

# Boletines de laboratorio

\*\*\*

Introducción a la Ingeniería del Software  
y los Sistemas de Información I

Carlos Arévalo  
Daniel Ayala  
Margarita Cruz  
Fernando Sola  
Inma Hernández  
Alfonso Márquez  
David Ruiz

Curso 2024/25



Escuela Técnica Superior de  
**Ingeniería Informática**



# Índice general

---

<b>1. Creación de tablas I</b>	<b>1</b>
1.1. Objetivo . . . . .	1
1.2. Preparación del entorno . . . . .	1
1.3. Consideraciones de estilo . . . . .	2
1.4. Creación de tabla Degrees (Grados) . . . . .	2
1.5. Creación de tabla Subjects (Asignaturas) . . . . .	3
1.6. Creación de tabla Groups (Grupos) . . . . .	4
1.7. Creación de la tabla Students (Alumnos) . . . . .	5
1.8. Creación de la tabla Grades (Notas) . . . . .	6
1.9. Restricciones de borrado de tablas . . . . .	7
<b>2. Creación de tablas II</b>	<b>9</b>
2.1. Objetivo . . . . .	9
2.2. Preparación del entorno . . . . .	9
2.3. Restricciones en tabla Degrees . . . . .	10
2.4. Restricciones en la tabla Subjects . . . . .	10
2.5. Restricciones en la tabla Groups . . . . .	11
2.6. Restricciones en las tablas Students y GroupsStudents . . . . .	11
2.7. Restricciones en la tabla Grades . . . . .	12
2.8. Comportamientos en borrado para claves ajenas . . . . .	12

<b>3. Instrucciones SQL I</b>	<b>15</b>
3.1. Objetivo . . . . .	15
3.2. Preparación del entorno . . . . .	15
3.3. INSERT . . . . .	16
3.4. UPDATE . . . . .	17
3.5. DELETE . . . . .	17
3.6. SELECT . . . . .	18
3.7. Vistas . . . . .	19
3.8. Consultas varias . . . . .	20
3.9. Ejercicios . . . . .	21
<b>4. Instrucciones SQL II</b>	<b>23</b>
4.1. Objetivo . . . . .	23
4.2. Preparación del entorno . . . . .	23
4.3. ORDER BY, LIMIT, y OFFSET . . . . .	23
4.4. JOIN . . . . .	25
4.5. GROUP BY . . . . .	26
4.6. Consultas varias . . . . .	27
4.7. Ejercicios . . . . .	28
<b>5. APIs REST y Silence</b>	<b>29</b>
5.1. Objetivo . . . . .	29
5.2. Introducción . . . . .	29
5.3. Preparación del entorno . . . . .	30
5.4. Creación de un nuevo proyecto . . . . .	31
5.5. Configuración del proyecto . . . . .	31
5.6. Definición de la BD del proyecto . . . . .	33
5.7. Definición de los endpoints del proyecto . . . . .	34

5.8. Ejecución del proyecto y uso de endpoints . . . . .	36
5.9. Creación, actualización y borrado . . . . .	41
5.10. Definición de endpoints personalizados . . . . .	43
<b>6. Procedimientos, funciones y disparadores</b>	<b>47</b>
6.1. Objetivo . . . . .	47
6.2. Preparación del entorno . . . . .	47
6.3. Procedimientos . . . . .	47
6.4. Funciones . . . . .	49
6.5. Disparadores . . . . .	50
<b>7. Transacciones</b>	<b>53</b>
7.1. Objetivo . . . . .	53
7.2. AUTOCOMMIT . . . . .	53
7.3. Control de transacciones en un procedimiento SQL almacenado . . . . .	54
7.4. Transacciones concurrentes . . . . .	56
<b>A. Entorno de trabajo</b>	<b>57</b>
A.1. Objetivo . . . . .	57
A.2. Instalación de MariaDB y HeidiSQL . . . . .	57
A.3. Creación de una conexión con HeidiSQL . . . . .	58
A.4. Creación de una base de datos . . . . .	60
A.5. Creación de usuarios . . . . .	60
A.6. Conexión con el nuevo usuario . . . . .	61
A.7. Ejecutar script de prueba . . . . .	62
A.8. Instalación y configuración de Python . . . . .	63
A.9. Instalación de Visual Studio Code . . . . .	66
A.10. Instalación de Git . . . . .	68



---

## Laboratorio 1

# Creación de tablas I

---

## 1.1. Objetivo

El objetivo de esta práctica es aprender las instrucciones SQL para crear las tablas de una base de datos. El alumno aprenderá a:

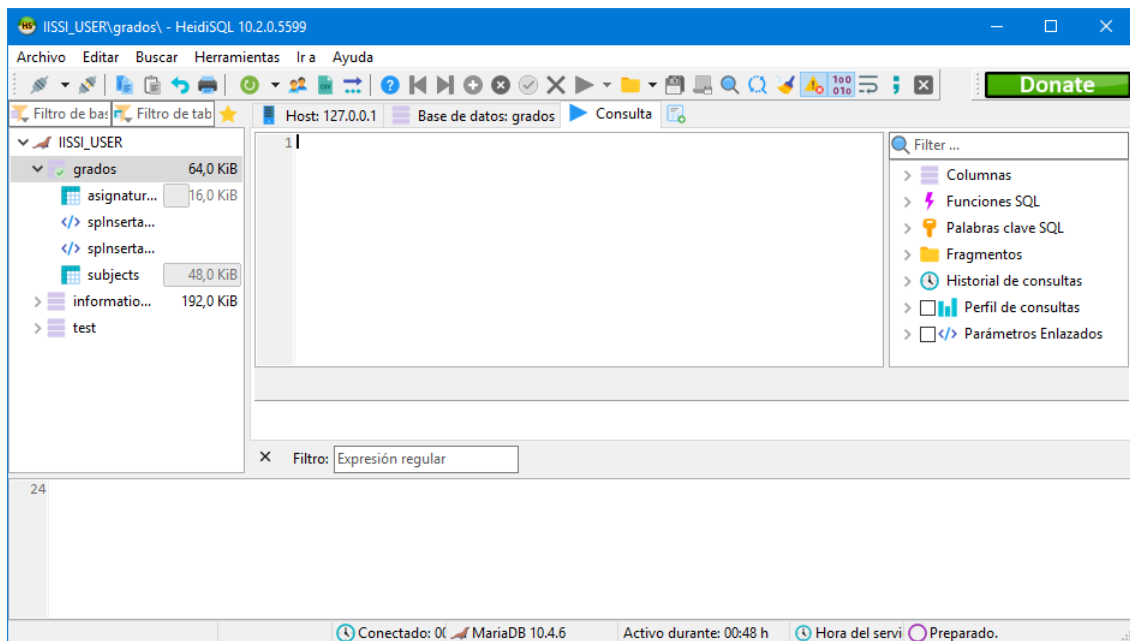
- Crear tablas mediante instrucciones SQL.
- Definir atributos con diferentes tipos de datos.
- Definir claves primarias.
- Definir claves ajenas e implementar relaciones 1:N y N:M.
- Definir campos con autoincremento.
- Definir valores por defecto para campos.

Se usará como base el [documento de requisitos](#) de proyecto de laboratorio proporcionado.

## 1.2. Preparación del entorno

Si se ha instalado MariaDB siguiendo el procedimiento mostrado en el anexo de este documento, el SGBD ha quedado registrado como un servicio Windows, por lo que debería iniciarse automáticamente con el sistema operativo.

Iniciamos HeidiSQL y nos conectamos con el usuario iissi\_user mediante la conexión creada en el laboratorio anterior. Seleccionamos la base de datos grados y abrimos la pestaña de Consulta. En esa pestaña podemos escribir código SQL para que se ejecute en la base de datos seleccionada. El contenido de la pestaña puede escribirse manualmente o ser un archivo .sql cargado:



### 1.3. Consideraciones de estilo

SQL no distingue entre mayúsculas y minúsculas, por lo que es posible que código de diferentes fuentes las usen de forma diferente. Nosotros seguiremos las siguientes normas para unificar el estilo del código:

- Las palabras reservadas del lenguaje SQL (es decir, todo lo que no son nombres definidos por nosotros) se escribirán enteramente en mayúsculas. Por ejemplo: INSERT, UPDATE, DELETE.
- Los nombres de tablas se escribirán con la primera letra en mayúsculas, en CamelCase<sup>1</sup> si están formados por varias palabras, y en plural. Por ejemplo: Degrees, Subjects, Photos, UserComments.
- Los nombres de atributos, constraints, y otros elementos que no sean tablas, se escribirán completamente en minúsculas, en CamelCase si están formados por varias palabras. Por ejemplo: degreeId, course, photoUrl, text.
- El código estará escrito en inglés.

### 1.4. Creación de tabla Degrees (Grados)

Primero implementamos en MariaDB la relación correspondiente a los grados:

```
Degrees(degreeId, name, years)
  PK(degreeID)
  AK(name)
```

<sup>1</sup>El estilo CamelCase une varias palabras sin espacios, con la primera letra de cada una en mayúsculas: EstoEsUnEjemplo. La primera letra de todas puede estar en mayúsculas o minúsculas.



Para ello escribimos el siguiente código en la pestaña de Consulta:

```
DROP TABLE IF EXISTS Degrees;

CREATE TABLE Degrees(
    degreeId INT AUTO_INCREMENT,
    name VARCHAR(60),
    years INT DEFAULT(4),
    PRIMARY KEY (degreeId)
);
```

Observe lo siguiente:

- La primera línea, `DROP TABLE IF EXISTS Degrees;`, se encarga de que se borre la tabla si ya existía. Sin esta línea, se produciría un error si intentamos crear la tabla cuando ya existe (por ejemplo, porque haciendo pruebas queramos ejecutar el script varias veces). Es conveniente que, en un script de creación de tablas, lo primero que se haga sea eliminar todas las tablas si existen.
- El contenido dentro de `CREATE TABLE Degrees` contiene la definición de la tabla. Separamos con comas la definición de atributos o restricciones.
- `INT` y `VARCHAR` son tipos de datos. `VARCHAR` corresponde a un String, y el número indicado entre paréntesis es el número máximo de caracteres.
- En la última línea indicamos cuál es la clave primaria. Se indican entre paréntesis los atributos que forman parte de ella. Si una clave primaria estuviera formada por varios atributos, se incluirían todos separados por comas.
- Nos aseguramos de que los nombres de los IDs sean siempre diferentes (por ejemplo: `degreeId`, `subjectId`). Darle distintos nombres a los ID de todas las tablas evita problemas en caso de despistes.
- A los atributos marcados como `AUTO_INCREMENT` se les dará valor automáticamente siguiendo una secuencia. Los atributos marcados con `AUTO_INCREMENT` deben ser una clave, ya sea primaria o alternativa (que se verán en el siguiente boletín).
- Con `DEFAULT()` indicamos el valor por defecto de un atributo.

## 1.5. Creación de tabla Subjects (Asignaturas)

Implementamos la relación correspondiente a las asignaturas:

```
Subjects(subjectId, degreeId, name, acronym, credits, year, type)
    PK(subjectId)
    FK(degreeId)
    AK(name)
    AK(acronym)
```

con el siguiente código SQL:

```

DROP TABLE IF EXISTS Subjects;

CREATE TABLE Subjects(
    subjectId INT AUTO_INCREMENT,
    degreeId INT,
    name VARCHAR(100),
    acronym VARCHAR(8),
    credits INT,
    year INT,
    type VARCHAR(20),
    PRIMARY KEY (subjectId),
    FOREIGN KEY (degreeId) REFERENCES Degrees (degreeId)
);

```

Añada la eliminación de la tabla si existe antes de crearla y observe lo siguiente:

- Cambiamos la longitud máxima de campos VARCHAR según corresponda.
- Por ahora, no hay nada que evite que el campo type tome cualquier valor, en vez de los valores de enumerado definidos.
- Para definir una clave ajena, incluimos un atributo **del mismo tipo** que la clave que queremos referenciar, y usamos la sentencia FOREIGN KEY. La sentencia recibe primero los atributos que forman parte de la clave ajena, la tabla a la que referencian, y los atributos de la tabla que se referencian. Estos últimos deben ser una clave primaria o alternativa.
- El nombre de la clave ajena no tiene que ser igual que el de la clave primaria referenciada, pero es común que sea así.
- Si una clave ajena está formada por varios atributos, se escribirían entre paréntesis y separados por comas.
- En este caso hemos implementado una relación 1:N. Para estas relaciones, la clave ajena se añade en el lado N de la relación.
- La existencia de claves ajenas puede crear problemas, por ejemplo, si se intenta borrar o modificar una fila que es referenciada por otra mediante una clave ajena. Por defecto, no se permiten eliminar filas referenciadas en otras tablas.

## 1.6. Creación de tabla Groups (Grupos)

Implementamos la relación correspondiente a los grupos:

```

Groups(groupId, subjectId, name, activity, year)
    PK(groupId)
    FK(subjectId)

```

con el siguiente código SQL:

```
CREATE TABLE Groups(
    groupId INT AUTO_INCREMENT,
    name VARCHAR(30),
    activity VARCHAR(20),
    year INT,
    subjectId INT,
    PRIMARY KEY (groupId),
    FOREIGN KEY (subjectId) REFERENCES Subjects (subjectId)
);
```

Añada la eliminación de la tabla si existe antes de crearla. Observe cómo la composición se implementa igual que una relación normal (una asignatura se compone de varios grupos). Sin embargo, se podría indicar que al borrar una asignatura, en vez de producirse error, deben borrarse los grupos que la referencian.

## 1.7. Creación de la tabla Students (Alumnos)

Implementamos la relación correspondiente a los alumnos:

```
Students(studentId, accessMethod, dni, firstName, surname, birthDate, email)
    PK(studentId)
    AK(dni)
    AK(email)
```

con el siguiente código SQL:

```
CREATE TABLE Students(
    studentId INT AUTO_INCREMENT,
    accessMethod VARCHAR(30),
    dni CHAR(9),
    firstName VARCHAR(100),
    surname VARCHAR(100),
    birthDate DATE,
    email VARCHAR(250),
    PRIMARY KEY (studentId)
);
```

Añada la eliminación de la tabla si existe antes de crearla y observe lo siguiente:

- Cuando se usa el tipo CHAR no se indica la longitud máxima del atributo, sino la exacta que tiene que tener, en este caso 9 caracteres. A diferencia de VARCHAR, en este caso no se permitiría introducir un dato con una longitud diferente a 9 caracteres.
- El tipo DATE permite guardar una fecha sin hora. Para guardar una hora se usaría el tipo TIME, y para una fecha con hora, el tipo DATETIME.
- Todavía no se ha implementado la relación N:M entre alumnos y grupos. Con una clave ajena en una o ambas tablas no se podría implementar bien la relación. Hace falta una tabla adicional.

Para implementar la relación N:M debemos crear una tabla adicional que contenga

pares relacionados de alumnos y grupos. La creamos con el siguiente código:

```
CREATE TABLE GroupsStudents(
    groupStudentId INT AUTO_INCREMENT,
    groupId INT,
    studentId INT,
    PRIMARY KEY (groupStudentId),
    FOREIGN KEY (groupId) REFERENCES Groups (groupId),
    FOREIGN KEY (studentId) REFERENCES Students (studentId)
);
```

Añada la eliminación de la tabla si existe antes de crearla y observe lo siguiente:

- La tabla solo tiene los atributos necesarios para relacionar grupos con alumnos.
- El nombre de una tabla usada para representar una relación N:M es la unión de los nombres de las dos tablas que relaciona.
- Ahora mismo, sería posible introducir de forma repetida el mismo alumno asociado al mismo grupo. Esto es incorrecto, ya que un alumno solo puede pertenecer a un grupo una vez. En el siguiente boletín veremos cómo arreglar este problema mediante una restricción.

## 1.8. Creación de la tabla Grades (Notas)

Implementamos la relación correspondiente a las notas:

```
Grades(gradeId, studentId, groupId, value, gradeCall, withHonours)
    PK(gradeId)
    FK(studentId)
    FK(groupId)
```

con el siguiente código SQL:

```
CREATE TABLE Grades(
    gradeId INT AUTO_INCREMENT,
    value DECIMAL(4,2),
    gradeCall INT,
    withHonours BOOLEAN,
    studentId INT,
    groupId INT,
    PRIMARY KEY (gradeId),
    FOREIGN KEY (studentId) REFERENCES Students (studentId),
    FOREIGN KEY (groupId) REFERENCES Groups (groupId)
);
```

Añada la eliminación de la tabla si existe antes de crearla y observe lo siguiente:

- Se ha usado el nombre de atributo gradeCall en vez de simplemente call (convocatoria) debido a que call es una palabra reservada en MariaDB, y no puede usarse como nombre ya que se produciría un error. Las palabras reservadas en MariaDB se pueden consultar en [el listado oficial](#).

- El tipo DECIMAL es usado para definir números que pueden tener decimales. De sus dos parámetros, el primero indica el número total de dígitos, y el segundo, cuántos de ellos son decimales. Como la nota máxima es 10.00, y queremos que haya una precisión de dos cifras decimales, damos como parámetros (4,2).
- El tipo BOOLEAN es usado para definir booleanos. Internamente se almacenan como valores numéricos 0 o 1.
- Varias de las palabras usadas para definir tipos en MariaDB son sinónimas y representan varias formas de escribir el mismo tipo. Por ejemplo, INT es sinónimo de INTEGER; DECIMAL es sinónimo de DEC, NUMERIC de FIXED; y BOOLEAN es sinónimo de BOOL o TINYINT(1).
- Aparte de DECIMAL, existen los tipos FLOAT y DOUBLE. Estos tipos también permiten almacenar números decimales, pero usando números con coma flotante, una manera de representar los números decimales de forma aproximada (debido a los sistemas binarios usados por ordenadores) con una cantidad fija de bits. El tipo DECIMAL almacena los números de forma exacta, aunque a costa de usar más memoria y ser menos eficiente.

## 1.9. Restricciones de borrado de tablas

Por defecto, las bases de datos SQL no permiten borrar una tabla mientras existan otras que la referencien mediante claves ajenas. Por ello, al principio de nuestros scripts de creación de tablas, las borraremos en caso de que ya existan **en orden contrario a su creación**. Ésto garantiza que, en el momento de borrar cada tabla, no existe ninguna otra que la referencie. En el caso de las tablas creadas en este boletín, resultaría:

```
DROP TABLE IF EXISTS Grades;
DROP TABLE IF EXISTS GroupsStudents;
DROP TABLE IF EXISTS Students;
DROP TABLE IF EXISTS Groups;
DROP TABLE IF EXISTS Subjects;
DROP TABLE IF EXISTS Degrees;
```

Realizar estas operaciones al principio del script de creación de tablas permite poder ejecutarlos tantas veces como se desee, ya que si se intenta crear de nuevo una tabla cuando ésta ya existe se produciría un error.

