U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Information Systems - SINF 2020/2021

LAB GUIDE – SPRINT #2

Information Model & Database Management

Gil Gonçalves – gil@fe.up.pt
Luis Neto – lcneto@fe.up.pt
João Reis – jpcreis@fe.up.pt
Vítor Pinto – vitorpinto@fe.up.pt

1. Introduction & Context

The current project consists of the development of **G**reen **Man**ufacturing information system (GMAN) that addresses the issue of efficient energy in an Industry 4.0 scenario, which includes:

1. A multi-threaded software application for sensing/actuation using a Raspberry Pi 3.
2. A relational database in PostgreSQL for persistent storage of information.
3. A web interface, in order to present the information and accept commands of an end user.

Topic 1 was already covered in Sprint 1. At this point, in the Sprint 2 you will tackle topic 2, which introduce to students know how regarding information modelling, UML Class Diagram, database management and basic SQL.

Note that, in this Sprint, you will have dependencies regarding the development results of Sprint 1. In case you have failed to implement a functional C application, that is able to collect sensor readings and generate actuator commands based on control rules, a sample working version of a C application will be provided to you, in order to allow a successful completion of this Sprint goals. This working version will be available in the GitHub repository, were you will also find all the information to compile and run this application and all the necessary configuration files. This way you can have a stable version of the C application in order to move forward on your project.

This document focuses on the following aspects:

1. Introduction & Context
2. Database Introduction
3. PostgreSQL
4. SQL
5. Pre-requirements
6. C Application – Database Interaction
7. Sprint 2 Main Goals

2. Database Introduction

A **database (DB)** is an organized collection of data, generally stored and accessed electronically from a computer system. DBs can be used to support internal operations of organizations and to underpin online interactions with customers and suppliers. Also, DBs are used to hold administrative information and more specialized data, such as engineering data or economic models. Examples include computerized library systems, flight reservation systems, computerized parts inventory systems, and many content management systems that store websites as collections of webpages in a database. In a DB one should be able to: 1) have a collection with different types of data; 2) record relationships among different data items; 3) have varying sizes of data collections.

Typically, access to DB data is usually provided by a **database management system (DBMS)**, consisting of an integrated set of computer software that allows users to interact with one or more databases and provides access to all of the data contained in the database. Often the term "database" is also used to loosely refer to any of the DBMS, the database system or an application associated with the database. Physically, database servers are dedicated computers that hold the actual databases and run the DBMS and related software. With a DBMS, one should be able to: 1) insert new data, delete old data, and emend existing data in the collection; 2) retrieve data from the collection; 3) manage the collection so that it can be permanently store.

Databases and DBMSs can be categorized according to the database model(s) that they support, the type(s) of computer they run on, the query language(s) used to access the database, and their internal engineering, which affects performance, scalability, resilience, and security. A database model is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized and manipulated. The most popular example of a database model, and the one we will introduce in this course unit, is the **relational database management system (RDBMS**), which uses a table-based format, with columns and rows. Also, a query language is a computer language used to make queries in databases and information systems. A well-known query language and data manipulation language for relational databases, and the one we will introduce in this course unit, is the **Structured Query Language (SQL)**. In this course unit, we will introduce the **PostgreSQL**, often simply Postgres, as RDBMS, because of its ubiquitous support for the SQL standard among most relational databases.

Database Design & Modelling

In order to build a database, the first task focus on producing a conceptual information model that reflects the structure of the information to be held in the database. A common approach to this is to develop an **UML Class Diagram**, with the aid of drawing tools. In this course unit, the suggestion is to use the web tool draw.io (https://www.draw.io/). Figure 1 represents a UML Class Diagram example for a Medical Clinic Information System.

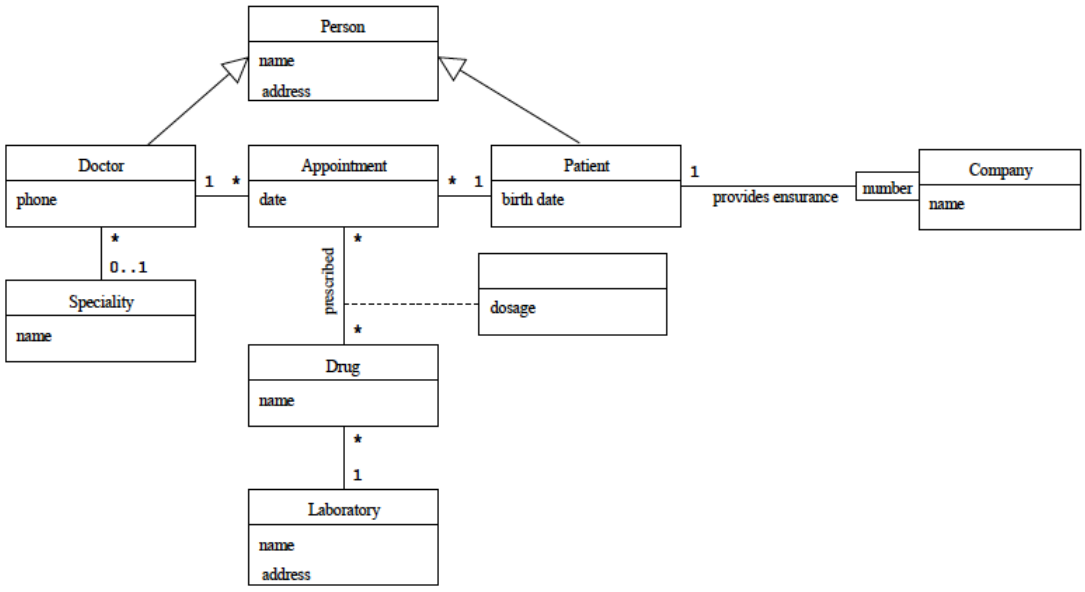


Figure 1 - UML Class Diagram example for a Medical Clinic Information System.

A successful information model will accurately reflect the possible state of the external world being modelled. Designing a good conceptual information model requires a good understanding of the application domain. The UML Class Diagram will help to develop an entity-relationship model, which establish definitions of the terminology used for entities, their relationships and attributes.

Having produced a conceptual information model, the next stage is to translate this into a schema that implements the relevant data structures within the database. This process is often called logical database design, and the output is a logical information model expressed in the form of a schema. The most popular database model for general-purpose databases, and the one we will introduce in this course unit, is the **relational model**, or more precisely, the relational model as represented by the SQL language. Figure 2 represents a relation model example for the Hospital Information System, based on the UML Class Diagram in Figure 1.

Speciality (cod, name)

Person (num, name, address)
Doctor (#num → Person, phone, #cod → Speciality)
Patient (#num → Person, birth_date)
Appointment (id, #num → Doctor [NN], #num → Patient [NN], date)

Laboratory (name, address)
Drug (ref, name, #lab → Laboratory [NN])
Prescribed (#id → Appointment, #ref → Drug, dosage) Company (name)

Ensurance (#num → Patient, #company → Company, number) {UK: company, number}

Figure 2 - Relation Model example for the Hospital Information System.

