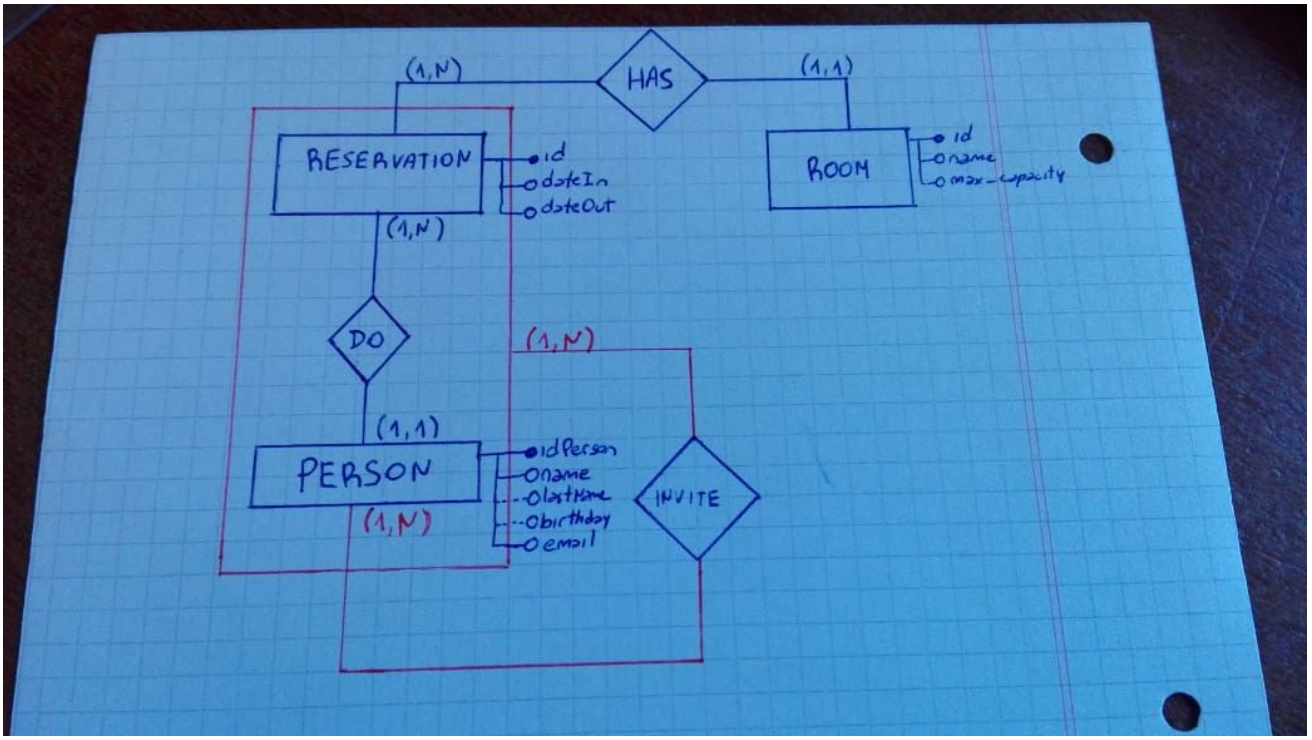
Si es un requerimiento por parte del cliente, utilizaremos el servidor como nexo entre el usuario y el motor, el cual solamente sería el encargado de enviar los datos de un lugar al otro.

Ejercicio 2

Punto A:

Se adjunta una imagen con el Diagrama de Entidades y Relaciones para el ejercicio de Conference Rooms Booking application.

Vale destacar, que hay varias variantes sobre este todo de diseños, pero para simplificar el ejercicio solo se modelo la reserva de un turno.



Punto B:

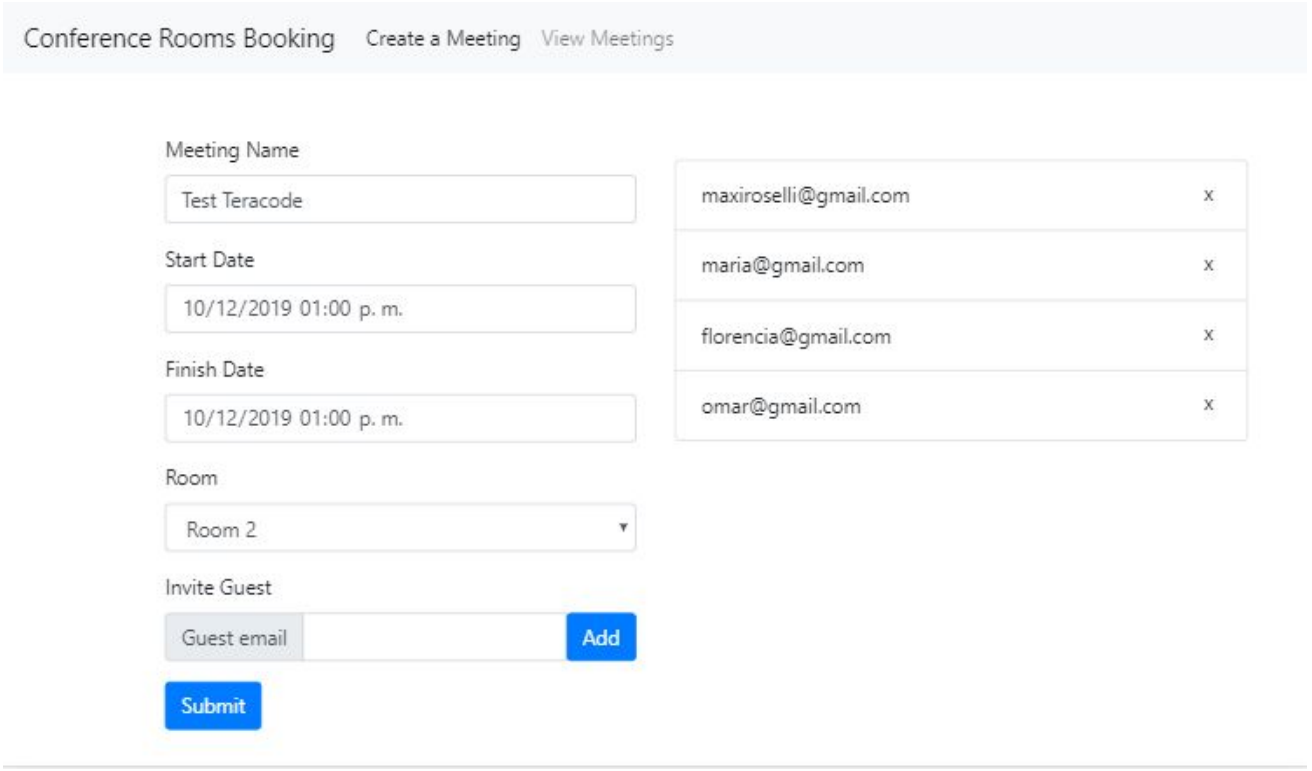
El punto B fue desarrollado con el Framework Angular 2+. Para poder ejecutarlo, se debe tener previamente instalado Node.js, npm y Angular Framework. Las versiones con las que fue desarrollado son las siguientes:

Node.js: v10.15.1
npm: 6.4.1
Angular CLI: 7.3.9

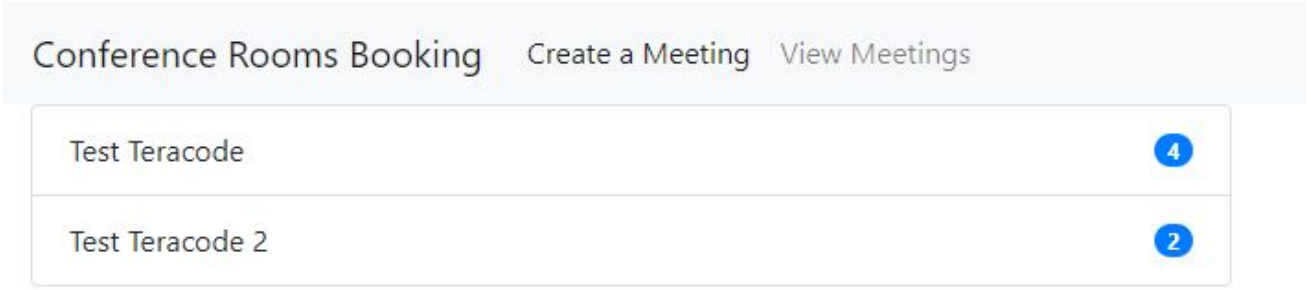
Si ya se tiene previamente instaladas estas aplicaciones, debemos correr sobre la raíz del directorio el comando npm install para que se instalen las dependencias del package.json

Una vez instaladas las dependencias, sobre el proyecto ejecutamos ng serve -o para ejecutar la aplicación. De todas formas, adjunto imagenes:

Create Meeting.



View Meeting



Punto C:

La validación correspondiente, se habria que hacer con un trigger antes de hacer el insert en la base de datos. El trigger es el siguiente.

```
CREATE TRIGGER RulesMeeting
BEFORE INSERT ON RESERVATION FOR EACH ROW
BEGIN
  DECLARE @DIFFMIN INT
  SET @DIFFMIN = DATEDIFF(MINUTE, NEW.DATEIN, NEW.DATEOUT)
  IF @DIFFMIN > 15 AND @DIFFMIN < 180 THEN

          IF NOT EXISTS( SELECT 1 FROM RESERVATION R
                                WHERE IDROOM = NEW.IDROOM
                                 AND (
                                       NEW.DATEIN BETWEEN R.DATEIN AND R.DATEOUT
                                    OR  NEW.DATEOUT BETWEEN R.DATEIN AND R.DATEOUT
                                    )
                           ) THEN
                  INSERT INTO RESERVATION (DATEIN, DATEOUT, IDPERSON, IDROOM)
                       VALUE(NEW.DATEIN, NEW.DATEOUT, NEW.IDPERSON, NEW.IDROOM)
          END IF;
  END IF;
END;
```