Para poder utilizar el código de la pestaña Consulta, guárdelo con el nombre `tables.sql` por medio de Archivo → Guardar.



# Creación de tablas II

---

## 2.1. Objetivo

El objetivo de esta práctica es implementar restricciones en la creación de tablas mediante scripts SQL. El alumno aprenderá a:

- Añadir restricciones a los atributos al definir tablas.
- Definir claves alternativas.
- Implementar enumerados.
- Definir comportamientos en caso de borrado de una tabla relacionada.

En la práctica anterior se realizó un script SQL de creación de tablas con sus atributos y relaciones. Sin embargo, tal y como se crearon, las tablas permitían la introducción de valores no válidos en los atributos, como números negativos en algunos de ellos o valores que no forman parte de un enumerado. En esta práctica refinaremos las tablas para incluir restricciones de datos con las que implementar claves alternativas, enumerados, y reglas de negocio simples.

## 2.2. Preparación del entorno

Conéctese a la base de datos y abra el archivo `tables.sql` usado en el laboratorio anterior.

## 2.3. Restricciones en tabla Degrees

Las restricciones más comunes se pueden implementar mediante código en la misma línea en la que se define el atributo implicado, mientras que las que requieran una expresión que deba ser verdadera se indican por separado. Añadimos restricciones a la tabla Degrees de la siguiente manera:

```
CREATE TABLE Degrees(
  degreeId INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(60) NOT NULL UNIQUE,
  years INT DEFAULT(4) NOT NULL,
  PRIMARY KEY (degreeId),
  CONSTRAINT invalidDegreeYear CHECK (years >=3 AND years <=5)
);
```

Observe lo siguiente:

- Mediante NOT NULL indicamos que un atributo no puede ser nulo, es decir, no tener valor. Los atributos marcados como clave primaria son NOT NULL por defecto. Sin embargo, puede indicarse como recordatorio y por consistencia.
- Mediante UNIQUE indicamos que un atributo no puede tomar valores repetidos y debe ser único. Ésto lo convierte en clave alternativa.
- Las restricciones indicadas con CONSTRAINT indican expresiones que deben cumplirse. Escribimos el nombre de la constraint (que aparecerá en el mensaje de error si no se cumple), CHECK, y la expresión booleana que debe cumplirse. En este caso, la constraint comprueba que la cantidad de años del grado esté comprendida entre 3 y 5.

## 2.4. Restricciones en la tabla Subjects

Añadimos restricciones a la tabla Subjects de la siguiente manera:

```
CREATE TABLE Subjects(
  subjectId INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL UNIQUE,
  acronym VARCHAR(8) NOT NULL UNIQUE,
  credits INT NOT NULL,
  year INT NOT NULL,
  type VARCHAR(20) NOT NULL,
  degreeId INT NOT NULL,
  PRIMARY KEY (subjectId),
  FOREIGN KEY (degreeId) REFERENCES Degrees (degreeId),
  CONSTRAINT negativeSubjectCredits CHECK (credits >0),
  CONSTRAINT invalidSubjectCourse CHECK (year >= 1 AND year <= 5),
  CONSTRAINT invalidSubjectType CHECK (type IN ('Formacion Básica',
    'Optativa',
    'Obligatoria'))
);
```

Observe lo siguiente:

- La restricción NOT NULL es muy común, ya que por ahora ningún atributo es opcional.
- Una clave ajena también puede ser NOT NULL, indicando que la relación no es opcional. Es la diferencia entre multiplicidad 0..1 y multiplicidad 1.



- En la última constraint hemos implementado el atributo "type" como un enumerado, comprobando que su valor está en un conjunto de posibles valores.

## 2.5. Restricciones en la tabla Groups

Añadimos restricciones a la tabla Groups de la siguiente manera:

```
CREATE TABLE Groups(
    groupId INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(30) NOT NULL,
    activity VARCHAR(20) NOT NULL,
    year INT NOT NULL,
    subjectId INT NOT NULL,
    PRIMARY KEY (groupId),
    FOREIGN KEY (subjectId) REFERENCES Subjects (subjectId),
    UNIQUE (name, year, subjectId),
    CONSTRAINT negativeGroupYear CHECK (year > 0),
    CONSTRAINT invalidGroupActivity CHECK (activity IN ('Teoría', 'Laboratorio'))
);
```

Observe cómo esta vez hemos usado UNIQUE en una línea separada en vez de en la misma línea de un atributo. La diferencia es que esta vez la restricción UNIQUE involucra a tres columnas que se han indicado entre paréntesis: name, year, y subjectId, indicando que no puede haber dos filas en las que la combinación de estos tres atributos sea la misma. De esta manera, logramos que no pueda haber dos grupos con el mismo nombre en la misma asignatura y el mismo año.

## 2.6. Restricciones en las tablas Students y GroupsStudents

Añadimos restricciones a la tabla Students de la siguiente manera:

```
CREATE TABLE Students(
    studentId INT NOT NULL AUTO_INCREMENT,
    accessMethod VARCHAR(30) NOT NULL,
    dni CHAR(9) NOT NULL UNIQUE,
    firstName VARCHAR(100) NOT NULL,
    surname VARCHAR(100) NOT NULL,
    birthDate DATE NOT NULL,
    email VARCHAR(250) NOT NULL UNIQUE,
    PRIMARY KEY (studentId),
    CONSTRAINT invalidStudentAccessMethod CHECK (accessMethod IN
        ('Selectividad', 'Ciclo', 'Mayor', 'Titulado Extranjero'))
);

CREATE TABLE GroupsStudents(
    groupStudentId INT NOT NULL AUTO_INCREMENT,
    groupId INT NOT NULL,
    studentId INT NOT NULL,
    PRIMARY KEY (groupStudentId),
    FOREIGN KEY (groupId) REFERENCES Groups (groupId),
    FOREIGN KEY (studentId) REFERENCES Students (studentId),
    UNIQUE (groupId, studentId)
);
```

Observe la restricción `UNIQUE` en la tabla `GroupsStudents`. Haciendo único el par de claves ajenas, hacemos que cada alumno solo pueda estar asociado a un grupo una vez.

## 2.7. Restricciones en la tabla `Grades`

Añadimos restricciones a la tabla `Grades` de la siguiente manera:

```
CREATE TABLE Grades(
  gradeId INT NOT NULL AUTO_INCREMENT,
  value DECIMAL(4,2) NOT NULL,
  gradeCall INT NOT NULL,
  withHonours BOOLEAN NOT NULL,
  studentId INT NOT NULL,
  groupId INT NOT NULL,
  PRIMARY KEY (gradeId),
  FOREIGN KEY (studentId) REFERENCES Students (studentId),
  FOREIGN KEY (groupId) REFERENCES Groups (groupId),
  CONSTRAINT invalidGradeValue CHECK (value >= 0 AND value <= 10),
  CONSTRAINT invalidGradeCall CHECK (gradeCall >= 1 AND gradeCall <= 3),
  CONSTRAINT RN_002_duplicatedCallGrade UNIQUE (gradeCall, studentId, groupId)
);
```

Observe lo siguiente:

- Esta vez, la restricción `UNIQUE` se ha incluido como parte de una constraint. De esta manera se le da un nombre que se mostrará si no se cumple. En este caso, no hay que incluir la palabra reservada `CHECK`.
- Mediante la restricción de unicidad, nos aseguramos de que no haya varias notas para un mismo alumno, en una misma asignatura, en una misma convocatoria. Sin embargo, puede tener varias notas en convocatorias diferentes.

## 2.8. Comportamientos en borrado para claves ajenas

Como se explicó en el boletín anterior, las claves ajenas imponen por defecto una restricción en el borrado de las filas relacionadas: por ejemplo, consideremos las tablas `Degrees` y `Subjects`, donde `degreeId` es clave primaria de `Degrees` y ajena de `Subjects`, relacionando las asignaturas con el grado en que se imparten.

En este caso, un grado no podría borrarse si hay al menos una asignatura que lo referencia mediante la clave ajena `degreeId` de `Subject`. Sin embargo, este comportamiento de la clave ajena se puede cambiar mediante instrucciones SQL:

- `ON DELETE RESTRICT` impide el borrado en la fila referenciada. Es el comportamiento por defecto.
- `ON DELETE CASCADE` borra la fila de la clave ajena cuando la fila referenciada se borra (borrado en cascada, de ahí su nombre).
- `ON DELETE SET NULL` establece la clave ajena a `NULL` cuando la fila referenciada se borra. Sólo es posible si la clave ajena no tiene la restricción `NOT NULL`, es decir, para multiplicidad 0..1.

- `ON DELETE SET DEFAULT` establece la clave ajena a su valor por defecto cuando la fila referenciada se borra. Sólo es posible si se ha definido un valor por defecto para la clave ajena mediante `DEFAULT()`.

Por ejemplo, supongamos que queremos borrar una asignatura cuando el grado en el que se imparte es borrado de la base de datos. En ese caso, deberíamos modificar la declaración de la tabla `Subjects`:

```
FOREIGN KEY (degreeId) REFERENCES Degrees (degreeId) ON DELETE CASCADE,
```

Observe cómo se ha añadido la instrucción `ON DELETE CASCADE` a la declaración de clave ajena de `degreeId`, lo que provocará que se borre la asignatura si se borra el grado referenciado.

Guardar los cambios del archivo `tables.sql` para utilizarlo en próximos laboratorios.



---

## Laboratorio 3

# Instrucciones SQL I

---

### 3.1. Objetivo

El objetivo de esta práctica es manejar los datos almacenados en la base de datos mediante scripts SQL. El alumno aprenderá a:

- Usar `INSERT` para insertar filas.
- Usar `UPDATE` para actualizar filas.
- Usar `DELETE` para borrar filas.
- Usar `SELECT` para consultar filas.

Hasta ahora hemos creado las tablas que darán soporte a los datos, pero no hemos introducido datos en ellas, ni los hemos manejado. En esta práctica usaremos instrucciones para insertar, alterar, borrar y consultar las filas de cada tabla.

### 3.2. Preparación del entorno

Conéctese a la base de datos y ejecute el archivo `tables.sql` contra la base de datos `grados`. Cree un archivo `queries.sql` en el que se escribirán las instrucciones que se irán desarrollando en esta práctica. Antes de cada ejecución de sus consultas, se recomienda ejecutar el script de creación de tablas, para que éstas se encuentren en un estado controlado antes de la consulta o modificación.

### 3.3. INSERT

Para insertar filas en una tabla, usamos INSERT de la siguiente manera:

```
INSERT INTO Degrees VALUES (NULL, 'Tecnologías Informáticas', 4);
```

Observe lo siguiente:

- Se indica primero el nombre de la tabla, y luego los valores de la fila separados por comas y entre paréntesis.
- Por defecto, hay que introducir los valores en el orden que tienen en la tabla.
- Al ID le damos valor NULL, para que se le de un valor automático con incremento. Si diéramos un número, se usaría ese en vez del generado automáticamente.
- Al escribir cadenas de texto deben usarse comillas simples. Aunque MariaDB técnicamente permite usar comillas dobles, no están aconsejadas ya que en lenguajes y SGBD similares causan errores.

Añadamos algunas filas a todas las tablas:

```
INSERT INTO Degrees (name, years) VALUES
('Ingeniería del Software', 4),
('Ingeniería del Computadores', 4),
('Tecnologías Informáticas', 4);

INSERT INTO Subjects (name, acronym, credits, year, type, degreeId) VALUES
('Fundamentos de Programación', 'FP', 12, 1, 'Formacion Básica', 3),
('Lógica Informática', 'LI', 6, 2, 'Optativa', 3);

INSERT INTO Groups (name, activity, year, subjectId) VALUES
('T1', 'Teoría', 2019, 1),
('L1', 'Laboratorio', 2019, 1),
('L2', 'Laboratorio', 2019, 1);

INSERT INTO Students (accessMethod, dni, firstname, surname, birthdate, email) VALUES
('Selectividad', '12345678A', 'Daniel', 'Pérez', '1991-01-01',
'daniel@alum.us.es'),
('Selectividad', '22345678A', 'Rafael', 'Ramírez', '1992-01-01',
'rafael@alum.us.es'),
('Selectividad', '32345678A', 'Gabriel', 'Hernández', '1993-01-01',
'gabriel@alum.us.es');

INSERT INTO GroupsStudents (groupId, studentId) VALUES
(1, 1),
(3, 1);

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(4.50, 1, 0, 1, 1);
```

Observe lo siguiente:

- Hemos añadido a las sentencias INSERT, antes de los valores, las columnas a las que vamos a dar valor. A las columnas no indicadas se les da valor NULL (o el default). El orden de las columnas especificadas no tiene por qué coincidir con el que tienen en la tabla, pero sí debe ser coherente con los valores que se indiquen a continuación.
- En un mismo INSERT a una tabla se pueden introducir varias filas separadas por comas.
- Las fechas se introducen como cadenas, siguiendo el formato YYYY-MM-DD.
- Los booleanos se introducen como valores numéricos 0 o 1.

## 3.4. UPDATE

Para modificar una o varias filas, usamos UPDATE de la siguiente manera:

```
42 UPDATE Students
43     SET birthdate='1998-01-01', surname='Fernández'
44     WHERE studentId=3;
```

```
UPDATE Students
  SET birthdate = '1998-01-01', surname='Fernández'
  WHERE studentId = 3;
```

Observe lo siguiente:

- Se pueden actualizar varios atributos a la vez.
- La query tiene tres partes: la tabla afectada, los atributos que van a ser modificados, y una condición limitando las filas afectadas.
- Si omitimos la cláusula WHERE, todas las filas serán actualizadas. Cuidado.

Las actualizaciones también pueden usar los valores antiguos al dar nuevos valores, por ejemplo, la siguiente actualización reduce a la mitad los créditos de todas las asignaturas:

```
UPDATE Subjects
  SET credits = credits/2;
```

## 3.5. DELETE

Para borrar filas de una tabla se usa la instrucción DELETE FROM. Podemos borrar filas de la siguiente manera:

```
DELETE FROM Grades
  WHERE gradeId = 1;
```

Observe lo siguiente:

- La query tiene dos partes: la tabla afectada, y una condición limitando las filas borradas.
- **¡Cuidado!** Si omitimos la cláusula WHERE, todas las filas de la tabla serán borradas.
- Por defecto, no se puede borrar una fila que esté referenciada por otra mediante una clave ajena. Habría primero que eliminar la referencia.

Si queremos que al borrar una fila se borren aquellas filas que la referencian mediante claves ajenas (en vez de producirse un error), tenemos que activar el borrado en cascada mediante ON DELETE CASCADE, como se explicó en el boletín anterior. Nótese que, en cualquier caso, se mantiene la integridad referencial de la base de datos, impidiéndose que existan referencias a filas que no existen.

### 3.6. SELECT

La consulta de datos de una base de datos es fundamental. El resultado de este tipo de consultas es siempre una tabla con filas y columnas determinadas por la consulta.

Por ejemplo, podríamos seleccionar los nombres y apellidos de alumnos de acceso por selectividad de la siguiente manera:

```

55 SELECT firstname, surname
56 FROM Students
57 WHERE accessMethod = 'Selectividad';

```

students (2×3)	
firstname	surname
Daniel	Pérez
Rafael	Ramírez
Gabriel	Fernández

Observe lo siguiente:

- La query tiene tres partes: las columnas a obtener, las tablas implicadas (puede haber varias), y una condición (opcional) limitando las filas seleccionadas.
- Si queremos obtener todas las columnas presentes en la tabla seleccionada, podemos usar \*: `SELECT * FROM...`

Las columnas a seleccionar no tienen por qué ser sólo las ya existentes en las tablas. Podemos definir cálculos a devolver como columnas. Por ejemplo:

```

SELECT credits >3
FROM Subjects;

```

En ese caso, el valor devuelto será un boolean, indicando si en cada fila el número de créditos es mayor que 3.

También podemos pedir valores agregados (sumas, medias, etc.). Pedir estos valores implica que solo se devolverá una fila. Si además solo se pide una columna, se devolverá un valor único:

```

62 SELECT AVG(credits)
63 FROM Subjects;

```

Resultado #1 (1×1)	
AVG(credits)	4,5000

Observe lo que ocurre cuando se piden como columnas valores agregados junto con una columna de la tabla:



```

65 SELECT AVG(credits), SUM(credits), name
66 FROM Subjects;

```

subjects (3×1)		
AVG(credits)	SUM(credits)	name
4,5000	9	Fundamentos de Programación

Al pedirse valores agregados, solo se devuelve una fila, que corresponde al valor en cuestión calculado para toda la tabla. El nombre devuelto es simplemente el de la primera de las filas. No tiene sentido pedir valores agregados junto con atributos que cambian de fila a fila.

Uno de los valores agregados más útiles es `COUNT`. En su variante más común, cuenta el número de filas devueltas:

```

65 SELECT COUNT(*)
66 FROM Subjects
67 WHERE credits > 4;

```

Resultado #1 (1×1)	
COUNT(*)	
1	

Sin embargo, podemos incluir una expresión que limite las filas que se están contando:

```

70 SELECT COUNT(DISTINCT accessMethod)
71 FROM Students;

```

Resultado #1 (1×1)	
COUNT(DISTINCT accessMethod)	
1	

## 3.7. Vistas

En ocasiones, es útil guardar los resultados de una consulta para luego utilizarlos en otras consultas como si fueran una tabla más. De esta manera se simplifica en gran medida la creación de consultas anidadas complejas. Esto se puede hacer mediante vistas, en las que damos un nombre a una consulta `SELECT` que luego se puede usar como si fuera una tabla.