The final stage consists on defining and building the DB schema on PostgreSQL, using SQL, more specifically the Data Definition Language (DDL) (see more information in Section 4). In this case, DDL statements can be used to create, modify, and remove database objects such as tables, indexes, and users.
For further understanding, a modelling exercise is available on Moodle that focus in the development of UML Class Diagram and mapping the developed diagram into a relation model. Each team will be asked to practice the development of UML Class Diagrams and mapping into a relation model, by solving this exercise in the first two LP classes in this Sprint.

3. PostgreSQL

PostgreSQL is a free and open source RDBMS with an emphasis on extensibility and standards compliance. It can handle workloads ranging from small single-machine applications to large Internet-facing applications (or for data warehousing) with many concurrent users. PostgreSQL claims high conformance with the SQL standard.

PostgreSQL is available for several operating systems. In this course unit, we will use PostgreSQL as a hosting service provided by CICA at FEUP, which already have PostgreSQL installed and available to be used in different scenario applications. Also, there are several open sources front-ends and tools for administering PostgreSQL. Regarding the PostgreSQL hosting service provided by CICA at FEUP, we will use the web tool **phpPgAdmin**, as represented in Figure 3. phpPgAdmin is a web-based administration tool for PostgreSQL written in PHP, which allow users to manage and manipulate data, such as browse tables, views and reports, execute arbitrary SQL and use the select, insert, update and delete commands over available data. One can also dump table data in several formats and import SQL scripts.

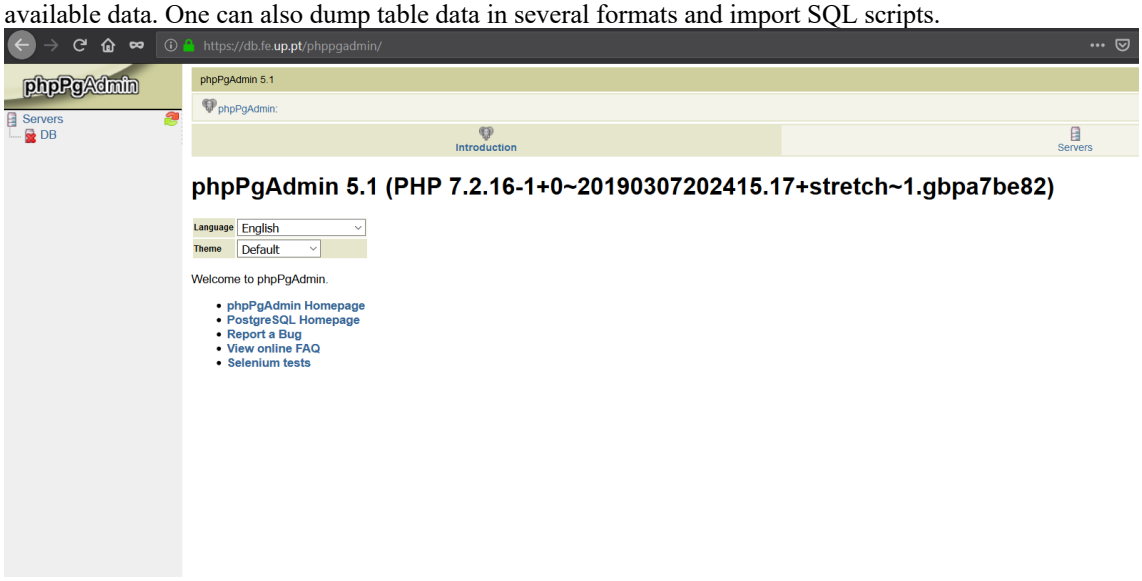


Figure 3 - phpPgAdmin web tool.

In this course unit, teachers already requested the creation of PostgreSQL database accounts for each team. You can access to this service by using a client app or a web tool such as phpPgAdmin. Regarding a client app, you should build an app, using a specific programming language, which includes libraries to connect to PostgreSQL databases (see more information in Section **C Application – Database Interaction**).

In order to access the created PostgreSQL database account via phpPgAdmin:

- Access phpPgAdmin using the url https://db.fe.up.pt/phppgadmin/ .
- Click on *DB* under *Servers* and introduce the credentials to login the *DB* (credentials will be provided soon by the teacher). You will see an interface like the one represented in Figure 4. Note that you must be connect to FEUP network eduroam or connect to the eduroam via VPN (see more information in Section **Pre-requirements**).

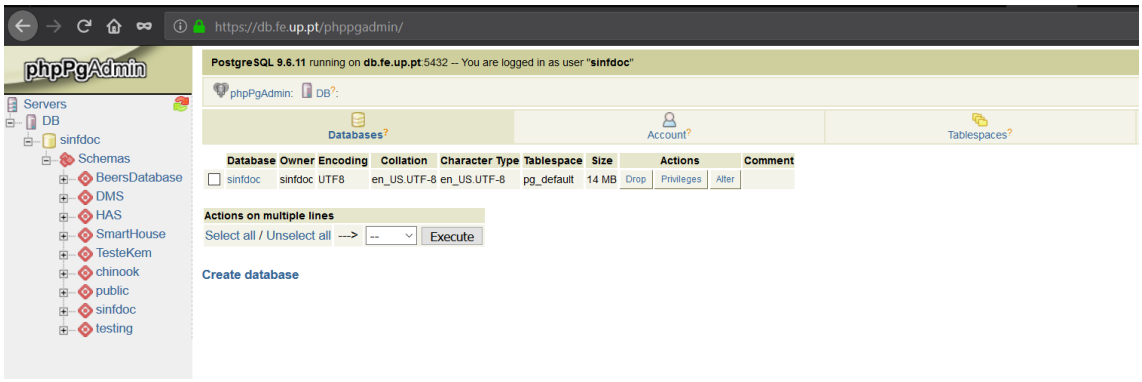


Figure 4 - PostgreSQL Management using phpPgAdmin.

Managing Databases with phpPgAdmin

With phpPgAdmin one can administer multiple databases and multiple schemas. All the available DB schemas are available under the tab *Schemas*. Also, if you click on that tab, it should open a page like Figure 5. Possibly, at the beginning, only the *public* schema is available, since it is a default schema, containing several tables and relationships among them.

Figure 5 - phpPgAdmin - Viewing a Database.

In each schema, you have available 3 actions:

- *Drop* – Remove a schema from the database, by automatically drop objects (tables, functions, etc.) that are contained in the schema.
- *Privileges* – Define access privileges on a database object to one or more roles of users.
- *Alter* – Alter a schema will change the definition of a schema, regarding the name of the schema and new owner.