Por ejemplo, podríamos crear una vista que contenga las notas del grupo con ID 18:

```

CREATE OR REPLACE VIEW ViewGradesGroup18 AS
SELECT * FROM Grades WHERE groupId = 18;

```

Y a continuación usarla en diferentes consultas:

```
SELECT MAX(value) FROM ViewGradesGroup18;
SELECT COUNT(*) FROM ViewGradesGroup18;
SELECT * FROM ViewGradesGroup18 WHERE gradeCall = 2;
```

También podemos usar una vista dentro de otra:

```
CREATE OR REPLACE VIEW ViewGradesGroup18Call1 AS
    SELECT * FROM ViewGradesGroup18 WHERE gradeCall = 1;

SELECT * FROM ViewGradesGroup18Call1;
```

## 3.8. Consultas varias

La posibilidades a la hora de realizar consultas son casi ilimitadas. A continuación se realizarán una serie de consultas que cubren muchos de los casos posibles. Antes de ejecutar las consultas, ejecute el archivo [populate.sql](#), que inserta una mayor cantidad de datos en las tablas.

- Todas las asignaturas.

```
SELECT *
    FROM Subjects;
```

- Asignatura con acrónimo 'FP'.

```
SELECT *
    FROM Subjects
    WHERE acronym='FP';
```

- Nombres y acrónimos de todas las asignaturas.

```
SELECT name, acronym
    FROM Subjects;
```

- Media de las notas del grupo con ID 18.

```
SELECT AVG(VALUE)
    FROM Grades
    WHERE groupId=18;
```

- Total de créditos de las asignaturas del grado de Tecnologías Informáticas (ID 3).

```
SELECT SUM(credits)
    FROM Subjects
    WHERE degreeId=3;
```

- Notas con valor menor que 4 o mayor que 6.

```
SELECT *
    FROM Grades
    WHERE value <4 OR value >6;
```

- Nombres de grupos diferentes.

```
SELECT DISTINCT NAME
    FROM Groups;
```

- Máxima nota del alumno con ID 1.

```
SELECT MAX(VALUE)
    FROM Grades
    WHERE studentId=1;
```

- Alumnos con un apellido igual al acrónimo de alguna asignatura.

```
SELECT *
FROM Students
WHERE surname IN (SELECT acronym FROM Subjects);
```

- IDs de alumnos del curso 2019.

```
SELECT DISTINCT(StudentId)
FROM GroupsStudents
WHERE groupId IN (SELECT groupId FROM Groups WHERE year = 2019);
```

- Alumnos con un DNI terminado en la letra C. Observe cómo % representa cualquier cantidad de caracteres.

```
SELECT *
FROM Students
WHERE dni LIKE('%C');
```

- Alumnos con un nombre de 6 letras. Observe cómo \_ representa un caracter cualquiera.

```
SELECT *
FROM Students
WHERE firstName LIKE('_____'); -- 6 guiones bajos
```

- Alumnos nacidos antes de 1995.

```
SELECT *
FROM Students
WHERE YEAR(birthdate) <1995;
```

- Alumnos nacidos entre enero y febrero.

```
SELECT *
FROM Students
WHERE (MONTH(birthdate) >= 1 AND MONTH(birthdate) <= 2);
```

Observe cómo en las dos últimas consultas es posible incluir otra consulta dentro de la condición de la primera, en lugar de usar valores establecidos a mano.

## 3.9. Ejercicios

Puede practicar las consultas implementando las siguientes:

- Nombre de las asignaturas que son obligatorias.
- Media de las notas del grupo con ID 19, usando el agregador AVG.
- La misma consulta anterior, sin usar AVG.
- Cantidad de nombres de grupo diferentes.
- Notas entre 4 y 6, inclusive.
- Notas con valor igual o superior a 9, pero que no son matrícula de honor. Cree una vista para las notas que son matrícula de honor.



# Instrucciones SQL II

---

## 4.1. Objetivo

El objetivo de esta práctica es realizar consultas `SELECT` avanzadas en las que sean necesarios nuevos comandos. El alumno aprenderá a:

- Usar `ORDER BY`, `LIMIT` y `OFFSET` para devolver filas ordenadas, limitadas, y con paginación.
- Usar `JOIN` para agregar columnas de varias tablas.
- Usar `GROUP BY` para agrupar filas.

## 4.2. Preparación del entorno

Conéctese a la base de datos “grados” y ejecute los scripts `tables.sql` y `populate.sql`.

Cree un archivo `queries-2.sql` para la escritura de las consultas.

## 4.3. ORDER BY, LIMIT, y OFFSET

Las consultas hasta ahora han devuelto las filas ordenadas según el orden en el que fueron almacenadas. Si queremos ordenarlas podemos usar `ORDER BY`. Obtenemos las notas ordenadas por valor (de menor a mayor) de la siguiente manera:

```
SELECT *  
FROM Grades  
ORDER BY value;
```

Podemos realizar consultas más sofisticadas. Por ejemplo, obtenemos las notas aprobadas, ordenadas por el apellido del alumno que las obtuvo en orden inverso:

```
SELECT *  
FROM Grades  
WHERE VALUE >= 5  
ORDER BY (SELECT surname  
FROM Students  
WHERE Students.studentId = Grades.studentId)  
DESC;
```

Observe lo siguiente:

- ORDER BY se coloca después de WHERE, si lo hay. Si no, donde iría WHERE.
- En ORDER BY podemos usar cualquier expresión aplicable a una fila, incluso otras consultas que devuelvan un valor.
- El orden es ascendente (ASC) por defecto. Si queremos que sea descendente, tenemos que incluir DESC.

En muchas ocasiones, no es necesario obtener todas las filas resultantes de una consulta, sino solo unas pocas (por ejemplo, porque se muestre una página con unos pocos resultados). Para limitar los resultados obtenidos, usamos LIMIT. Obtenemos las 5 mejores notas:

```
SELECT *  
FROM Grades  
ORDER BY value DESC  
LIMIT 5;
```

Podemos indicar también que queremos las primeras filas a partir de una posición. Por ejemplo, si queremos paginar las notas, y queremos obtener la segunda página de 5 notas, usamos la siguiente consulta:

```
SELECT *  
FROM Grades  
ORDER BY value DESC  
LIMIT 5 OFFSET 5;
```

Observe cómo a OFFSET no se le indica una página, sino el número de filas a partir de las cuales se empiezan a devolver resultados. Si quisiéramos la tercera página (con 5 notas por página), sería OFFSET 10.

## 4.4. JOIN

Hasta ahora, las consultas `SELECT` han involucrado una tabla (indicada con `FROM`). Si indicamos varias tablas se obtiene el producto cartesiano, es decir, todas las posibles combinaciones de filas entre las tablas:

```
SELECT * FROM Groups, GroupsStudents, Students;
```

Ejecute la query anterior. El resultado contiene todas las columnas de las tablas y 6048 filas, correspondientes a todas las posibles combinaciones de filas de las tablas. Normalmente se desean las combinaciones de filas que están relacionadas por alguna columna, normalmente una que es clave primaria en una tabla y ajena en la otra. De esta forma podemos ampliar las filas de una tabla con columnas de otra.

Para obtener la unión de filas de tablas diferentes a partir de uno o varios atributos, usamos `JOIN`:

```
SELECT *  
FROM Groups  
JOIN GroupsStudents ON (Groups.groupId = GroupsStudents.groupId)  
JOIN Students ON (GroupsStudents.studentId = Students.studentId);
```

Observe lo siguiente:

- Indicamos la tabla cuya información hay que unir a las filas, seguido de cómo se unirán las filas.
- Al haber varias tablas con columnas del mismo nombre, siempre debemos indicar la tabla de la que proviene la columna. En este caso, da igual si usamos una u otra tabla en cada unión.
- Pueden unirse varias tablas a la vez.
- Si un grupo no tiene alumnos, no aparecerá. Igualmente, si un alumno no tiene grupos, no aparecerá. Pueden usarse variantes de `JOIN` para que se devuelvan todas las filas de una tabla añadiendo, **si la hay**, información de la otra. Son los comandos `LEFT JOIN`, `RIGHT JOIN` y similares, consultables en [la página de MariaDB](#).

Cuando se quieren unir tablas que tienen un nombre de columna en común, en lugar de especificar manualmente las columnas sobre las que se realiza la operación `JOIN`, se puede usar en su lugar `NATURAL JOIN`, que realiza la operación automáticamente. Ésto es especialmente útil para unir dos o más tablas mediante relaciones de claves ajenas, ya que tienen el mismo nombre en ambas tablas relacionadas.

Así, la consulta anterior se puede escribir como:

```
SELECT *
FROM Groups
NATURAL JOIN GroupsStudents
NATURAL JOIN Students;
```

## 4.5. GROUP BY

Algunas de las consultas realizadas hasta ahora obtenían valores agregados mediante comandos como MAX, COUNT, o AVG. De esta forma, podíamos obtener un solo máximo, mínimo, etc. Sin embargo, puede ser necesario obtener varias medidas agregadas correspondientes a varios grupos. Para ello usamos GROUP BY. Para obtener la nota media de cada alumno junto con su nombre y apellidos, realizamos la siguiente consulta:

```
SELECT firstName, surname, AVG(value)
FROM Students
JOIN Grades ON (Students.studentId = Grades.studentId)
GROUP BY Students.studentId;
```

Observe lo siguiente:

- Indicamos con GROUP BY el criterio por el que formar grupos.
- Los atributos a seleccionar se seleccionarán por cada grupo. Los atributos no agregados se tomarán de la primera fila del grupo.
- Como sabemos que el nombre y el apellido de los alumnos de cada grupo de filas es el mismo, es correcto pedirlos.

A continuación, realizamos una petición que obtenga la nota media en cada convocatoria de cada asignatura de 2018, teniendo en cuenta solo los aprobados:

```
CREATE OR REPLACE VIEW ViewSubjectGrades AS
SELECT Students.studentId, Students.firstName, Students.surname,
Subjects.subjectId, Subjects.name,
Grades.value, Grades.gradeCall,
Groups.year
FROM Students
JOIN Grades ON (Students.studentId = Grades.studentId)
JOIN Groups ON (Grades.groupId = Groups.groupId)
JOIN Subjects ON (Groups.subjectId = Subjects.subjectId);

SELECT gradeCall, name, AVG(value)
FROM ViewSubjectGrades
WHERE value >= 5 AND year = 2018
GROUP BY gradeCall, subjectId;
```

Observe lo siguiente:

- No existe ninguna tabla que contenga las notas de cada alumno en cada asignatura, ya que las notas están asociadas a grupos, y los grupos a asignaturas.



En vez de crear una consulta muy complicada, creamos primero una vista que junte la información de varias tablas para disponer cómodamente de las notas de cada alumno en cada asignatura.

- Podemos filtrar las filas antes de agruparlas.
- Podemos agrupar según dos criterios, de manera que los grupos se crearán con filas que tengan el mismo valor en todos los atributos especificados.

Por último, obtenemos la nota media de las asignaturas con más de 2 notas (en todos los años):

```
SELECT name, AVG(value), COUNT(*)
FROM ViewSubjectGrades
GROUP BY name
HAVING COUNT(*) >2;
```

Observe cómo filtramos los grupos con una condición en HAVING. No lo confunda con WHERE, que filtra las filas **antes de agruparlas**.

## 4.6. Consultas varias

- Número de alumnos nacidos en cada año.

```
SELECT YEAR(birthdate), COUNT(*)
FROM Students
GROUP BY YEAR(birthdate);
```

- Número de alumnos por grado en el curso 2019.

```
-- Vista con los estudiantes de cada grado
CREATE OR REPLACE VIEW ViewDegreeStudents AS
SELECT Students.*, Degrees.*, Groups.year
FROM Students
JOIN GroupsStudents ON
  (Students.studentId = GroupsStudents.studentId)
JOIN Groups ON (GroupsStudents.groupId = Groups.groupId)
JOIN Subjects ON (Groups.subjectId = Subjects.subjectId)
JOIN Degrees ON (Subjects.degreeId = Degrees.degreeId);

SELECT name, COUNT(DISTINCT(studentId))
FROM ViewDegreeStudents
WHERE year = 2019
GROUP BY degreeId;
```

- Nota máxima de cada alumno, con el nombre y apellidos.

```
SELECT firstName, surname, MAX(VALUE)
FROM ViewSubjectGrades
GROUP BY studentId;
```

- Nombre y número de grupos de teoría de las 3 asignaturas con mayor número de grupos de teoría en el año 2019.

```
-- Vista con las asignaturas de cada grupo
CREATE OR REPLACE VIEW ViewSubjectGroups AS
SELECT Subjects.*, Groups.name AS groupName, Groups.activity, Groups.year AS groupYear
FROM Subjects JOIN Groups ON (Subjects.subjectId = Groups.subjectId);

SELECT name, COUNT(*)
FROM ViewSubjectGroups
WHERE year = 2019 AND activity = 'Teoría'
GROUP BY subjectId
ORDER BY COUNT(*) DESC LIMIT 3;
```

- Nombre y apellidos de alumnos por año que tuvieron una nota media mayor que la nota media del año.

```
-- Vista con la nota media anual
CREATE OR REPLACE VIEW ViewAvgGradesYear AS
SELECT year, AVG(VALUE) AS average
FROM ViewSubjectGrades
GROUP BY year;

SELECT firstName, surname, year AS yearAvg, AVG(VALUE) AS studentAverage
FROM ViewSubjectGrades
GROUP BY studentId, year
HAVING (studentAverage > (SELECT average
FROM ViewAvgGradesYear
WHERE ViewAvgGradesYear.year = yearAvg)
);
```

- Nombre de asignaturas que pertenecen a un grado con más de 4 asignaturas.

```
-- Vista con el numero de asignaturas de cada grado
CREATE OR REPLACE VIEW ViewDegreeNumSubjects AS
SELECT degreeId, COUNT(*) AS numSubjects
FROM Subjects
GROUP BY degreeId;

SELECT name
FROM Subjects
JOIN ViewDegreeNumSubjects ON (Subjects.degreeId = ViewDegreeNumSubjects.degreeId)
WHERE numSubjects >4
```

## 4.7. Ejercicios

Puede practicar las consultas implementando las siguientes:

- Número de suspensos de cada alumno, dando nombre y apellidos.
- La tercera página de 3 grupos, ordenados según su año por orden descendente.
- Un listado de los grupos, añadiendo el acrónimo de la asignatura a la que pertenecen y el nombre del grado.
- Número de métodos de acceso diferentes de los alumnos de cada grupo, dando el id del grupo.
- Nota ponderada por créditos de cada alumno, dando nombre y apellidos, del curso 2019 en la primera convocatoria. Pista: modifique la vista ViewSubjectGrades añadiendo el atributo que falta. La nota ponderada es igual a la suma de cada nota multiplicada por los créditos de su asignatura, dividida entre la suma de todos los créditos de las asignaturas.

---

## Laboratorio 5

# APIs REST y Silence

---

### 5.1. Objetivo

El objetivo de esta práctica es usar el framework de backend Silence para implementar una API RESTful con la que acceder y modificar los elementos existentes en una base de datos relacional. El alumno aprenderá a:

- Instalar el framework Silence
- Crear y configurar un proyecto Silence
- Crear los scripts de creación y poblado de la BD del proyecto
- Crear los endpoints asociados a la base de datos
- Realizar pruebas sobre la API RESTful creada.

### 5.2. Introducción

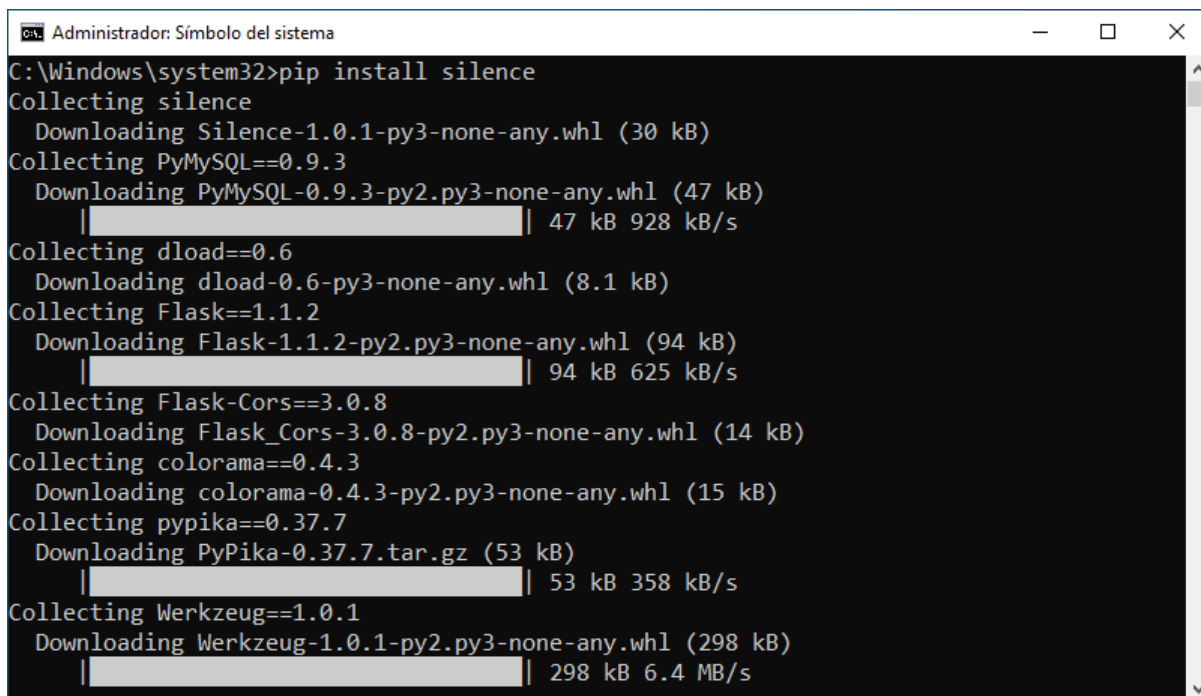
Silence es un framework con propósito educativo creado en la Universidad de Sevilla para facilitar la construcción de APIs RESTful y aplicaciones web a partir de una base de datos relacional. Silence cuenta con una metodología de trabajo basada en proyectos, en la que cada proyecto contiene una estructura de base de datos, aplicación web y configuración propia.

Silence es una herramienta de código abierto, y su código fuente está disponible en su [repositorio en GitHub](#).

### 5.3. Preparación del entorno

Silence se encuentra publicado en el índice de paquetes de Python, por lo que puede instalarse haciendo uso del gestor de paquetes `pip`, que debería estar disponible si se siguieron las instrucciones de instalación del primer laboratorio.