Users also have other options, as seen in the tabs of Figure 5, such as:

- *SQL* - Use of the SQL language in PostgreSQL, where users can insert SQL commands supported by PostgreSQL.
- *Find* - Searching for database objects.
- *Variables* – List of configuration parameters that affect the behaviour of the database system.

- *Processes* – Tool for monitoring database activity and analysing performance, via a PostgreSQL's statistics collector.
- *Locks* - Provides access to information about the locks held by open transactions within the database server.
- *Admin* – Several administrative tasks are available to be applied on the whole database, such as: 1) *VACUUM* - garbage-collect and optionally analyse a database; 2) *ANALYZE* - collect statistics about a database; 3) *CLUSTER* - cluster a table according to an index; 4) *REINDEX* - rebuild indexes.
- *Privileges* - For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges can be granted in this tab.
- *Export* – Exporting a database.

Back to the *Schema* tab, you can open a specific schema, where you can find more options for working with tables, functions, views, domains, sequences and others. In the *Tables* tab, as represented in the example in Figure 6, you can view and/or create the database structures that will hold your data.

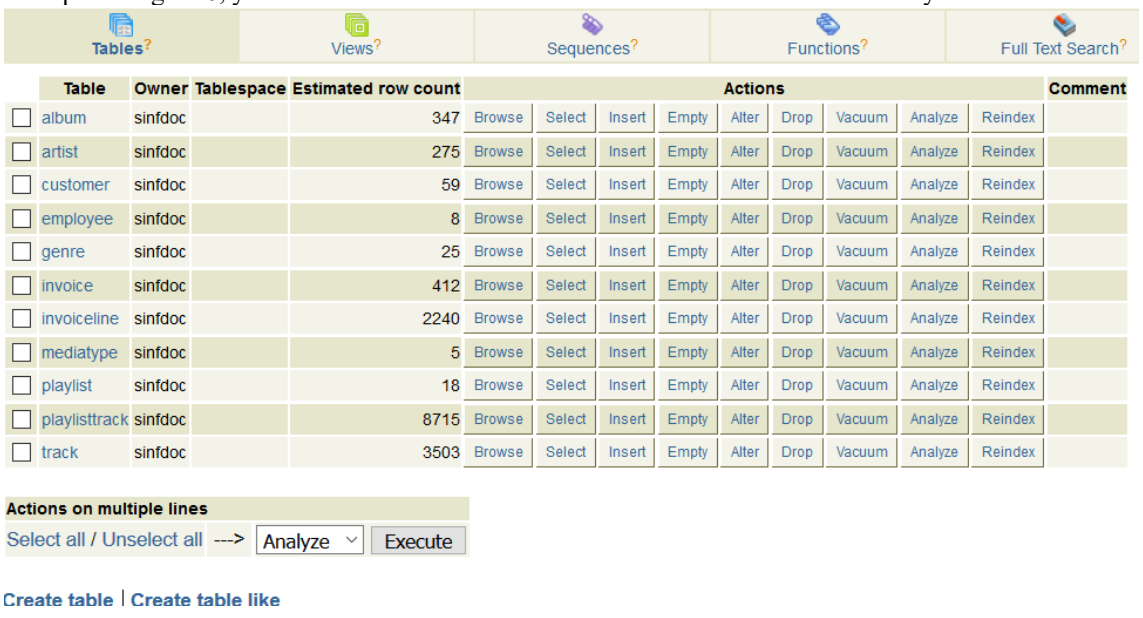


Figure 6 - phpPgAdmin - Viewing a DB Schema Table Structure Example.

Since in a relational database, the raw data is stored in tables, creating a database structure consists in create and modify tables, and explore available features in each table, to control what data is stored in the tables. You can create a new table by clicking in *Create table*, where you must specify the name of the table, the number of columns and, for each column, define the field/attribute, by specifying the field name, type of data, length available for the data, and data constraints, such the acceptance of null values, or if the field represents a unique or primary keys, as represented in Figure 7.

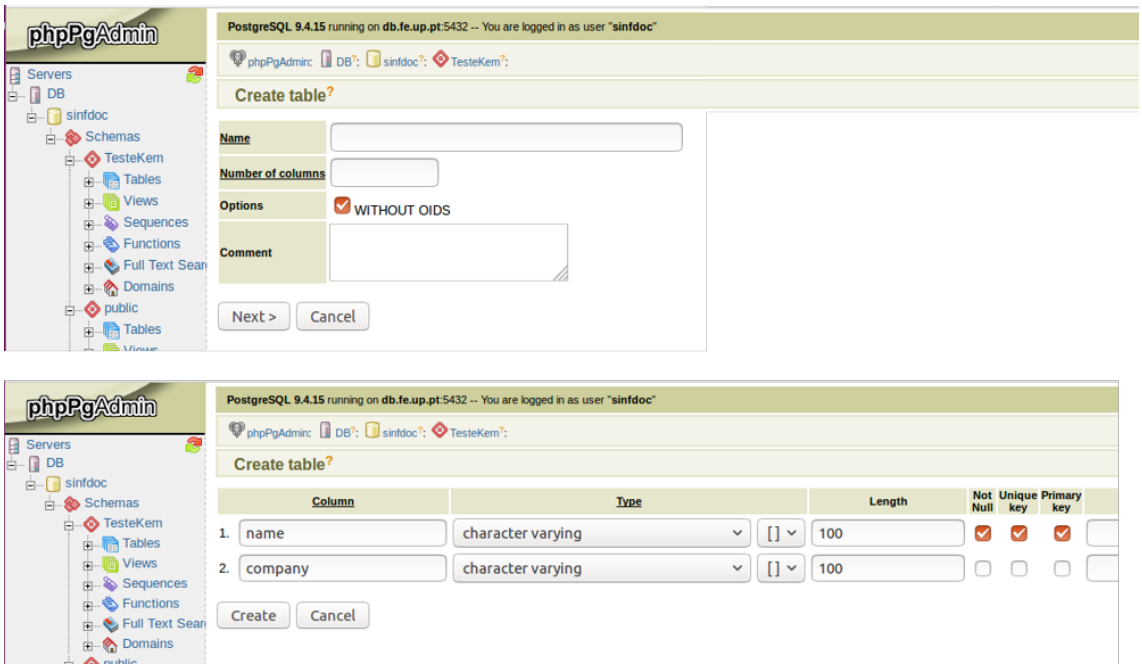


Figure 7 - phpPgAdmin - Create Table.

Also, by clicking in the name of an existing table, you can see the table structure, as represented in the example in Figure 8. You can see all the information that is also requested in a table creation, such as column name, type of data, not null, unique and primary key constraints.

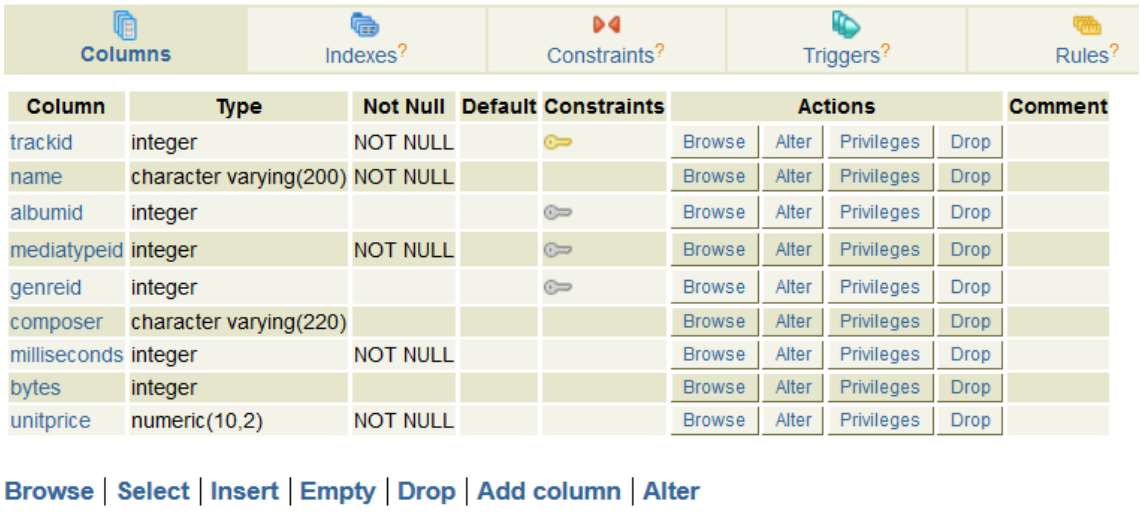
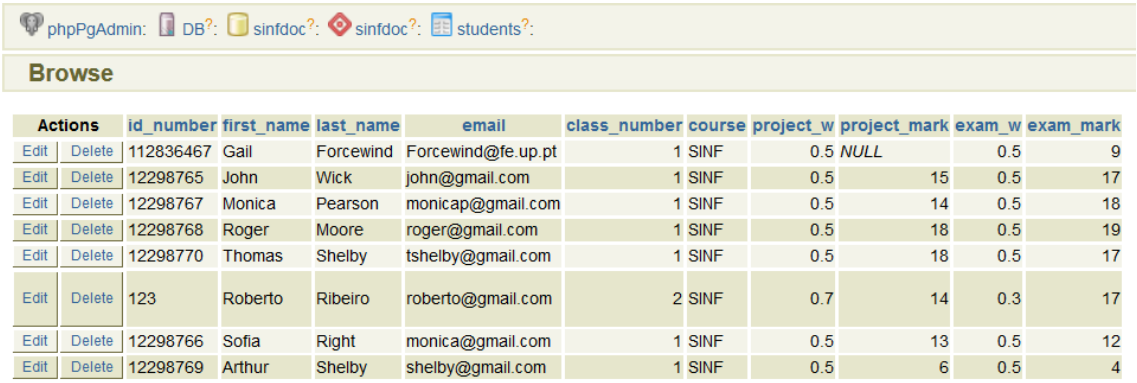


Figure 8 - phpPgAdmin - Viewing a Table Structure.

Regarding the features for each DB Schema Table Structure (see Figure 6) they are:

- *Browse* – View the data that is contained in a given table. Basically, data in a table is organized in rows, where each row is a new tuple that was inserted in the table. Regarding columns, each one corresponds a given table field or attribute of the inserted tuple. A table example, which contains student data, is represented in Figure 9.



phpPgAdmin: DB?: sinfdoc?: sinfdoc?: students?:

Browse

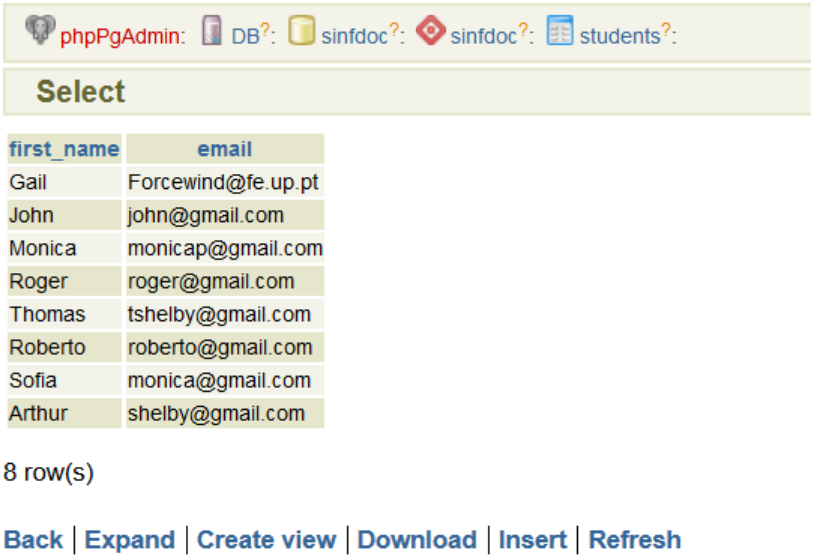
Actions		id_number	first_name	last_name	email	class_number	course	project_w	project_mark	exam_w	exam_mark
Edit	Delete	112836467	Gail	Forcewind	Forcewind@fe.up.pt	1	SINF	0.5	*NULL*	0.5	9
Edit	Delete	12298765	John	Wick	john@gmail.com	1	SINF	0.5	15	0.5	17
Edit	Delete	12298767	Monica	Pearson	monicap@gmail.com	1	SINF	0.5	14	0.5	18
Edit	Delete	12298768	Roger	Moore	roger@gmail.com	1	SINF	0.5	18	0.5	19
Edit	Delete	12298770	Thomas	Shelby	tshelby@gmail.com	1	SINF	0.5	18	0.5	17
Edit	Delete	123	Roberto	Ribeiro	roberto@gmail.com	2	SINF	0.7	14	0.3	17
Edit	Delete	12298766	Sofia	Right	monica@gmail.com	1	SINF	0.5	13	0.5	12
Edit	Delete	12298769	Arthur	Shelby	shelby@gmail.com	1	SINF	0.5	6	0.5	4

8 row(s)

Back | Expand | Insert | Refresh

Figure 9 - phpPgAdmin - Table Browse.

- *Select* - Retrieves rows from the table. You can specify which attributes (table fields) should be selected and specify constrains, via operators, to data, in order to filter rows out. Figure 10 represents a select of the browsed table *students*, where only the *first_name* and *email* attributes were considered.



phpPgAdmin: DB?: sinfdoc?: sinfdoc?: students?:

Select

first_name	email
Gail	Forcewind@fe.up.pt
John	john@gmail.com
Monica	monicap@gmail.com
Roger	roger@gmail.com
Thomas	tshelby@gmail.com
Roberto	roberto@gmail.com
Sofia	monica@gmail.com
Arthur	shelby@gmail.com

8 row(s)

Back | Expand | Create view | Download | Insert | Refresh

Figure 10 - phpPgAdmin - Table Select.