Para instalar Silence, abriremos una consola **con permisos de administrador**<sup>1</sup> y ejecutaremos el comando `pip install silence`:



```

C:\Windows\system32>pip install silence
Collecting silence
  Downloading Silence-1.0.1-py3-none-any.whl (30 kB)
Collecting PyMySQL==0.9.3
  Downloading PyMySQL-0.9.3-py2.py3-none-any.whl (47 kB)
    |████████████████████| 47 kB 928 kB/s
Collecting dload==0.6
  Downloading dload-0.6-py3-none-any.whl (8.1 kB)
Collecting Flask==1.1.2
  Downloading Flask-1.1.2-py2.py3-none-any.whl (94 kB)
    |████████████████████| 94 kB 625 kB/s
Collecting Flask-Cors==3.0.8
  Downloading Flask_Cors-3.0.8-py2.py3-none-any.whl (14 kB)
Collecting colorama==0.4.3
  Downloading colorama-0.4.3-py2.py3-none-any.whl (15 kB)
Collecting pypika==0.37.7
  Downloading PyPika-0.37.7.tar.gz (53 kB)
    |████████████████████| 53 kB 358 kB/s
Collecting Werkzeug==1.0.1
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
    |████████████████████| 298 kB 6.4 MB/s

```

Esto instalará Silence y todas sus dependencias. Si el proceso es correcto, se nos informará de que todo ha quedado instalado:

```

Successfully installed Flask-1.1.2 Flask-Cors-3.0.8 Jinja2-2.11.2 MarkupSafe-1.1.1
PyMySQL-0.9.3 Six-1.15.0 Werkzeug-1.0.1 certifi-2020.6.20 chardet-3.0.4 click-7.1.2
colorama-0.4.3 dload-0.6 idna-2.10 itsdangerous-1.1.0 pypika-0.37.7 requests-2.24.
0 silence-1.0.1 urllib3-1.25.10

```

Podemos comprobar que tenemos acceso al comando `silence` usándolo en la consola, por ejemplo, comprobando la versión que se ha instalado con el comando `silence --version`:

```

PS C:\Users\Agu\Desktop> silence --version
Silence v2.1.0

```

<sup>1</sup>Los permisos de administrador son necesarios para instalar Silence a nivel de sistema y poder usar el comando `silence` en cualquier lugar (el equivalente en Linux es `sudo pip install silence`). Se puede realizar una instalación sin permisos especiales usando un [entorno virtual de Python](#), pero esta operación no está cubierta por este boletín.

Si se desea actualizar el framework Silence para estar al día con una actualización publicada basta con ejecutar el comando de instalación, pero introduciendo `--upgrade` antes del paquete a actualizar: `pip install --upgrade silence`

Si ya cuenta con una versión de Silence instalada del curso pasado, recomendamos encarecidamente que la actualice para asegurarse de que cuenta con la última versión, usando el comando anterior: `pip install --upgrade silence`.

## 5.4. Creación de un nuevo proyecto

Desde este punto en adelante, no es necesario tener permisos de administración en la consola

Para crear un proyecto Silence, navegaremos con la consola a la ubicación donde deseemos crear el proyecto y ejecutaremos el comando `silence new <nombre> --template blank`, donde `<nombre>` es el nombre que le daremos a este proyecto. La opción `--template blank` indica que se creará un proyecto con una estructura de base de datos vacía, por lo que podemos usar la base de datos en la que hemos trabajado en los anteriores laboratorios:

```
PS C:\Users\Agu\Desktop> silence new proyecto_lab --template blank
The Silence project "proyecto_lab" has been created using the template 'blank'.
```

## 5.5. Configuración del proyecto

Cada proyecto contiene un archivo `settings.py` que almacena la configuración específica del proyecto. A continuación mostraremos las principales opciones a configurar en este archivo:

El parámetro `DEBUG_ENABLED` controla si se muestran o no mensajes de depuración. Si se activan, aparecerán mensajes que permiten conocer qué está haciendo internamente el framework en cada momento. Los mensajes de depuración aparecen en gris en la consola.

```
DEBUG_ENABLED = False
```

El parámetro `DB_CONN` configura los datos de acceso a la BD y la base de datos a usar para el proyecto:

```
DB_CONN = {
  "host": "127.0.0.1",
  "port": 3306,
  "username": "iissi_user",
  "password": "iissi$user",
  "database": "grados",
}
```

---

El parámetro SQL\_SCRIPTS controlan qué scripts SQL se ejecutarán, y en qué orden, cuando se le indique a Silence que debe crear la base de datos del proyecto. Los scripts SQL deben colocarse en la carpeta sql/ del proyecto.

Para que nuestro proyecto contenga toda la información necesaria sobre la base de datos, proporcionamos todos los scripts SQL en el repositorio de la asignatura [IISSI1-ArchivosAuxiliares](#) dentro de la carpeta laboratorio/sql/ incluyendo:

- [tables.sql](#)
- [populate.sql](#)

Y cualquier otro script que sea necesario para crear nuestra BD.

A continuación, configuraremos el parámetro SQL\_SCRIPTS para que se ejecuten en el orden correcto:

```
SQL_SCRIPTS = [
  "tables.sql",
  "populate.sql",
]
```

---

Es importante tener en cuenta que los scripts SQL deben estar en el orden adecuado, ya que si no, podríamos tener problemas al crear la base de datos e insertar los datos correspondientes.

Los parámetros HTTP\_PORT y API\_PREFIX permiten configurar el puerto a usar para el servidor HTTP y el prefijo que tendrán todas las rutas de la API, respectivamente. Mantendremos los valores por defecto:

```
HTTP_PORT = 8080
API_PREFIX = "/api/v1"
```

---

El parámetro USER\_AUTH\_DATA permite indicar qué tabla es la que se usará para identificar usuarios, y dentro de esta tabla, qué columnas corresponden al identificador del usuario y a su contraseña. Estos datos son necesarios para que Silence pueda proveer registro y login de usuarios, así como protección de endpoints para usuarios registrados y control de roles.

En los scripts SQL proporcionados para este boletín, se ha añadido un campo `password` a la tabla de estudiantes, para permitir hacer login y registro con ellos.

```
USER_AUTH_DATA = {  
    "table": "Students",  
    "identifier": "email",  
    "password": "password",  
}
```

---

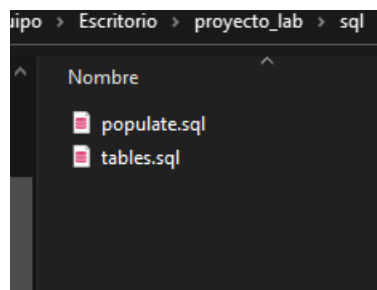
Finalmente, el parámetro `SECRET_KEY` se utiliza para elementos internos del framework que requieren una cadena secreta aleatoria criptográficamente segura. Este parámetro se genera de manera automática cuando se crea un nuevo proyecto usando el comando `silence new`.

Existen muchos otros parámetros de configuración que pueden encontrarse en [la Wiki de Silence en GitHub](#).

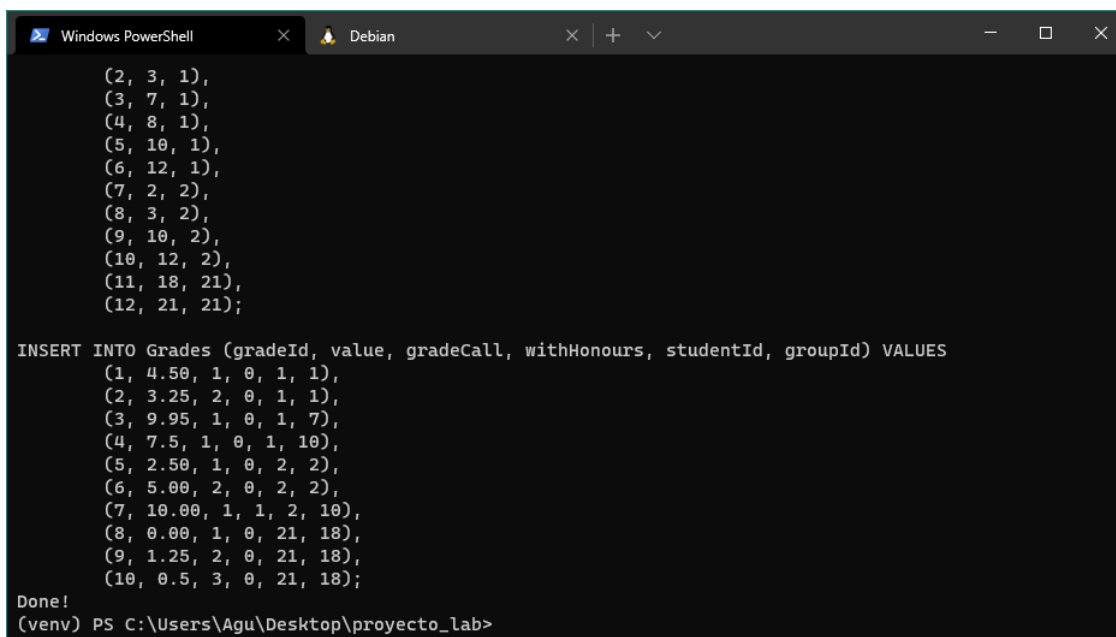
## 5.6. Definición de la BD del proyecto

Cada proyecto Silence tiene asociada una estructura de base de datos y una carga inicial de datos, de manera que sea sencillo desplegar éstas siempre que la configuración de acceso al SGBD sea correcta.

Una vez hemos configurado en la sección anterior los datos de acceso a la BD, añadiremos a nuestro proyecto los scripts de creación de la BD y de la carga inicial de datos, en la carpeta `sql/`. Usaremos para ello los scripts proporcionados en conjunto con este boletín:



Una vez estos scripts se encuentran en la carpeta adecuada, y su orden de ejecución ha sido definido en el parámetro `SQL_SCRIPTS` del archivo `settings.py`, podemos inicializar automáticamente la base de datos usando el comando `silence createdb` desde la carpeta raíz del proyecto. Si el proceso es correcto, se mostrará todo el código SQL que se está ejecutando:



```

(2, 3, 1),
(3, 7, 1),
(4, 8, 1),
(5, 10, 1),
(6, 12, 1),
(7, 2, 2),
(8, 3, 2),
(9, 10, 2),
(10, 12, 2),
(11, 18, 21),
(12, 21, 21);

INSERT INTO Grades (gradeId, value, gradeCall, withHonours, studentId, groupId) VALUES
(1, 4.50, 1, 0, 1, 1),
(2, 3.25, 2, 0, 1, 1),
(3, 9.95, 1, 0, 1, 7),
(4, 7.5, 1, 0, 1, 10),
(5, 2.50, 1, 0, 2, 2),
(6, 5.00, 2, 0, 2, 2),
(7, 10.00, 1, 1, 2, 10),
(8, 0.00, 1, 0, 21, 18),
(9, 1.25, 2, 0, 21, 18),
(10, 0.5, 3, 0, 21, 18);

Done!
(venv) PS C:\Users\Agu\Desktop\proyecto_lab>

```

Tener la estructura de la BD y sus datos almacenados en el proyecto favorecen un despliegue de las mismas más sencillo, y permite devolverla a un estado controlado en cualquier momento simplemente ejecutando el comando `silence createdb`.

## 5.7. Definición de los endpoints del proyecto

Un endpoint representa una operación que puede realizarse sobre un recurso, que se ejecuta cuando se recibe una petición HTTP determinada a una ruta concreta. En Silence, los endpoints del proyecto se definen usando la notación JSON, donde para cada endpoint se define obligatoriamente:

- La ruta del endpoint
- El método HTTP asociado
- La consulta SQL que debe ejecutarse

Opcionalmente, también puede especificarse:

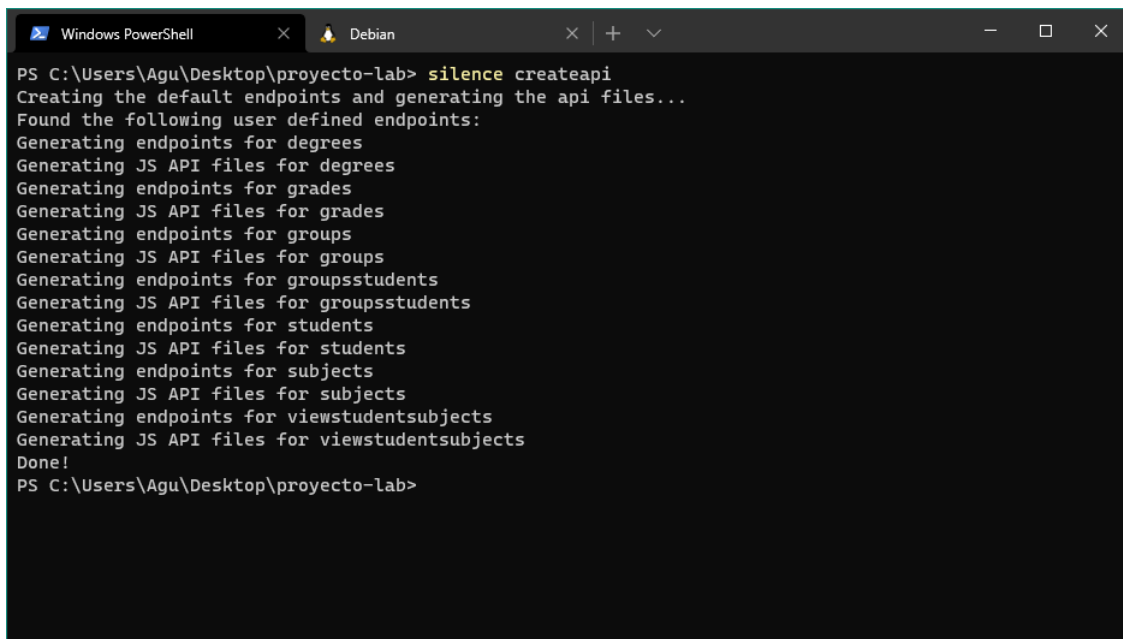
- La descripción del endpoint
- Si puede ser usado por todos los usuarios o sólo por los usuarios autenticados
- Si puede ser usado por cualquier usuario autenticado, o sólo por aquellos que tengan un determinado rol

Generalmente, se desean definir las operaciones CRUD (crear, leer, actualizar y borrar) para cada recurso, representados como tablas en nuestra BD, y operaciones de lectura para las vistas. La definición de estos endpoints se realizan en archivos JSON que deben estar en la carpeta `endpoints/`.

Silence es capaz de analizar la estructura de una base de datos, y generar automáticamente los endpoints que implementan las operaciones descritas



anteriormente para todas las entidades encontradas en ella. Para ello, basta con usar el comando `silence createapi` desde la carpeta raíz del proyecto:



```
PS C:\Users\Agu\Desktop\proyecto-lab> silence createapi
Creating the default endpoints and generating the api files...
Found the following user defined endpoints:
Generating endpoints for degrees
Generating JS API files for degrees
Generating endpoints for grades
Generating JS API files for grades
Generating endpoints for groups
Generating JS API files for groups
Generating endpoints for groupsstudents
Generating JS API files for groupsstudents
Generating endpoints for students
Generating JS API files for students
Generating endpoints for subjects
Generating JS API files for subjects
Generating endpoints for viewstudentssubjects
Generating JS API files for viewstudentssubjects
Done!
PS C:\Users\Agu\Desktop\proyecto-lab>
```

Si el proceso finaliza correctamente, se muestra una lista de las entidades para las que se han generado endpoints. Es importante recordar que este proceso se realiza en base a **la estructura de la BD**, por lo que es necesario tener definida en ella las tablas antes de ejecutar el comando.

Los endpoints autogenerados se encuentran en la carpeta `endpoints/auto/`, agrupados en un archivo JSON por cada entidad encontrada en la BD. A modo de ejemplo, podemos analizar el endpoint de consulta para la tabla `Degrees`:

```
"getAll": {
  "route": "/degrees",
  "method": "GET",
  "sql": "SELECT * FROM degrees",
  "description": "Gets all degrees",
  "auth_required": false,
  "allowed_roles": ["*"]
}
```

Como se observa, este endpoint asocia la consulta SQL `SELECT * From degrees` a la ruta `/degrees` y el método `GET`. Además, se especifica que no es necesario estar autenticado para acceder al endpoint, que puede ser usado por usuarios con cualquier rol (`"*"`), y se provee una descripción del mismo.

Por defecto, los endpoints de consulta pueden ser usados por usuarios no autenticados, mientras que los de modificación (creado, actualizado y borrado) sólo pueden ser usados por usuarios autenticados. Es posible definir endpoints personalizados mediante archivos JSON en la carpeta `endpoints/`, usando la sintaxis mostrada.

## 5.8. Ejecución del proyecto y uso de endpoints

Una vez está la base de datos inicializada y los endpoints definidos, podemos lanzar nuestro proyecto usando el comando `silence run` desde la carpeta raíz. Silence nos informará de la dirección en la que está corriendo el servidor, así como de los endpoints disponibles:

```
Windows PowerShell
PS C:\Users\Agu\Desktop\proyecto-lab> silence run
Silence v2.1.0

Endpoints loaded:
  - http://127.0.0.1:8080/api/v1 (GET)
  - http://127.0.0.1:8080/api/v1/degrees (GET/POST)
  - http://127.0.0.1:8080/api/v1/degrees/<degreeId> (GET/PUT/DELETE)
  - http://127.0.0.1:8080/api/v1/grades (GET/POST)
  - http://127.0.0.1:8080/api/v1/grades/<gradeId> (GET/PUT/DELETE)
  - http://127.0.0.1:8080/api/v1/groups (GET/POST)
  - http://127.0.0.1:8080/api/v1/groups/<groupId> (GET/PUT/DELETE)
  - http://127.0.0.1:8080/api/v1/groupsstudents (GET/POST)
  - http://127.0.0.1:8080/api/v1/groupsstudents/<groupStudentId> (GET/PUT/DELETE)
  - http://127.0.0.1:8080/api/v1/login (POST)
  - http://127.0.0.1:8080/api/v1/register (POST)
  - http://127.0.0.1:8080/api/v1/students (GET/POST)
  - http://127.0.0.1:8080/api/v1/students/<studentId> (GET/PUT/DELETE)
  - http://127.0.0.1:8080/api/v1/subjects (GET/POST)
  - http://127.0.0.1:8080/api/v1/subjects/<subjectId> (GET/PUT/DELETE)
  - http://127.0.0.1:8080/api/v1/viewstudentssubjects (GET)

Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
```

Para hacer uso de los endpoints utilizaremos la extensión [REST Client](#) para VSCode, cuyo uso explicamos a continuación.