- *Insert* - Create new rows in a table. You can insert one or more rows, where in each table field you can specify a value or a *Null* expression, as represented in Figure 11. Figure 12 represents the data of the table *student* after a new row is inserted, where the fields *first_name*, *last_name* and *email* were attributed with values and the rest of the fields are *Null*.

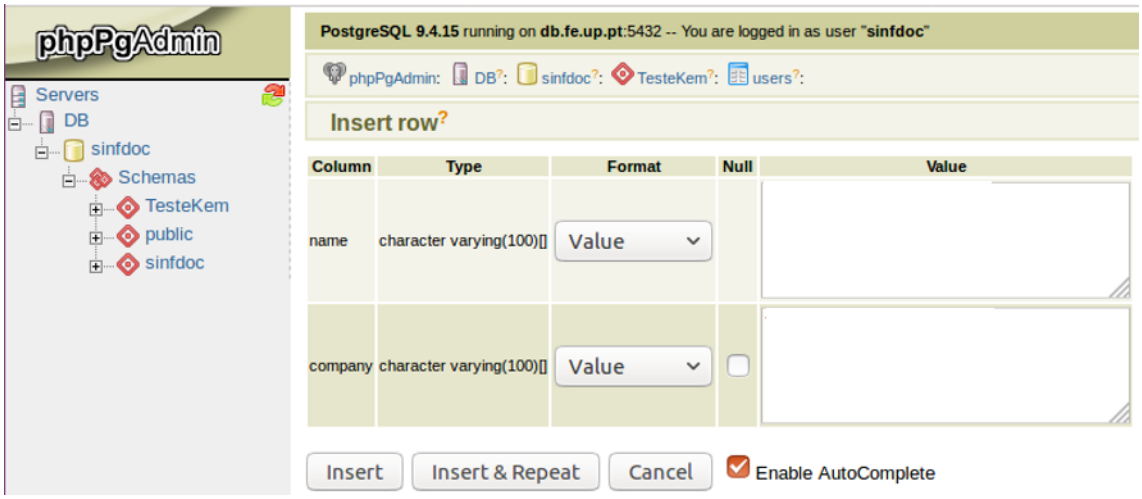


Figure 11 - phpPgAdmin - Insert Row in Table.

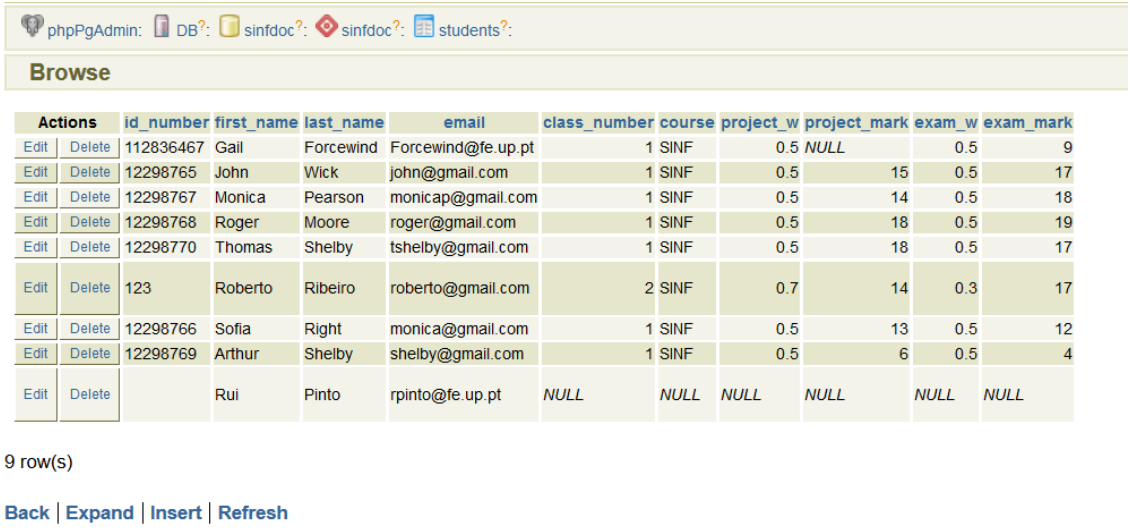


Figure 12 - phpPgAdmin - Table Browse After Insert.

- *Empty* - Quickly removes all rows from the table, leaving the table with no data.
- *Alter* - Change the definition of an existing table, namely the name of the table and the schema that the table belongs.
- *Drop* - Removes the table from the database. On contrary of the *Empty* option, dropping a table will eliminate the table, instead of just empty its data.
- *Vacuum*, *Analyze* and *Reindex* are basically the same as described earlier, but specific for a table instead of an entire database.

4. SQL

SQL is a standard domain-specific language used in programming and designed for managing data held in a RDBMS. It is particularly useful in handling structured data where there are relations between different entities/variables of the data. SQL introduced the concept of accessing many records with one single command and it eliminates the need to specify how to reach a record. SQL consists of many types of statements, which may be informally classed as sublanguages, namely:

- Data query language (DQL), which allows searching for information and computing derived information;
- Data definition language (DDL), which defines data types such as creating, altering, or dropping and the relationships among them;
- Data control language (DCL), which controls access to data;
- Data manipulation language (DML), which performs tasks such as inserting, updating, or deleting data occurrences.

In the next sections, there will be detailed the DDL and DML. Note that, using the phpPgAdmin, you have always available a tab for using the SQL language in PostgreSQL. In this tab you can insert several SQL commands from the four sublanguages. You can find a *SQL* tab at the top right corner of the page, alongside with the *History*, *Find* and *Logout* tabs.

Data Definition Language

A DDL is a syntax of SQL for defining data structures, especially database schemas. DDL statements are used to create, modify, and remove database objects such as tables, indexes, and users. Common DDL statements are *CREATE*, *ALTER*, and *DROP*. Basically, several functionalities available in phpPgAdmin, which were presented in Section **PostgreSQL** can be performed using DDL. In this Sprint you will be required to use DDL to implement your database schema.

Table Basics

DDL statements to define tables consists on basic commands, such as *CREATE* and *DROP*. While the *CREATE* command is used to create tables, the *DROP* command is used to delete tables.

The basic structure of a table creation statement in SQL is represented in Figure 13, where values between <> are to be replaced.

```
CREATE TABLE <table_name> (
  <column_name> <data_type>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Figure 13 - SQL CREATE Command.

On the other side, to delete a table, we can use the SQL statement represented in Figure 14. Note that if there are foreign keys referencing the table, we must use the cascade keyword.