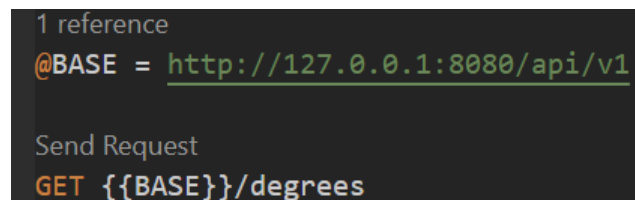
### 5.8.1. Consultas GET

REST Client utiliza archivos `.http` que pueden contener una o varias consultas. En nuestra asignatura, agruparemos estas consultas en varios archivos, según la entidad a la que estemos accediendo. A modo de ejemplo, haremos consultas GET a Degrees, para lo cual crearemos un archivo `degrees.http` **en una nueva carpeta, a la que llamaremos requests**.

En primer lugar, definiremos la URL base, a partir de la cual se construirán todas las consultas a nuestra API, para evitar tener que repetirla cada vez:

```
@BASE = http://127.0.0.1:8080/api/v1
```

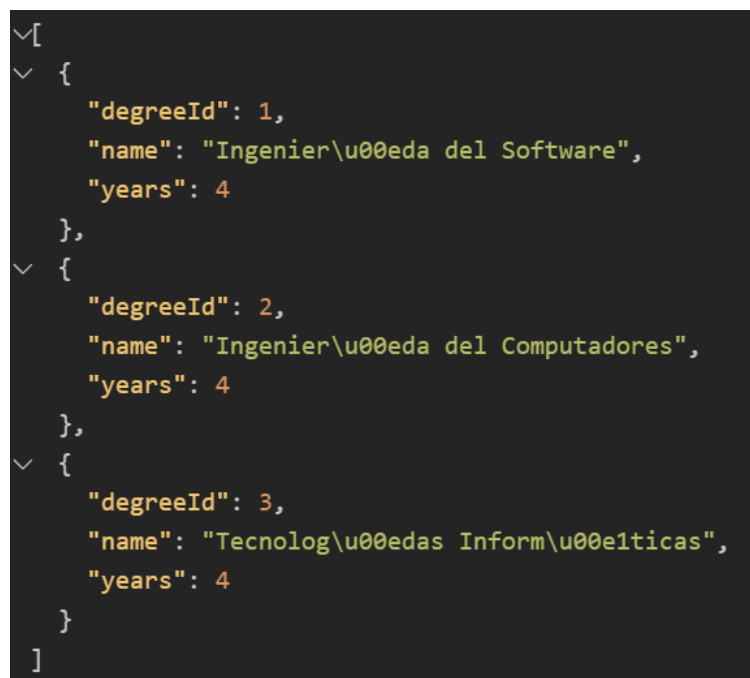
A continuación, podemos usarla para realizar consultas GET al endpoint `/degrees`. Para hacer una petición HTTP basta con indicar el verbo y la URL, y pulsar en el botón "Send request" que aparece en la parte superior. Se pueden usar las variables definidas anteriormente usando dobles llaves:



```
1 reference
@BASE = http://127.0.0.1:8080/api/v1

Send Request
GET {{BASE}}/degrees
```

El servidor ejecutará la consulta SQL asociada a la ruta y método, y devolverá una respuesta en formato JSON:



```
[
  {
    "degreeId": 1,
    "name": "Ingenier\u00eda del Software",
    "years": 4
  },
  {
    "degreeId": 2,
    "name": "Ingenier\u00eda del Computadores",
    "years": 4
  },
  {
    "degreeId": 3,
    "name": "Tecnolog\u00edas Inform\u00e1ticas",
    "years": 4
  }
]
```

Además de una consulta a todas las entradas de la tabla Degrees, los endpoints generados automáticamente permiten consultar una de ellas por su ID, mediante la

ruta /degrees/id. Para ello, basta con indicar el ID en la URL, y pulsar en el botón "Send request":

```
5   ###
6
7   Send Request
8   GET {{BASE}}/degrees/2
```

A lo que el servidor responde:

```
{
  "degreeId": 2,
  "name": "Ingenier\u00eda del Computadores",
  "years": 4
}
```

Es importante destacar que, para tener **varias consultas en el mismo archivo**, éstas deben **separarse mediante una línea con tres almohadillas (###)**.

Intentemos ahora, por ejemplo, borrar el grado con ID 2 mediante el endpoint asociado al verbo DELETE:

```
Send Request
DELETE {{BASE}}/degrees/2
```

```
{
  "code": 401,
  "message": "Unauthorized"
}
```

En este caso, el servidor responde con un código de error HTTP 401 (no autorizado). Esto se debe a que el endpoint correspondiente está protegido, y sólo los usuarios autenticados pueden acceder a él. En la siguiente sección registraremos un nuevo usuario, y aprenderemos a hacer login con uno de los usuarios ya existentes para tener acceso a los endpoints protegidos.

### 5.8.2. Registro

Silence provee un endpoint por defecto, `/register`, para realizar el registro de un nuevo usuario. Para que funcione correctamente, el parámetro `USER_AUTH_DATA` debe estar definido correctamente en `settings.py`, indicando el nombre de la tabla que almacena los usuarios y los campos (columnas) que se usan para el login. En nuestro caso, serán la tabla "Students" y las columnas "email" y "password" respectivamente.

Una vez configurado, se le pueden enviar al servidor los datos de un nuevo usuario en formato JSON, incluyendo tantos campos como sean necesarios en la tabla Students. La contraseña enviada será almacenada de manera segura en forma de hash<sup>2</sup>, y no será visible en la respuesta.

Para registrar un nuevo usuario, enviaremos una petición HTTP POST a la ruta `/register`, con los datos del nuevo usuario en formato JSON. Cuando se envían datos en una petición POST o PUT en el cuerpo de la petición, es importante indicarle al servidor el formato de los mismos, que en este caso es `"application/json"`:

```
Send Request
POST {{BASE}}/register
Content-Type: application/json

{
  "firstName": "Elvis",
  "surname": "Presley",
  "dni": "98421245J",
  "birthDate": "1935-01-08",
  "accessMethod": "Titulado Extranjero",
  "email": "elvis@alum.us.es",
  "password": "elvis1234"
}
```

Si el registro es exitoso, el servidor devolverá los datos del usuario creado, y un token de sesión. Este token es necesario para acceder a los endpoints protegidos, ya que contiene la información del usuario que ha iniciado sesión:

---

<sup>2</sup>Más información sobre los conceptos básicos de este proceso: <https://security.stackexchange.com/a/31846>

```

{
  "sessionToken": ".eJwtjstqwzAQRX9FiCxDbQp1rzqoqE0kNKFF2BsjbASW071aBtK_72R6TCrM4e594fCMGBKZ8zjYm1La0dzmcAu5PidI4QLxoVuCe19zOMzZFydg1vCND1BINzsFWGsXZe8nLtq2-CrZ7QUXEh1qgxn8F01OH369ARTmXc17TDVo_Mx5TeY1_fHK1R6g5S-1rj2uvVX60SbRhDq0HLFHRNhflEf92u3cY0Z0B1mDQotXSM0cnCcH_aSaa01HoyzstHKCjZwKV1fFSGgefTXTta01IvFkF9rnBAV1Bj-K71HTBP6e8fyJ5Yng.YYkreQ.dGI0jayT3dkCZv3PgABLJ0Wqb7k",
  "user": {
    "accessMethod": "Titulado Extranjero",
    "birthDate": "Tue, 08 Jan 1935 00:00:00 GMT",
    "dni": "98421245J",
    "email": "elvis@alum.us.es",
    "firstName": "Elvis",
    "studentId": 22,
    "surname": "Presley"
  }
}

```

Por defecto, los tokens de sesión son válidos durante 24 horas. Pasado ese tiempo, caducan y no son considerados tokens válidos.

Si se intentara registrar de nuevo el mismo usuario, el servidor devolverá un error:

```

{
  "code": 400,
  "message": "There already exists another user with that email"
}

```

### 5.8.3. Login

Como se ha mostrado en la sección anterior, una manera de obtener un token de sesión es registrar un usuario. Otra manera es iniciar sesión con un usuario ya existente. Para ello, se debe enviar una petición POST a la ruta `/login`, con los datos de inicio de sesión del usuario en formato JSON. El servidor emitirá una respuesta idéntica a la del registro, con los datos del usuario que ha iniciado sesión (excepto su contraseña) y un token de sesión:

```
POST {{BASE}}/login
Content-Type: application/json

{
  "email": "elvis@alum.us.es",
  "password": "elvis1234"
}
```

```
{
  "sessionToken": ".eJwtjstqwzAQRX9FiCxDbQp1rzqoqE0
kNKFf2BsJbASW071aBtK_72R6TCrM4e594fCMGBKZ8zjYm1LaOdz
mcAu5PidI4QLxoVuCe19zOMzZFydg1vCND1BINzsFWGsXZe8nLtq
2-CrZ7QUXEh1qgxn8F01OH369ARTmXcl7TDVo_Mx5TeY1_fHK1R6
g5S-1rj2uvVX60SbRhDq0HLFHRNhflEf92u3cY0Z0B1mDQotXSM0
cnCcH_aSaa01HoyzstHKCjZwKV1fFSGgefTXtTa01IvFkF9rnBAV
lBj-K71HTBP6e8fyJ5Yng.YYkreQ.dGI0jayT3dkCZv3PgABLJ0
Wqb7k",
  "user": {
    "accessMethod": "Titulado Extranjero",
    "birthDate": "Tue, 08 Jan 1935 00:00:00 GMT",
    "dni": "98421245J",
    "email": "elvis@alum.us.es",
    "firstName": "Elvis",
    "studentId": 22,
    "surname": "Presley"
  }
}
```

## 5.9. Creación, actualización y borrado

Si queremos acceder a los endpoints protegidos para usuarios autenticados, debemos enviar el token de sesión como cabecera de la petición HTTP. En nuestro caso, el nombre de la cabecera será Token, y el valor será el token de sesión.

Dada la longitud de estos tokens, y el hecho de que expiran a las 24h de ser creados, no es una buena idea proveerlos directamente como cadenas en la cabecera de nuestras peticiones HTTP, ya que dificultaría la lectura del archivo y las peticiones asociadas dejarían de funcionar al día siguiente. En su lugar, podemos darle un nombre a la petición de login anterior y, usando los datos que nos ha devuelto el servidor en ella, referenciarlo en las siguientes peticiones.

Como ejemplo, en el siguiente fragmento de código creamos un nuevo grado (Degree) mediante una petición POST a la ruta /degrees. Para que el servidor nos permita realizar la operación, antes realizaremos un login, y referenciaremos el token de sesión obtenido:

```
# @name login
Send Request
POST {{BASE}}/login
Content-Type: application/json

{
  "email": "elvis@alum.us.es",
  "password": "elvis1234"
}

####

Send Request
POST {{BASE}}/degrees
Content-Type: application/json
Token: {{login.response.body.sessionToken}}

{
  "name": "Ingeniería de Sistemas",
  "years": 4
}
```

Ahora sí, el servidor aceptará la operación y nos devolverá el ID del nuevo recurso creado:

```
{
  "lastId": 6
}
```

Para modificar un recurso, usaremos el método HTTP PUT. Para ello, en la petición HTTP, indicaremos el ID del recurso que queremos modificar, y en el cuerpo de la petición, los nuevos atributos del recurso. Es importante destacar que, en una petición de tipo PUT, se deben enviar todos los atributos del recurso, y no sólo los que se desean modificar:

```
Send Request
PUT {{BASE}}/degrees/6
Content-Type: application/json
Token: {{login.response.body.sessionToken}}

{
  "name": "Ingeniería de la Salud",
  "years": 4
}
```



Podemos comprobar que los cambios se han efectuado en la base de datos:

degreeId	name	years
1	Ingeniería del Software	4
2	Ingeniería del Computadores	4
3	Tecnologías Informáticas	4
6	Ingeniería de la Salud	4

Finalmente, podemos eliminar un recurso mediante una petición DELETE. Al igual que en las peticiones anteriores, los endpoints generados automáticamente requieren el token de sesión para modificar el estado de la base de datos. Para eliminar el grado que acabamos de crear, en la petición HTTP, indicaremos el ID del recurso que queremos eliminar:

```
Send Request
DELETE {{BASE}}/degrees/6
Token: {{login.response.body.sessionToken}}
```

En el caso de las peticiones DELETE, no es necesario enviar contenido de ningún tipo en el cuerpo de la petición.

## 5.10. Definición de endpoints personalizados

Todos los endpoints que hemos usado hasta el momento han sido generados automáticamente por Silence, usando el comando `silence createapi`. Aunque son útiles, en ocasiones queremos definir nuestros propios endpoints, para ejecutar consultas más complejas.

Por ejemplo, consideremos que es necesario tener un endpoint que, a partir de un grado concreto, nos devuelva las asignaturas que se imparten en él. De acuerdo a la teoría, la ruta adecuada para tal endpoint sería:

```
GET /degrees/$degreeId/subjects
```

Asimismo, la consulta SQL que devuelve las asignaturas de un grado es:

```
SELECT * FROM Subjects WHERE degreeId = $degreeId
```

Donde `$degreeId` es la ID del grado en cuestión.

Para asociar esa consulta a la ruta deseada, debemos definir un nuevo endpoint. Para ello, crearemos un nuevo archivo JSON en la carpeta `endpoints` (NO en la carpeta `auto`, que contiene los endpoints autogenerados). Dado que la entidad que se devolverá es `Degree`, llamaremos al archivo `degrees.json`.

Este archivo contendrá una lista de endpoints, con el mismo formato que los generados por Silence. Por ejemplo, el endpoint deseado se definiría así:

```
{
  "getByDegree": {
    "route": "/degrees/$degreeId/subjects",
    "method": "GET",
    "sql": "SELECT * FROM Subjects WHERE degreeId = $degreeId",
    "auth_required": false,
    "allowed_roles": ["*"],
    "description": "Gets the subjects of a degree"
  }
}
```

Si ejecutamos de nuevo el servidor con `silence run`, podremos ver que el endpoint definido se ha cargado correctamente. Podemos probarlo haciendo una petición GET:

```
Send Request
GET {{BASE}}/degrees/2/subjects
```

A lo que el servidor responde con las asignaturas de dicho grado:

```
✓[
✓ {
  "acronym": "IMD",
  "credits": 6,
  "degreeId": 2,
  "name": "Introducci\u00f3n a la Matematica Discreta",
  "subjectId": 6,
  "type": "Formacion Basica",
  "year": 1
},
✓ {
  "acronym": "RC",
  "credits": 6,
  "degreeId": 2,
  "name": "Redes de Computadores",
  "subjectId": 7,
  "type": "Obligatoria",
  "year": 2
},
✓ {
  "acronym": "TG",
  "credits": 6,
  "degreeId": 2,
  "name": "Teor\u00eda de Grafos",
```



---

## Laboratorio 6

# Procedimientos, funciones y disparadores

---

### 6.1. Objetivo

El objetivo de esta práctica es implementar disparadores y procedimientos en SQL. El alumno aprenderá a:

- Usar procedimientos y funciones para definir un conjunto de órdenes reutilizable.
- Usar disparadores para implementar restricciones complejas y reglas de negocio.

### 6.2. Preparación del entorno

Conéctese a la base de datos “grados” y ejecute en ella los scripts `tables.sql` y `populate.sql`.

Cree un archivo `triggers.sql` para la escritura de los disparadores y un archivo `procedures.sql` para los procedimientos y funciones.

### 6.3. Procedimientos

Un procedimiento es un conjunto de sentencias SQL a las que se les asigna un nombre y que reciben unos parámetros, de manera análoga a las funciones de otros lenguajes. La principal diferencia entre un procedimiento y una función en SQL es que

los primeros no devuelven ningún valor, mientras que las segundas tienen un valor de retorno.

Generalmente, se emplean para definir un conjunto de instrucciones reutilizable que se espera emplear a menudo. Por ejemplo, para implementar el requisito funcional por el cual se pide borrar las notas de un alumno con un DNI dado mediante el siguiente procedimiento:

```
-- RF-006
DELIMITER //
CREATE OR REPLACE PROCEDURE procDeleteGrades(studentDni CHAR(9))
BEGIN
    DECLARE id INT;
    SET id = (SELECT studentId FROM Students WHERE dni=studentDni);
    DELETE FROM Grades WHERE studentId=id;
END //
DELIMITER ;
```

Observe lo siguiente:

- En las instrucciones de código que forman parte del procedimiento (entre BEGIN y END), los puntos y coma pueden ser problemáticos, ya que el intérprete puede confundirlos con el fin del procedimiento. Para evitar esto, durante su definición **cambiamos el símbolo usado para delimitar instrucciones** a // mediante la sentencia DELIMITER. Al terminar de definir el procedimiento, reestablecemos ; como delimitador.
- La primera instrucción, CREATE OR REPLACE PROCEDURE, **declara el procedimiento que se va a definir** y lo reemplaza si ya existe uno con ese nombre.
- Por consistencia, y para distinguirlos visualmente más fácilmente de las funciones, todos los nombres de procedimientos que definamos **empezarán por proc.**
- Se indican los parámetros de entrada **entre paréntesis, incluyendo el tipo de los mismos**. Si hay más de un parámetro, éstos son separados por comas.
- Dentro de un procedimiento **se pueden declarar variables** mediante DECLARE incluyendo su tipo, y se les puede asignar un valor mediante SET. El valor a asignar puede ser el resultado de una consulta SQL.
- En este procedimiento, buscamos la ID del estudiante que tiene el DNI proporcionado, la almacenamos en una variable y eliminamos todas las notas del estudiante cuya ID hemos almacenado.

Los procedimientos almacenados pueden ser llamados mediante CALL, por ejemplo:

```
CALL procDeleteGrades('12345678A');
```

A continuación, crearemos un procedimiento que borre todos los datos de la base de datos:

```
DELIMITER //
CREATE OR REPLACE PROCEDURE procDeleteData()
BEGIN
    DELETE FROM Grades;
    DELETE FROM GroupsStudents;
    DELETE FROM Students;
    DELETE FROM Groups;
    DELETE FROM Subjects;
    DELETE FROM Degrees;
END //
DELIMITER ;
```

## 6.4. Funciones

Las funciones son muy parecidas a los procedimientos, pero se diferencian de ellos en que las funciones sí pueden devolver valores, por lo que deben declarar su tipo de retorno. Las funciones SQL pueden usarse para obtener datos que requieran varias instrucciones SQL y se quieran consultar a menudo.