```
DROP TABLE <table_name>;
```

```
DROP TABLE <table_name> CASCADE;
```

Figure 14 - SQL DROP Command.

Data Types

PostgreSQL has a rich set of native data types available to users when using SQL. The most important ones are: *NUMERIC*, *DATE/TIME*, *TEXT*, *BOOLEAN* and *SERIAL*.

- *NUMERIC(precision, scale)* stores numbers with a very large number of digits. The scale of a numeric data type is the count of decimal digits in the fractional part, to the right of the decimal point. The precision of a numeric is the total count of significant digits in the whole number. On the other side, they are the data types *SMALLINT*, *INTEGER* and *BIGINT* store whole numbers. Also, the data types *REAL* and *DOUBLE* are inexact, variable-precision numeric types. Figure 15 represents the numeric precision for numeric data types.

SMALLINT	2 bytes	-32768 to +32767
INTEGER	4 bytes	-2147483648 to +2147483647
BIGINT	8 bytes	-9223372036854775808 to +9223372036854775807
NUMERIC	user-specified	numeric(3,1) -99.9 to 99.9
REAL	4 bytes	inexact 6 decimal digits precision
DOUBLE	8 bytes	inexact 15 decimal digits precision

Figure 15 - Precision for numeric data types in PostgreSQL.

- *DATE/TIME* stores date, time or both in the ISO 8601 format. *DATE/TIME* can also store timezone information. The type *DATE* stores date values, e.g., “1980-12-25”. On the other side, the type *TIME* stores time values, e.g., “04:05” or “04:05:06.789”. Finally, the type *TIMESTAMP* stores date and time, e.g., “1980-12-25 04:05:06.789”.
- *TEXT* stores text format variables. The type *CHARACTER VARYING(n)*, or *VARCHAR(n),* stores variable-length text with a user defined limit. On the other side, the type *CHARACTER(n),* or *CHAR(n),* stores fixed-length, blank-padded text. Finally, the type *TEXT* stores variable unlimited length text. Note that *VARCHAR*, without a limit, is the same as *TEXT*. Also, *CHAR*, without a limit, is the same as *CHAR(1)*.
- *BOOLEAN* type can only have two possible values: true or false. Possible values representing true: TRUE, 't', 'true', 'y', 'yes', 'on' and '1'. On the other side, possible values representing false: FALSE, 'f', 'false', 'n', 'no', 'off' and '0'.

- *SERIAL* is a pseudo-type that can be used to define auto-generated identifiers or auto-counters in PostgreSQL.

Figure 16 represent an example of the usage of several data types in a table creation statement.

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2),
  taxes NUMERIC(6,2),
  card_number INTEGER,
  active BOOLEAN
);
```

Figure 16 - SQL CREATE Command Example with Data Types.

Defaults

Default values are used, in a table creation statement, for initialize one or more columns of the table with a given value. Figure 17 represents an example of the usage of default values in a table creation statement. In this case, the column referring to the card number has default value of 0, while employee is active by default.

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0,
  active BOOLEAN DEFAULT TRUE
);
```

Figure 17 - SQL CREATE Command Example with Default Values.

Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number. To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition. The most

usual types of constraints used in SQL are: *CHECK*, *NOT NULL*, *UNIQUE KEY*, *PRIMARY KEY* and *FOREIGN KEY*.

- CHECK

The *CHECK* constraint allows to define a constraint on the column values using an expression that the values must follow and can be used as represented in Figure 18. Also, Figure 19 represents an example of the *CHECK* constraint usage in a table creation statement, where, in this example, the salary of the employee must be larger than 500 and the card number must be larger or equal to 0. Finally, if the check constraint refers to more than one column, we must use a table-based constraint, as represented in Figure 20. In this case, taxes must be lower than salary. Note that it was given a name to the constraint, in this case "*taxes_lower_salary*", in order to allows us to better identify it when errors occur or when we want to refer to it.

```
CREATE TABLE <table_name> (
  <column_name> <data_type> CHECK <check_expression>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Figure 18 - SQL CHECK Constraint.

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2) CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE
);
```

Figure 19 - SQL CHECK Constraint Example.

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 20 - SQL CHECK Constraint Example in a Multiple Column Check.

- NOT NULL

The *NOT NULL* constraint can be used to define that a certain column does not allow NULL values, as represented in Figure 21. In the example represented in Figure 22, the employee name cannot be null.

```
CREATE TABLE <table_name> (
  <column_name> <data_type> NOT NULL,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Figure 21 - SQL NOT NULL Constraint.

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 22 - SQL NOT NULL Constraint Example.

- PRIMARY KEY

A *PRIMARY KEY* constraint indicates that a column, or group of columns, can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null. We can define one, and only one, primary key for the table, as represented in Figure 23. In the example represented in Figure 24, the employee id is used as primary key, meaning than it cannot be null and cannot have repeated values.

```
CREATE TABLE <table_name> (
  <column_name> <data_type> PRIMARY KEY,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Figure 23 – SQL PRIMARY KEY Constraint.

```sql
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 24 – SQL PRIMARY KEY Constraint Example.

There can be situation where a primary key is composed by more than one column. In these cases, we must use a table-based constraint, as represented in Figure 25. Also, Figure 26 represents a usage example, where an employee cannot have the same phone number twice.

```sql
CREATE TABLE <table_name> (
  <column_name> <data_type>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>,
  PRIMARY KEY (<column_name>, <column_name>)
);
```

Figure 25 - SQL Multiple Column PRIMARY KEY.

```sql
CREATE TABLE telephone (
  employee INTEGER,
  phone VARCHAR,
  PRIMARY KEY (employee, phone)
);
```

Figure 26 - SQL Multiple Column PRIMARY KEY Example.

- UNIQUE KEY

UNIQUE constraints ensure that the data contained in a column, or a group of columns, is unique among all the rows in the table. Unique keys are identical to primary keys, but they allow null values and there can be multiple unique keys in one table. Also, they can be created using the same type of syntax used in primary keys. Figure 27 represents a usage example, where card keys cannot have repeated values. Also, there can be situation where a unique key is composed by more than one column. In these cases, we must use a table-based constraint, as represented in Figure 28.

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 27 - SQL UNIQUE Constraint Example

```
CREATE TABLE <table_name> (
  <column_name> <data_type>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>,
  UNIQUE (<column_name>, <column_name>)
);
```

Figure 28 - SQL Multiple Column UNIQUE Example.

- FOREIGN KEY

A foreign key constraint (*REFERENCES* keyword) specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table, as represented in Figure 29. We say this maintains the referential integrity between two related tables, meaning that a foreign key must always reference a key (primary or unique) from another (or the same) table. Also, databases don't allow columns with a foreign key containing values that do not exist in the referenced column. Figure 30 represents a usage example, where the id of the department references the id column in the department table. In this case, employees cannot have a department number that doesn't exist in the department table.

```
CREATE TABLE <table_A> (
  <column_A> <data_type> PRIMARY KEY,
  <column_B> <data_type>,
  ...
  <column_C> <data_type>
);

CREATE TABLE <table_B> (
  <column_X> <data_type> PRIMARY KEY,
  <column_Y> <data_type>,
  ...
  <column_Z> <data_type> REFERENCES <table_A>(<column_A>)
);
```

Figure 29 - SQL FOREIGN KEY (REFERENCES) Constraint.

```sql
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  department_id INTEGER REFERENCES department(id),
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 30 - SQL FOREIGN KEY (REFERENCES) Constraint Example.

Also, if the referenced column is the primary key of the other table, we can omit the name of the column, as represented in Figure 31. Finally, if the referenced table has a key with multiple columns, we must use a table-based constraint to define our foreign key, as represented in Figure 32.

```sql
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  department_id INTEGER REFERENCES department,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 31 - SQL FOREIGN KEY to PRIMARY KEY.

```sql
CREATE TABLE telephone (
  employee INTEGER,
  phone VARCHAR,
  PRIMARY KEY (employee, phone)
);

CREATE TABLE call (
  id INTEGER PRIMARY KEY,
  employee INTEGER,
  phone INTEGER,
  when DATE,
  caller INTEGER,
  FOREIGN KEY (employee, phone) REFERENCES telephone (employee, phone)
);
```

Figure 32 - SQL Multiple Column FOREIGN KEY.

Declaring a foreign key means that values in one table must also appear in the referenced column. Deleting and updating referenced values, three different things can occur: An error is thrown; The referencing values becomes *NULL*; All referencing values are updated to the new value (this might cause a cascade effect). To define the desired behaviour, we should use the *ON DELETE* and *ON UPDATE* clauses, as represented in Figure 33, with one of three possible values: *RESTRICT* (throws an error); *SET NULL* (values become null); and *CASCADE* (lines are deletes or values updated). An example is represented in Figure 34. In this case, if a department with id 1 is deleted, all employees with department id equal to 1 will start having a null department number. Also, if a department with id 1 is updated to id 2, all employees with department id equal to 1 will start having a department number equal to 2.

```
CREATE TABLE <table_A> (
  <column_A> <data_type> PRIMARY KEY,
  <column_B> <data_type>,
  ...
  <column_C> <data_type>
);

CREATE TABLE <table_B> (
  <column_X> <data_type> PRIMARY KEY,
  <column_Y> <data_type>,
  ...
  <column_Z> <data_type> REFERENCES <table_A>(<column_A>)
    ON DELETE SET NULL ON UPDATE CASCADE
);
```

Figure 33 - SQL ON DELETE and ON UPDATE Clauses.

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  department_id INTEGER REFERENCES department
    ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

Figure 34 - SQL ON DELETE and ON UPDATE Clauses Example.

Data Manipulation Language

A DML is a syntax of SQL for retrieving and manipulating data in a relational database such as adding (inserting), deleting, and modifying (updating) data in a database. Also, query statements for selecting data

in a database can also be considered DML. In this Sprint you will be required to use DML to manipulate the data in the database schema created early using DDL.

Inserting

To insert values (a new row) into a table we use the *INSERT* command, as represented in Figure 35. Figure 36 represents an example of using the *INSERT* command, by inserting a new row on the table *employee*.

```
INSERT INTO <tablename> (<col1>, <col2>, ...) VALUES (<val1>, <val2>, ...);
```

Figure 35 - SQL INSERT Command.

```
INSERT INTO employee (id, name, salary) VALUES (1, 'John Doe', 1000);
```

Figure 36 - SQL INSERT Command Example.

We can omit the column names if we insert the values in the same order we used to create the table columns, as represented in Figure 37.

```
INSERT INTO employee VALUES (1, 'John Doe', 1000);
```

Figure 37 - SQL INSERT Command Example with Column Names Omitted.

Deleting

To delete values (rows) from a table, we use the *DELETE* command, as represented in Figure 38. This command can receive a condition specifying which rows to delete. If no condition is given, all rows are deleted from the table.

```
DELETE FROM <tablename> WHERE <condition>;
```

Figure 38 - SQL DELETE Command.

Figure 39 represent 3 examples of the *DELETE* command usage, where in the first example, an employee with id=1 is deleted, in the second example, all employees are deleted, and in the third example, employees with salary larger or equal to 1200 are deleted from the *employee* table.

```
DELETE FROM employee WHERE id = 1;
```

```
DELETE FROM employee;
```

```
DELETE FROM employee WHERE salary >= 1200;
```

Figure 39 - SQL DELETE Command Examples.

Updating

To modify values (rows) from a table, we use the *UPDATE* command, as represented in Figure 40. The update command can receive a condition specifying which rows to update. If no condition is given all rows are updated from the table.

```
UPDATE <tablename> SET <col1> = <val1>, <col2> = <val2>, ... WHERE <condition>;
```

Figure 40 - SQL UPDATE Command.

Figure 41 represent 4 examples of the *UPDATE* command usage, where in the first example, the salary of the employee with id=1 is changed to 1300, in the second example, the salary of the employee with id=1 is increased by 10%, in the third example, the salary of all employees is increases by 10%, and in the fourth example, the salary and taxes of all employees with a salary larger than 1200 are decreased by 10%.

```
UPDATE employee SET salary = 1300 WHERE id = 1;
```

```
UPDATE employee SET salary = salary * 1.1 WHERE id = 1;
```

```
UPDATE employee SET salary = salary * 1.1;
```

```
UPDATE employee SET salary = salary * 0.9, taxes = taxes * 0.9
WHERE salary > 1200;
```

Figure 41 - SQL UPDATE Command Examples.

Selecting

The most basic SQL statement used for database querying is the *SELECT* and *FROM*, which retrieves rows from tables. They allow to specify which tables (*FROM*) and columns (*SELECT*) we want to retrieve from the database. The result of an SQL query is also a table, as represented in Figure 42.

```
SELECT * FROM employee;
```

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3
5	Richard Roe	900	0	NULL

Figure 42 - SQL SELECT … FROM Command Example & Result.

Note that we can use an * to select all columns from the table. You can also choose the columns to be selected, as represented in Figure 43. In this case, only the *id* and *name* columns from table employee are selected.

```
SELECT id, name FROM employee;
```

id	name
1	John Doe
2	Jane Doe
3	John Smith
4	Jane Roe
5	Richard Roe

Figure 43 - SQL SELECT … FROM Command Example & Result with Column Choosing.

Also, you can perform any operations between columns, as represented in Figure 44. In this case, it is selected columns *id*, *name* and the difference between salary and taxes from table employee. Note that it was used the *AS* operator to rename the columns. In this case, it was renamed the columns *id* as *num* and the difference between salary and taxes to *net_salary*.

```
SELECT id AS num, name, salary - taxes AS net_salary FROM employee;
```

num	name	net_salary
1	John Doe	800
2	Jane Doe	700
3	John Smith	850
4	Jane Roe	800
5	Richard Roe	900

Figure 44 - SQL SELECT … FROM Command Example & Result with Column Operations.