Mediante una función SQL podemos implementar el requisito funcional para obtener la nota media de un alumno:

```
DELIMITER //
CREATE OR REPLACE FUNCTION avgGrade(studentId INT) RETURNS DOUBLE
BEGIN
    DECLARE avgStudentGrade DOUBLE;
    SET avgStudentGrade = (SELECT AVG(value) FROM Grades
                           WHERE Grades.studentId = studentId);
    RETURN avgStudentGrade;
END //
DELIMITER ;
```

Observe lo siguiente:

- El comienzo de la declaración es similar, sustituyendo PROCEDURE por FUNCTION e indicando los parámetros de entrada si los hay, pero se debe indicar el tipo que retorna la función mediante RETURNS.
- Al igual que en los procedimientos, se pueden declarar y asignar valores a variables mediante DECLARE y SET.
- Mediante la instrucción RETURN devolvemos el resultado. Puede devolverse una variable o el resultado de una consulta directamente.
- Como en los procedimientos, se debe realizar el cambio de delimitador para que el intérprete no confunda los ; del interior de la función con el final de la misma.

Al contrario que los procedimientos, las funciones se pueden usar en cualquier lugar en el que se podría usar una variable, como consultas, o el cuerpo de procedimientos/funciones/disparadores. Para consultar el valor de una función, en lugar de usar CALL, podemos hacer una consulta SELECT:

```
SELECT avgGrade(2);
```

Resultado #1 (1r × 1c)	
avgGrade(2)	
5,833333333	

También podemos consultarla como si fuera una columna más, por ejemplo, para obtener el nombre y los apellidos de un alumno junto con su nota media:

```
SELECT firstName, surname, avgGrade(studentId) FROM Students;
```

## 6.5. Disparadores

Mediante los disparadores (triggers) podemos asociar la ejecución de código a la inserción, modificación, o borrado de filas en una tabla. Esto nos puede servir, por ejemplo, para comprobar restricciones complejas e implementar reglas de negocio.

Como ejemplo, implementamos la regla de negocio según la cual para obtener matrícula de honor la nota debe ser mayor o igual a 9:

```
DELIMITER //
CREATE OR REPLACE TRIGGER RN001_triggerWithHonours
BEFORE INSERT ON Grades
FOR EACH ROW
BEGIN
    IF (new.withHonours = 1 AND new.value <9.0) THEN
        SIGNAL SQLSTATE '45000' SET message_text =
            'You cannot insert a grade with honours whose value is less than 9';
    END IF;
END//
```

Observe lo siguiente:

- Se debe cambiar el delimitador al igual que en los casos anteriores.
- Mediante `BEFORE INSERT ON Grades` indicamos que el disparador debe ejecutarse **justo antes de insertar filas** en la tabla Grades. Podríamos sustituir `BEFORE` por `AFTER`, pero en este caso, para cuando se lanzara el disparador, la nota ya habría sido insertada.
- En vez de `INSERT` podrían usarse `UPDATE` o `DELETE` para vincular disparadores a la actualización o borrado de filas, respectivamente.
- Con un `INSERT` podríamos insertar varias filas a la vez. Algo similar ocurre con `UPDATE` y `DELETE`. Con `FOR EACH ROW` indicamos que el disparador debe ejecutarse **por cada fila afectada**.
- Con `new` hacemos referencia a **la fila que está siendo insertada**, tanto si el disparador se ejecuta antes como después de insertarla.
- Mediante `SIGNAL` podemos hacer que se produzcan errores, **cancelándose la inserción de la fila**. El número después de `SQLSTATE` corresponde al código de error. Existe [una gran cantidad de códigos de error](#), aunque el usual para los errores personalizados es 45000. Con `SET message_text` indicamos cuál es el mensaje del error. Es muy útil incluir un mensaje **tan descriptivo como sea posible**.
- Sería conveniente que se hiciera la comprobación no sólo al insertar una nota, sino al actualizarla. Para que sea así, habría que repetir el trigger, cambiando el nombre y sustituyendo `INSERT` por `UPDATE`.

El disparador anterior es simple, ya que solo contiene la comprobación de un valor y el lanzamiento de un error. Implementemos ahora un disparador que implementa la regla de negocio por la que no se puede poner a un alumno una nota en un grupo en el que no pertenece:

```
CREATE OR REPLACE TRIGGER RN003_triggerGradeStudentGroup
BEFORE INSERT ON Grades
FOR EACH ROW
```



```

BEGIN
  DECLARE isInGroup INT;
  SET isInGroup = (SELECT COUNT(*)
    FROM GroupsStudents
      WHERE studentId = new.studentId AND groupId = new.groupId);
  IF(isInGroup <1) THEN
    SIGNAL SQLSTATE '45000' SET message_text =
      'A student cannot have grades for groups in which they are not registered';
  END IF;
END//

```

Pruebe el disparador anterior y observe lo siguiente:

- Se pueden declarar y asignar variables mediante DECLARE y SET al igual que en los procedimientos y funciones.
- En este caso, buscamos el número de asignaciones a grupos que coinciden con el estudiante y el grupo al que se está intentando asignar la nota. Si no hay ninguna, es porque el estudiante no está en ese grupo, y se lanza un error.

A continuación creamos un disparador que implementa la siguiente regla de negocio: cada vez que se actualice una nota, comprueba si ésta se ha subido en más de 4 puntos. En ese caso, se muestra un error con el nombre del estudiante y la diferencia de la nota nueva con respecto a la antigua:

```

CREATE OR REPLACE TRIGGER RN004_triggerGradesChangeDifference
BEFORE UPDATE ON Grades
FOR EACH ROW
BEGIN
  DECLARE difference DECIMAL(4,2);
  DECLARE student ROW TYPE OF Students;
  SET difference = new.value - old.value;
  IF(difference >4) THEN
    SELECT * INTO student FROM Students WHERE studentId = new.studentId;
    SET @error_message = CONCAT('You cannot add ', difference,
      ' points to a grade for the student',
      student.firstName, ' ', student.surname);
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @error_message;
  END IF;
END//

```

Observe lo siguiente:

- En este caso no se ha asignado el disparador a la inserción de filas, sino a la modificación de una fila, mediante BEFORE UPDATE ON.
- Una de las variables declaradas tiene como tipo una fila de la tabla Students, indicado mediante ROW TYPE OF. Así, podemos acceder a cualquier atributo del estudiante que almacenemos en esa variable.
- Se le asigna un valor mediante una consulta SELECT que sabemos que sólo devolverá una fila.
- La asignación en el caso de variables que representan filas se debe hacer de una forma diferente: incluyendo INTO student dentro de la consulta.
- Podemos hacer referencia a la fila tanto antes de la actualización (new) como después (old).
- Para crear un mensaje personalizado que requiera concatenar varias partes, usamos CONCAT. Como CONCAT no se puede usar en la misma instrucción en la que lanzamos el error, creamos primero el mensaje en una variable y luego lo usamos.
- La variable en la que hemos guardado el mensaje no se ha declarado antes, y

tiene en su nombre el símbolo '@'. Si se usa una variable de esta forma, en vez de ser una variable local es una variable a nivel de usuario, que sigue existiendo y teniendo el mismo valor fuera del disparador. La hemos usado por comodidad a la hora de guardar y usar rápidamente el mensaje de error.

Podemos probar el disparador intentando subir una nota más de 4 puntos:

```
UPDATE Grades SET value = 10.0 WHERE gradeId = 1;  
/* Error de SQL (1644): You cannot add 5.50 points to a grade for the student Daniel Pérez */
```

A continuación, modificaremos el último disparador para que, en vez de lanzarse un error, la nota sólo sea aumentada en 4 puntos:

```
CREATE OR REPLACE TRIGGER RN004_triggerGradesChangeDifference  
BEFORE UPDATE ON Grades  
FOR EACH ROW  
BEGIN  
    DECLARE difference DECIMAL(4,2);  
    SET difference = new.value - old.value;  
    IF(difference >4) THEN  
        SET new.value = old.value + 4;  
    END IF;  
END//
```

Pruebe el disparador anterior y observe cómo se puede reemplazar el valor de los atributos siendo actualizados con new.

## Laboratorio 7

# Transacciones

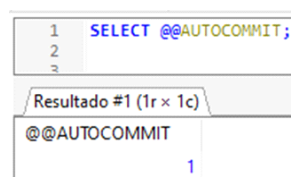
### 7.1. Objetivo

El objetivo de esta práctica es implementar transacciones en SQL. El alumno aprenderá a:

- Activar/desactivar el AUTOCOMMIT.
- Control de transacciones en procedimientos almacenados.
- Diferenciar entre procedimientos transaccionales o no transaccionales.

### 7.2. AUTOCOMMIT

En primer lugar vamos a ilustrar el concepto de transacción como Unidad Lógica de Trabajo. MariaDB, por defecto, fuerza un COMMIT automático después de cada instrucción. Esto está controlado por la variable de entorno AUTOCOMMIT, que por defecto está en ON (AUTOCOMMIT=1).



The screenshot shows a SQL query window with the command `SELECT @@AUTOCOMMIT;` and its result. The result is a single row with the value 1.

1	SELECT @@AUTOCOMMIT;
2	
3	
Resultado #1 (1r x 1c)	
	@@AUTOCOMMIT
	1

Para manejar transacciones que agrupan modificaciones sobre la BD emitiendo varias sentencias es necesario desactivar esta opción, con el comando SET

AUTO COMMIT=0, además de controlar el inicio de la transacción, con la instrucción START TRANSACTION, y la terminación confirmando (COMMIT o COMMIT WORK) o cancelando (ROLLBACK o ROLLBACK WORK). Hay que tener en cuenta que, estando en una transacción activa, si se produce una excepción grave (SQLEXCEPTION) y no se controla mediante código, entonces la transacción abortaría (ROLLBACK implícito).

Para comprobar que por defecto cada instrucción se realiza por separado, es suficiente con ejecutar un fragmento de código que incluya algunas que se realicen satisfactoriamente y otras que no. Por ejemplo, el siguiente código que introduce varias notas, siendo la tercera errónea. Al terminar de ejecutarse, puede observarse que las dos primeras notas sí se han introducido:

```
SET AUTOCOMMIT=1;

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(4.50, 3, 0, 1, 1);

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(7.50, 1, 0, 1, 3);

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(-7.50, 1, 0, 2, 2);
```

Si queremos que todas las instrucciones se ejecuten en una transacción, debemos desactivar la opción AUTO COMMIT y colocar el código dentro de una transacción:

```
SET AUTOCOMMIT=0;

START TRANSACTION;

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(4.50, 3, 0, 1, 1);

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(7.50, 1, 0, 1, 3);

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(-7.50, 1, 0, 2, 2);

COMMIT;
```

Sin embargo, se está ejecutando un script SQL sin código para el control de excepciones, por lo que la ejecución sigue, quedando la transacción activa incluso si se produce un error. Incluso si terminamos el bloque con ROLLBACK WORK, el error producido evitará que se siga ejecutando código, y no se evitará que se introduzcan las dos primeras notas. Para lograr el resultado deseado (que solo se realice el bloque de instrucciones si no hay errores, y que se deshagan todos los cambios si se producen), es necesario colocar el código SQL dentro de un procedimiento.

### 7.3. Control de transacciones en un procedimiento SQL almacenado

En este apartado se maneja un procedimiento almacenado SQL que maneja una transacción y controla excepciones (SQLEXCEPTIONS o WARNINGS). Para ello crearemos un procedimiento almacenado que inserta un nuevo grado y una nueva asignatura en

la base de datos asociada al nuevo grado. Deseamos que si ocurre algún error no se inserte ni el grado ni la asignatura. Sin embargo, con una versión inicial podemos confirmar que este no es el caso. Por ejemplo, si el nombre de la asignatura está repetido, el grado seguirá insertado:

```
DELIMITER //
CREATE OR REPLACE PROCEDURE procNewDegreeSubject(degreeName VARCHAR(60),
degreeYears INT, subjectName VARCHAR(100), subjectAcronym VARCHAR(8),
subjectCredits INT, subjectYear INT, subjectType VARCHAR(20))
BEGIN
    DECLARE newDegreeId INT;
    INSERT INTO Degrees (name, years) VALUES (degreeName, degreeYears);
    SET newDegreeId = (SELECT degreeId FROM Degrees WHERE name=degreeName);
    INSERT INTO Subjects (degreeId, name, acronym, credits, year, type)
VALUES (newDegreeId, subjectName, subjectAcronym,
subjectCredits, subjectYear, subjectType);
END //
DELIMITER ;

CALL procNewDegreeSubject('Nuevo Grado', 4, 'Lógica Informática', 'LI', 60, 3, 'Obligatoria');
```

A continuación, modificamos el procedimiento para que se lleve a cabo en una transacción, de forma que si se produce un error, se realizará un rollback:

```
DELIMITER //
CREATE OR REPLACE PROCEDURE procNewDegreeSubject(degreeName VARCHAR(60),
degreeYears INT, subjectName VARCHAR(100), subjectAcronym VARCHAR(8),
subjectCredits INT, subjectYear INT, subjectType VARCHAR(20))
BEGIN
    START TRANSACTION;
    tblock: BEGIN
        DECLARE newDegreeId INT;
        DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
        BEGIN
            ROLLBACK;
            RESIGNAL;
        END;
        INSERT INTO Degrees (name, years) VALUES (degreeName, degreeYears);
        SET newDegreeId = (SELECT degreeId FROM Degrees WHERE name=degreeName);
        INSERT INTO Subjects (degreeId, name, acronym, credits, year, type)
VALUES (newDegreeId, subjectName, subjectAcronym,
subjectCredits, subjectYear, subjectType);
        COMMIT;
    END tblock;
END //
DELIMITER ;
```

Observe lo siguiente:

- Si se realiza la misma llamada al procedimiento con datos erróneos, no se introduce ni el grado ni la asignatura.
- El contenido del procedimiento se ha introducido en un bloque denominado "tblock". Esto es necesario para poder definir qué hacer en caso de error.
- Mediante DECLARE EXIT HANDLER se define cómo tratar excepciones que se produzcan. En nuestro caso, hacemos rollback y dejamos que la excepción se lance.
- Solo si se llega al final de la transacción se hace commit.

## 7.4. Transacciones concurrentes

El uso de forma conjunta de HeidiSQL y Silence nos permite observar cómo los cambios no se hacen efectivos fuera de la transacción hasta que se hace commit.

Reinicie la base de datos y ejecute el siguiente código para iniciar una transacción e insertar dos notas sin llegar a cerrar la transacción:

```
SET AUTOCOMMIT=0;

START TRANSACTION;

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(4.50, 3, 0, 1, 1);

INSERT INTO Grades (value, gradeCall, withHonours, studentId, groupId) VALUES
(7.50, 1, 0, 1, 3);

SELECT * FROM Grades;
```

Observe que los resultados obtenidos por la consulta incluyen las notas añadidas justo antes:

gradeId	studentId	groupId	value	gradeCall	withHonours
9	21	18	1,25	2	0
10	21	18	0,50	3	0
32	1	1	4,50	3	0
33	1	3	7,50	1	0

Sin realizar commit, use Silence para pedir un listado de las notas mediante la consulta GET BASE/grades. Observe cómo las notas nuevas no aparecen en el listado. Ya que no se ha hecho commit, los cambios son solo visibles dentro de la transacción, mientras que fuera de ella (es decir, en otras transacciones) la base de datos permanece igual que antes de comenzar la transacción.

Por último, ejecute la instrucción COMMIT en SQL y vuelva a realizar la petición http. Observe que ahora sí aparecen las nuevas notas al haberse realizado los cambios de la transacción.

# Entorno de trabajo

---

## A.1. Objetivo

El objetivo de esta práctica es configurar el entorno de trabajo inicial para el desarrollo de la asignatura. El alumno aprenderá a:

- Instalar MariaDB y HeidiSQL.
- Crear conexiones con BBDD locales.
- Crear usuarios para las BBDD.
- Cargar y ejecutar un script SQL.
- Ejecutar una consulta simple sobre una BD.
- Instalar y configurar Python.
- Instalar Visual Studio Code y extensiones relacionadas.
- Instalar y configurar Git.

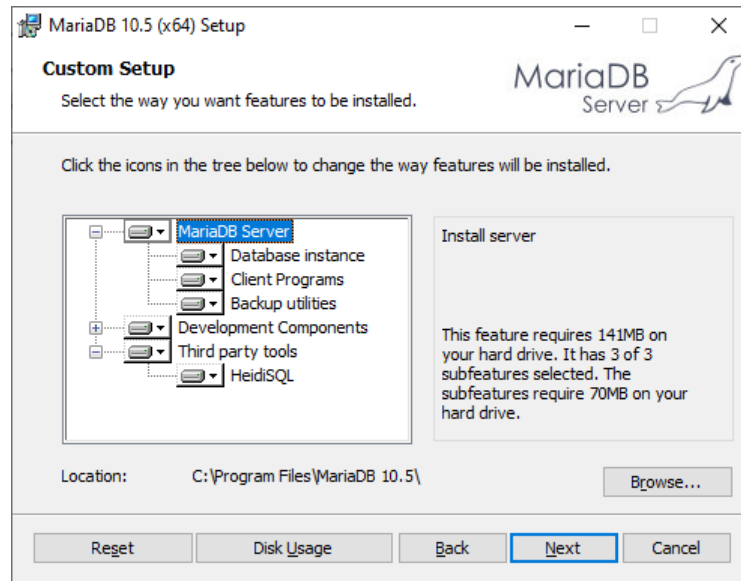
## A.2. Instalación de MariaDB y HeidiSQL

Durante el curso usaremos MariaDB, un *fork* de MySQL que comparte sus mismas funciones pero tiene una licencia más permisiva. En esta sección se explica su instalación para Windows, cuyo instalador puede encontrarse [aquí](#).

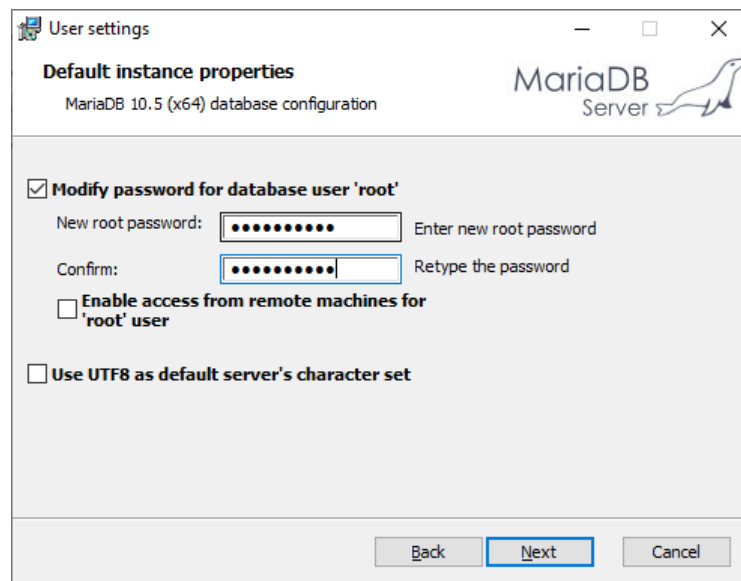
En Linux, puede instalarse el paquete `mariadb-server` usando `apt` o el gestor de paquetes correspondiente. En macOS, se puede instalar MariaDB usando Homebrew, como se explica [aquí](#). El cliente HeidiSQL sólo está disponible para Windows, en otros SO pueden usarse alternativas como [MySQL Workbench](#) o [DBeaver](#).

Accederemos a la [página de descargas de MariaDB](#) y descargaremos e iniciaremos

el instalador para Windows. Cuando se nos pregunte, dejaremos marcadas todas las características a instalar:



A continuación se nos preguntará por la contraseña del usuario *root* (administrador). Para nuestra BD usaremos la contraseña *iissi\$root*:



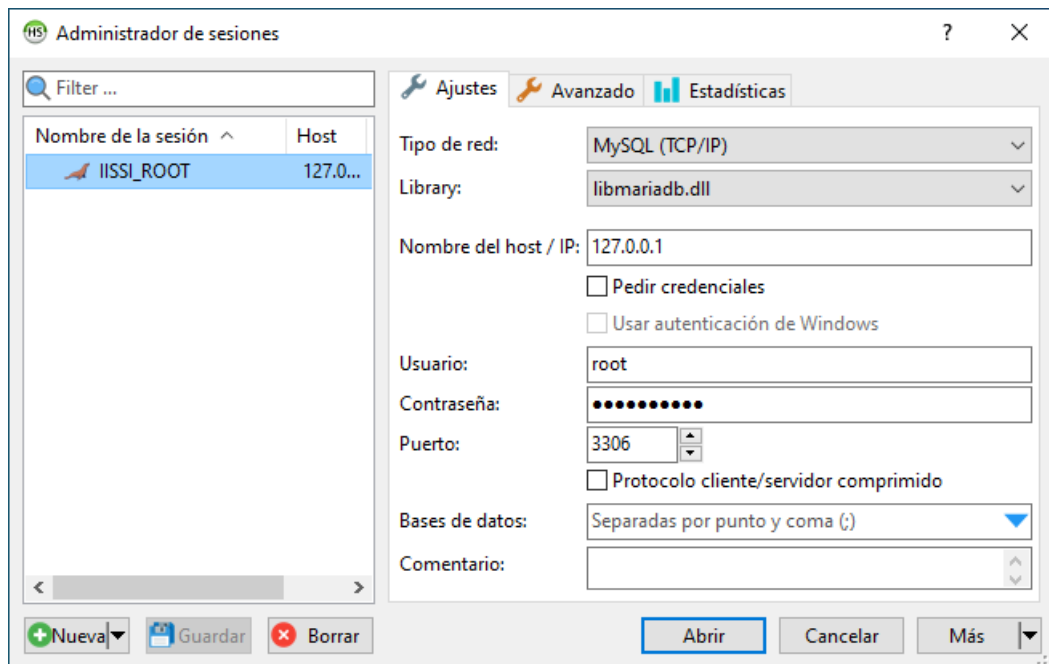
Aceptaremos las opciones por defecto a partir de este diálogo y esperaremos a que finalice la instalación, tras lo cual habrán quedado instalados tanto MariaDB como el cliente HeidiSQL.

### A.3. Creación de una conexión con HeidiSQL

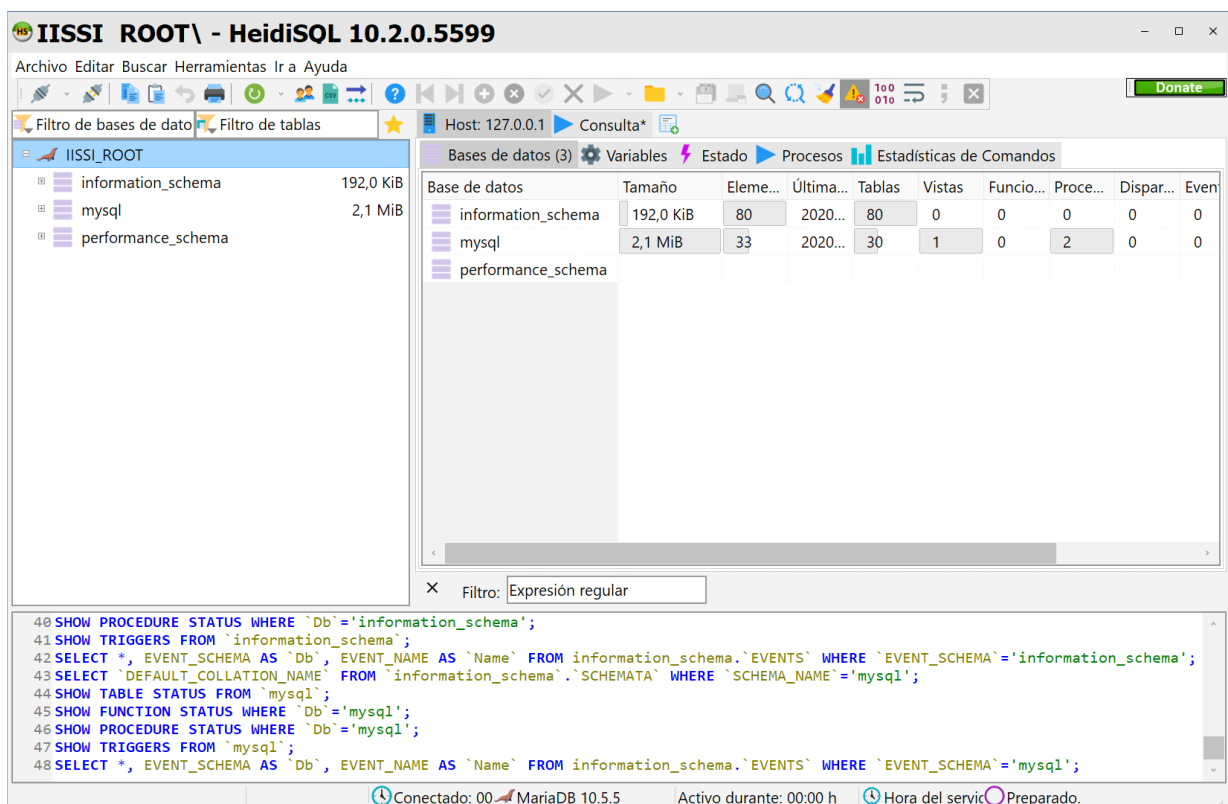
Para trabajar con MariaDB usaremos el cliente HeidiSQL, que debería estar instalado si se han seguido adecuadamente las pautas de la sección anterior.



Ejecutamos HeidiSQL y creamos una nueva sesión, a la que llamaremos IISSI\_ROOT, en la que indicaremos los datos de acceso del usuario root que hemos configurado anteriormente:




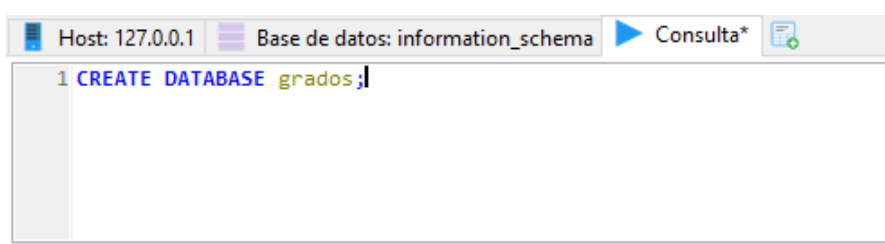
Tendremos acceso a todas las BD que tiene el SGBD. Las que aparecen por defecto (information\_schema, mysql y performance\_schema) son propias del SGBD y no deben ser modificadas manualmente.




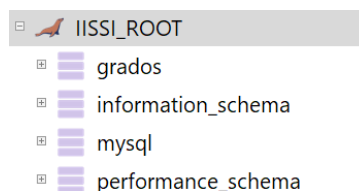
## A.4. Creación de una base de datos

Una “base de datos” o “database” en MariaDB (también llamada “schema”) es un conjunto de tablas, que normalmente se corresponde con un proyecto o aplicación concreto. Mediante el uso de databases, se pueden tener varias aplicaciones con conjuntos diferentes de tablas funcionando sobre un mismo SGBD.

Como ejemplo, crearemos la base de datos “grados”. Para ello, usando el usuario *root*, accederemos a la pestaña “Consulta” y ejecutaremos `CREATE DATABASE grados;`, tras ello, pulsaremos  para ejecutar nuestra consulta:



Si pulsamos sobre la conexión y la actualizamos mediante , podremos ver que se ha creado la base de datos “grados”:

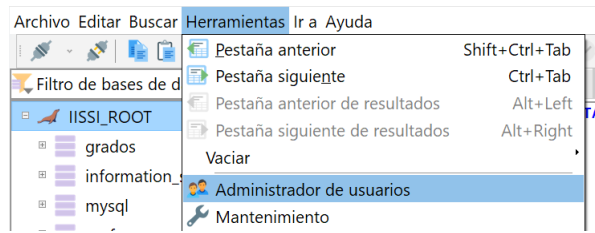



Sin embargo, realizar todas las operaciones con el usuario *root* no es aconsejable. En la siguiente sección, crearemos un nuevo usuario para operar con la base de datos recién creada.


## A.5. Creación de usuarios

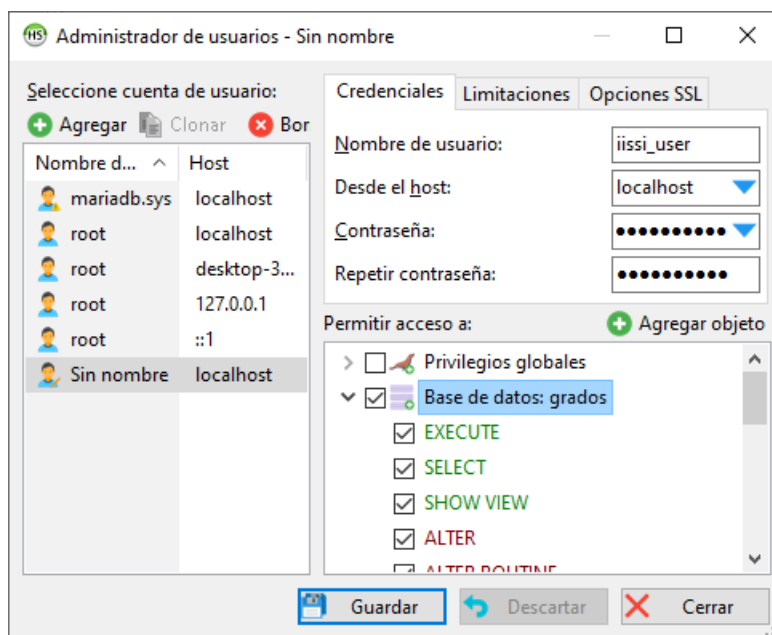
Operar directamente con el usuario *root* está altamente desaconsejado, ya que éste tiene permisos ilimitados sobre el SGBD, y cualquier error cometido puede resultar potencialmente grave. En su lugar, crearemos un usuario y le otorgaremos los permisos adecuados para poder crear y manipular bases de datos.

Crearemos un usuario usando Herramientas → Administrador de usuarios:



Pulsamos en  Agregar y creamos un nuevo usuario con nombre `iissi_user` y clave `iissi$user`. En "Desde el host" se deja marcado "localhost", lo cual indica que el usuario a crear sólo podrá ser accedido desde nuestra máquina, no por conexiones remotas.

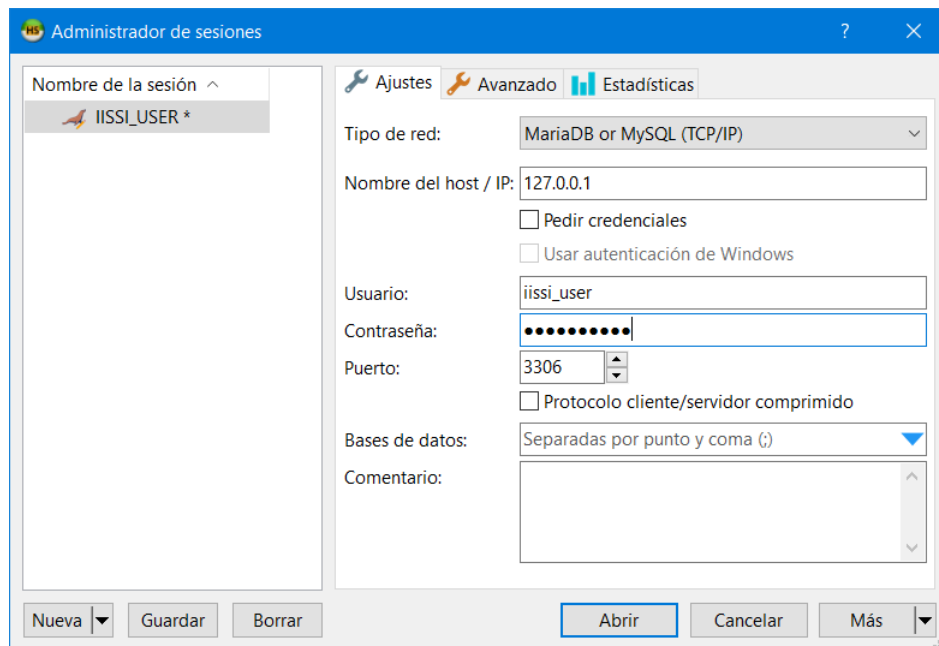
En la parte inferior podremos asignarle permisos al usuario que vamos a crear. Es aconsejable otorgar los mínimos permisos imprescindibles, por lo que el nuevo usuario sólo tendrá permisos para modificar la base de datos "grados". Para otorgarle permisos en la BD que acabamos de crear, la seleccionamos en  Agregar objeto y marcamos todos los permisos.



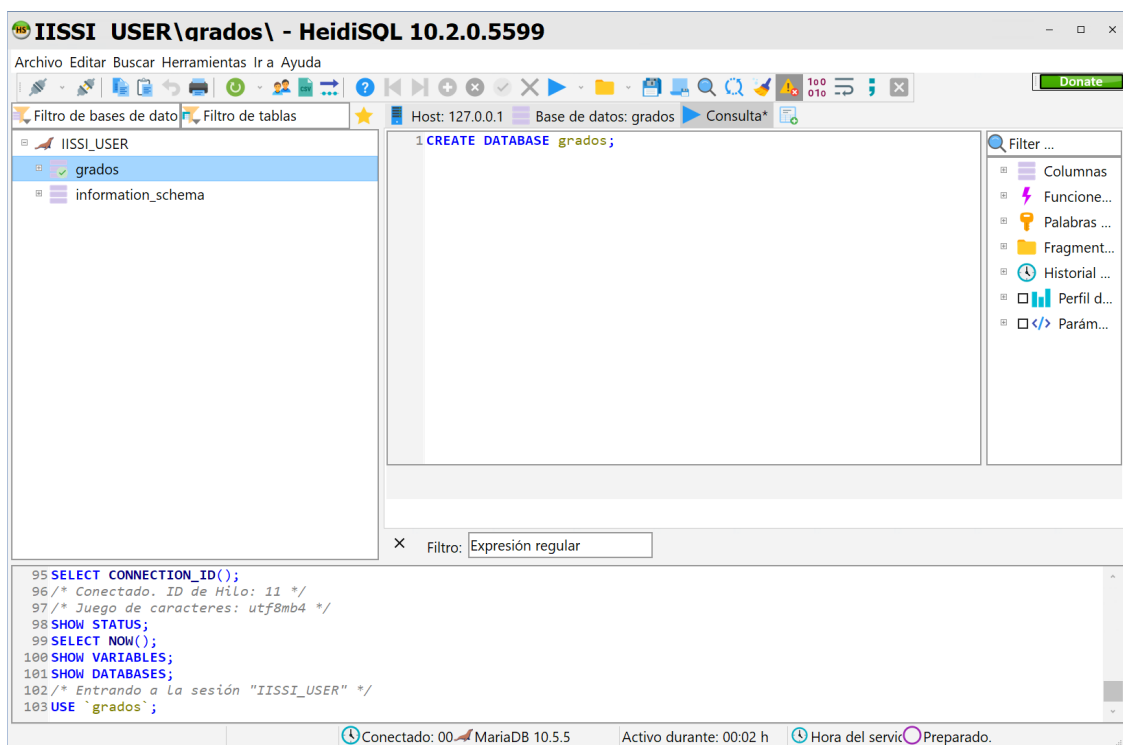
Finalmente, pulsaremos en "Guardar" para registrar el nuevo usuario.

## A.6. Conexión con el nuevo usuario


Crearemos una nueva conexión "IISSI\_USER" para el usuario que acabamos de crear, que será la que usaremos para trabajar con nuestras bases de datos:



En esta sesión sólo se tiene acceso a las BD del usuario, no a las del sistema:



## A.7. Ejecutar script de prueba

Para importar datos en la BD que acabamos de crear, utilizaremos el archivo [grados.sql](#). Seleccionamos. Archivo → “Cargar archivo SQL” → grados.sql (Ejecutar , “Enviar lote de una sola vez”). Durante su ejecución se pueden producir advertencias, pero no es algo inusual.

Al actualizar usando **F5**, podrá comprobar que se ha creado una nueva tabla "Asignaturas" en la BD "Grados".

Podemos ejecutar una consulta SQL para obtener el nombre, el número de créditos y el tipo de las asignaturas impartidas por el departamento "LENGUAJES Y SISTEMAS INFORMÁTICOS" usando la pestaña "Consulta":

```
SELECT nombre, creditos, tipo
FROM Asignaturas
WHERE departamento = "LENGUAJES Y SISTEMAS INFORMÁTICOS";
```

The screenshot shows the HeidiSQL interface. The SQL query is executed, and the results are displayed in a table with 3 rows and 3 columns: nombre, creditos, and tipo.

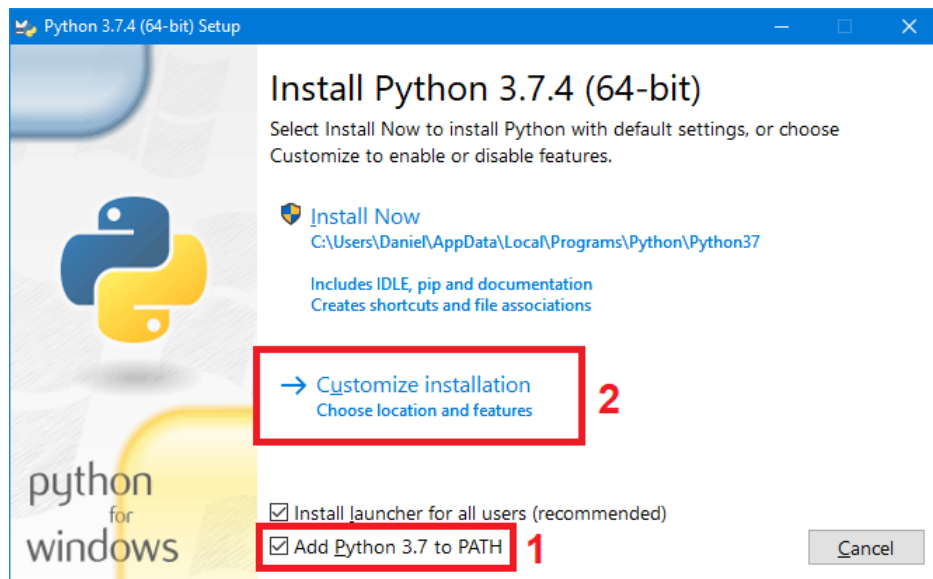
nombre	creditos	tipo
Fundamentos de Programación	12	Formación Básica
Análisis y Diseño de Datos y Algoritmos	12	Obligatoria
Sistemas Operativos	6	Obligatoria
Introducción a la Ingeniería del Software y los Sistemas de In...	6	Obligatoria
Introducción a la Ingeniería del Software y los Sistemas de In...	6	Obligatoria
Gestión de Sistemas de Información	6	Optativa
Procesadores de Lenguajes	6	Optativa
Sistemas de Información Empresariales	6	Optativa
Sistemas Orientados a Servicios	6	Optativa
Prácticas Externas	6	Optativa
Acceso Inteligente a la Información	6	Optativa
Gestión de Procesos y Servicios	6	Optativa
Interacción Persona-ordenador	6	Optativa
Seguridad en Sistemas Informáticos y en Internet	6	Optativa
Inteligencia Empresarial	6	Optativa
Modelado y Análisis de Requisitos en Sistemas de Información	6	Optativa

At the bottom, the status bar shows: 180 /\* Filas afectadas: 56 Filas encontradas: 0 Advertencias: 0 Duración para 18 consultas: 0,109 seg. \*/

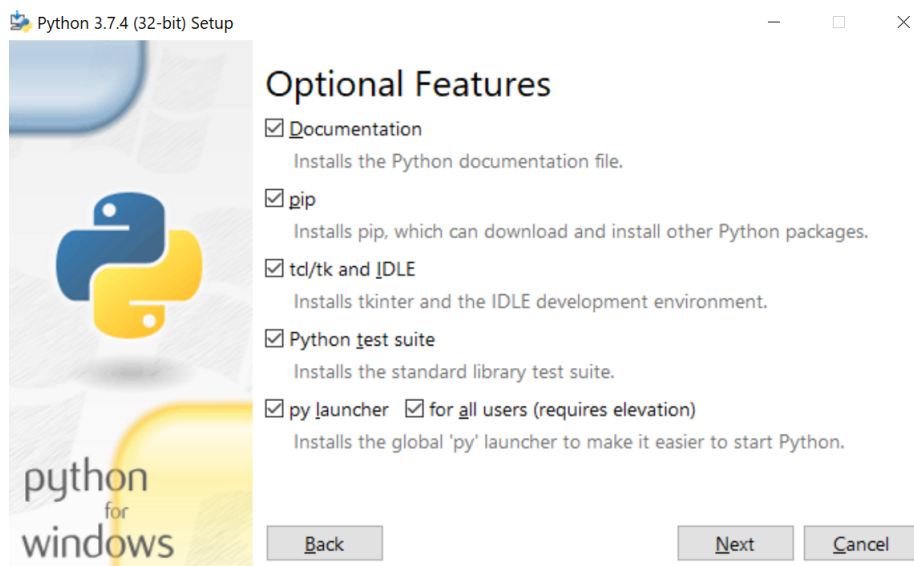
## A.8. Instalación y configuración de Python

**Nota:** si se tiene Python instalado mediante Anaconda, es posible que surjan problemas de compatibilidad. En ese caso, se recomienda desinstalar Anaconda antes de proceder.

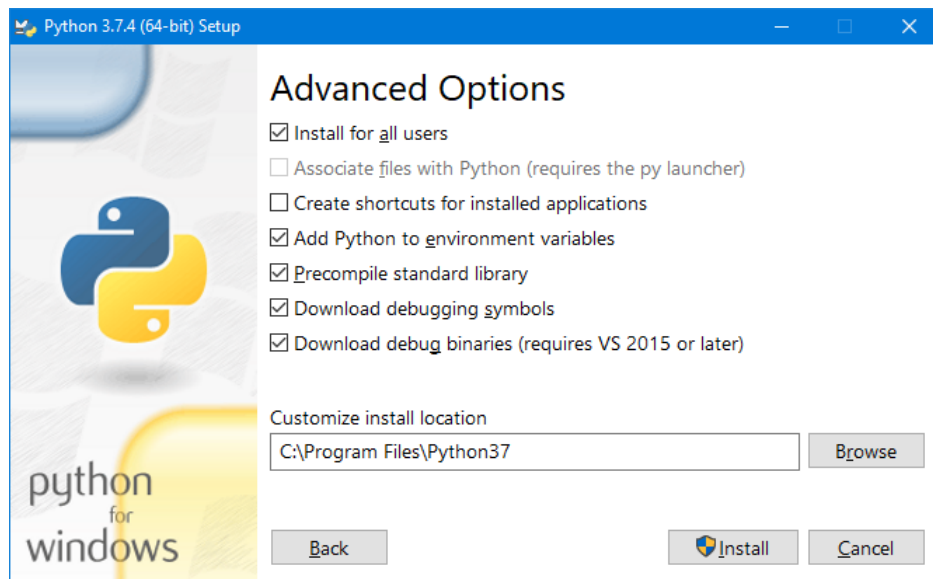
Python es necesario para usar el framework Silence, que se empleará al final de IISSI-1 y durante IISSI-2. Descargamos e instalamos la versión 3.X más reciente de [Python](#). Seleccionamos personalizar instalación ("customize installation"). **Es importante marcar antes la opción "Add Python 3.X to PATH":**



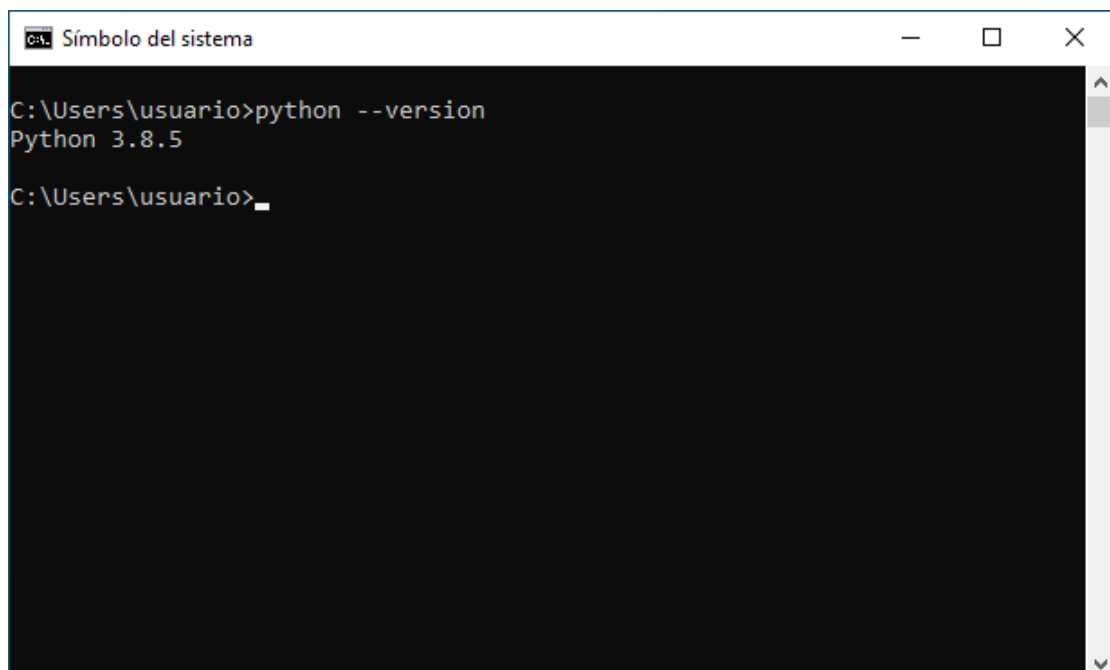
Entre las opciones de instalación opcionales, **dejamos marcadas todas las opciones:**



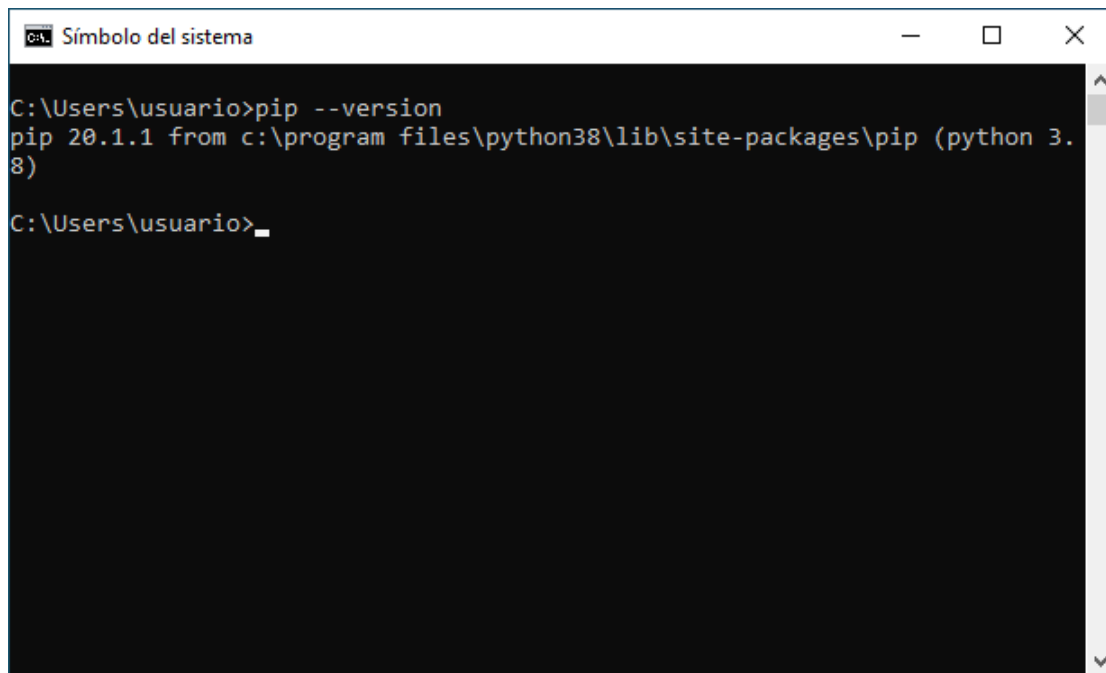
Marcamos las siguientes opciones avanzadas:



Una vez concluya la instalación, podemos comprobar que ésta ha sido correcta abriendo una consola y consultando la versión de Python instalada mediante `python --version`:



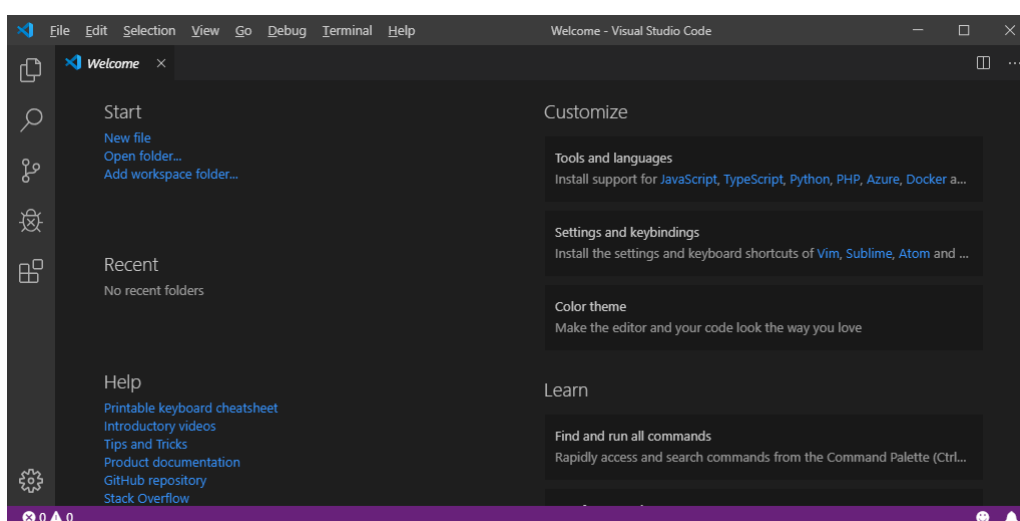
Igualmente, comprobaremos que pip, el gestor de paquetes de Python, está correctamente instalado, ya que lo necesitaremos más adelante. Para ello podemos ejecutar `pip --version`:



```
C:\Users\usuario>pip --version
pip 20.1.1 from c:\program files\python38\lib\site-packages\pip (python 3.8)
C:\Users\usuario>
```

## A.9. Instalación de Visual Studio Code

Como editor de código se usará [Visual Studio Code](#). Descargamos el instalador y lo ejecutamos. Mantenemos las opciones en sus valores por defecto e iniciamos Visual Studio Code una vez finalice la instalación:

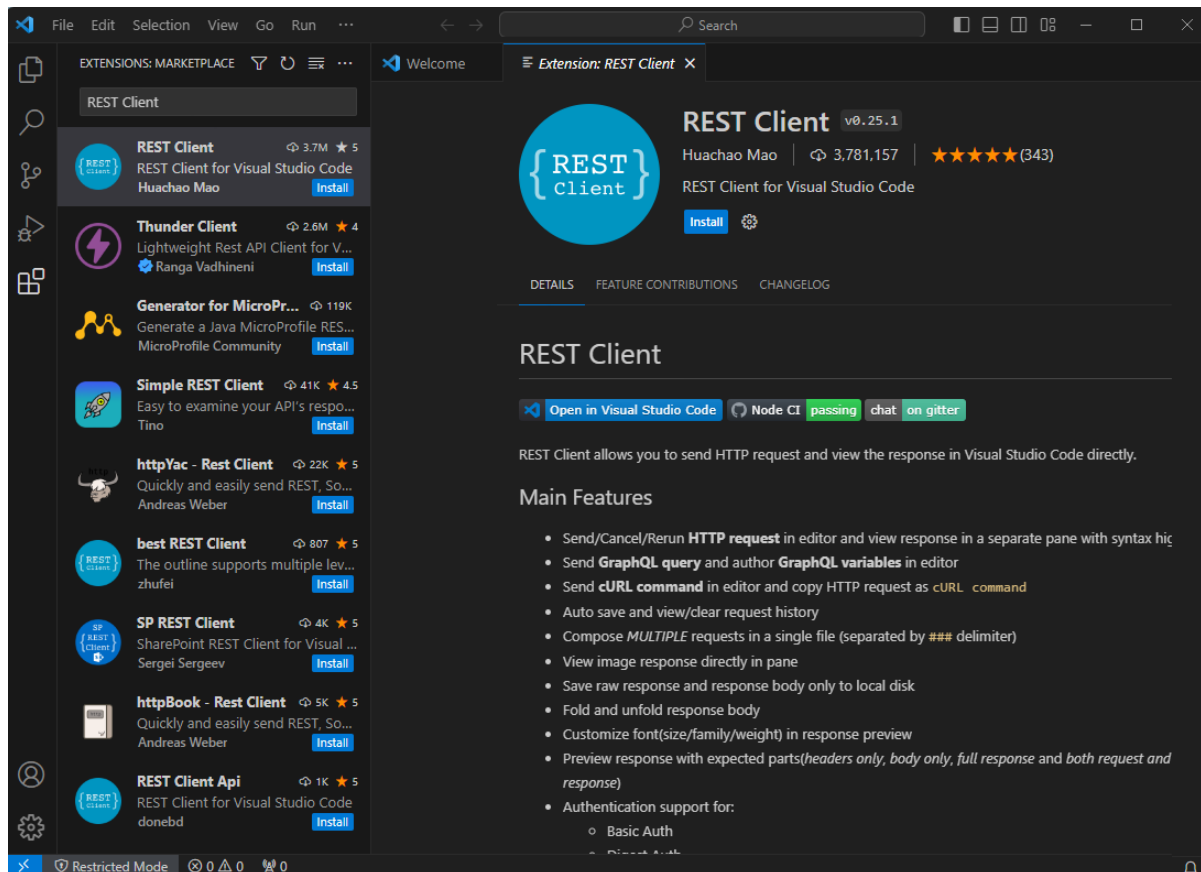


Accedemos a File → Preferences → Extensions, seleccionamos la [extensión de Python](#) y la instalamos:





Asimismo, instalaremos también la [extensión REST Client](#):



## A.10. Instalación de Git

Git es un sistema de control de versiones que usaremos a lo largo de IISSI-1 y 2 para descargar material relacionado con la asignatura, registrar nuestros cambios y mantener una copia de ellos en GitHub.

Para instalar Git, por favor, consulte el [boletín auxiliar de Git y GitHub](#) que está publicado.

Si desea consultar las nociones básicas sobre el funcionamiento de repositorios GitHub, en el boletín [Flujo de trabajo con GitHub](#) se describe cómo crear una cuenta y gestionar un repositorio. En este boletín se utiliza una versión de GitHub de la escuela, que puede encontrarse en <https://github.eii.us.es>