Filter Rows

In order to filter rows, you can use the *WHERE* command, which allows us to filter which rows we want in our result table according to a condition. The condition can use any comparison operator (<, >, <=, <>, ...) and can be composed using *AND*, *OR* and *NOT*. An example is represented in Figure 45, where, in this case, only employees from department 2 or a salary lower or equal to 900 are selected.

```
SELECT * FROM employee WHERE dep_num = 2 OR salary <= 900;
```

id	name	salary	taxes	dep_num
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
5	Richard Roe	900	0	NULL

Figure 45 - SQL SELECT ... FROM ... WHERE Command Example & Result with Row Filtering.

To test if a value is null, we must use the special *IS NULL* operator, as represented in Figure 46. On the other hand, we can use the *IS NOT NULL* operator to select rows where a certain attribute is not null.

```
SELECT * FROM employee WHERE dep_num IS NULL;
```

id	name	salary	taxes	dep_num
5	Richard Roe	900	0	NULL

Figure 46 - SQL IS NULL Operator.

We can also remove duplicates from the final result by using the *DISTINCT* operator, as represented in Figure 47. In this case, the query selects the different salaries in the database.

```
SELECT DISTINCT salary FROM employee;
```

salary
1000
800
1200
900

Figure 47 - SQL DISTINCT Operator.

Aggregate Functions

There are five special aggregate functions defined in the SQL language, as represented in Figure 48.

```
MIN, MAX, SUM, AVG and COUNT
```

value
1
2
NULL
2
3

```
SELECT MIN(value)              FROM table; -- 1
SELECT MAX(value)              FROM table; -- 3
SELECT SUM(value)              FROM table; -- 8
SELECT AVG(value)              FROM table; -- 2
SELECT COUNT(value)            FROM table; -- 4 (counts non null values)
SELECT COUNT(DISTINCT value)   FROM table; -- 3 (counts distinct non null values)
SELECT COUNT(*)                FROM table; -- 5 (counts lines)
```

Figure 48 - SQL Aggregate Functions MIN, MAX, SUM, AVG and COUNT.

The operator *GROUP BY* groups the rows into sets based on the value of a specific column or columns, as represented in Figure 49. In this case, all rows with same value in the *num* column get grouped together.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT AVG(salary) FROM employees GROUP BY num;
```

AVG(salary)
700
1150
1000

Figure 49 - SQL Operator Group By.

Grouped rows can also be filtered using the *HAVING* clause, as represented in Figure 50.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT num, d_name, AVG(salary) FROM employees
GROUP BY num, d_name HAVING AVG(salary) >= 1000;
```

d_name	AVG(salary)
Sales	1150
Production	1000

Figure 50 - SQL HAVING Clause.

Sorting Rows

The order in which rows are sorted in a query result is unpredictable. You can sort the end result using the *ORDER BY* clause. By default, values are sorted in ascending order. We can change the default order using the *ASC* and *DESC* clauses, as represented in Figure 51.

```
SELECT * FROM employees ORDER BY salary ASC; -- default
```

id	name	salary	taxes	num	d_name
1	John Doe	**700**	200	1	Marketing
2	Jane Doe	**800**	100	2	Sales
4	Jane Roe	**1000**	200	3	Production
3	John Smith	**1500**	350	2	Sales

```
SELECT * FROM employees ORDER BY salary DESC;
```

id	name	salary	taxes	num	d_name
3	John Smith	**1500**	350	2	Sales
4	Jane Roe	**1000**	200	3	Production
2	Jane Doe	**800**	100	2	Sales
1	John Doe	**700**	200	1	Marketing

Figure 51 - SQL ORDER BY ... ASC/DESC Clause.

Joining Tables

In a situation where we have multiple tables, the cartesian product allows us to combine rows from different tables. To use it, we just must indicate which tables we want to combine using commas to separate them. The result is a table containing the columns of all tables and all possible combinations of rows, also known as *CROSS JOIN*, as represented in Figure 52.

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3
5	Richard Roe	900	0	NULL

num	name
1	Marketing
2	Sales
3	Production

```
SELECT * FROM employee, department;
```

```
SELECT * FROM employee CROSS JOIN department;
```

id	name	salary	taxes	dep_num	num	name
1	John Doe	1000	200	1	1	Marketing
2	Jane Doe	800	100	2	1	Marketing
3	John Smith	1200	350	2	1	Marketing
4	Jane Roe	1000	200	3	1	Marketing
5	Richard Roe	900	0	NULL	1	Marketing
1	John Doe	1000	200	1	2	Sales
2	Jane Doe	800	100	2	2	Sales
3	John Smith	1200	350	2	2	Sales
4	Jane Roe	1000	200	3	2	Sales
5	Richard Roe	900	0	NULL	2	Sales
1	John Doe	1000	200	1	3	Production
2	Jane Doe	800	100	2	3	Production
3	John Smith	1200	350	2	3	Production
4	Jane Roe	1000	200	3	3	Production
5	Richard Roe	900	0	NULL	3	Production

Figure 52 - SQL CROSS JOIN.

When selecting from more than one table, columns with the same name might lead to ambiguities. To solve them we must use the table name before the column name, as represented in Figure 53.

```
SELECT id, employee.name, department.name FROM employee, department;
```

id	name	name
1	John Doe	Marketing
2	Jane Doe	Marketing
3	John Smith	Marketing
4	Jane Roe	Marketing
5	Richard Roe	Marketing
1	John Doe	Sales
2	Jane Doe	Sales
3	John Smith	Sales
4	Jane Roe	Sales
5	Richard Roe	Sales
1	John Doe	Production
2	Jane Doe	Production
3	John Smith	Production
4	Jane Roe	Production
5	Richard Roe	Production

Figure 53 - SQL Solving Ambiguities.

You can also join tables using the *WHERE* keyword. Instead of using a cartesian product followed by the *WHERE* keyword, we can use the more specific keywords: *JOIN ON*, as represented in Figure 54. These keywords allow us to specify simultaneously which tables to join and with which joining condition. These are also called *inner joins*.

```
SELECT * FROM employee, department WHERE dep_num = num;
```

```
SELECT * FROM employee JOIN department ON dep_num = num;
```

id	name	salary	taxes	dep_num	num	name
1	John Doe	1000	200	1	1	Marketing
2	Jane Doe	800	100	2	2	Sales
3	John Smith	1200	350	2	2	Sales
4	Jane Roe	1000	200	3	3	Production

Figure 54 - SQL JOIN ON.

If the columns used in the join operation have the same name, we can join them using in a simpler way with *JOIN USING*, as represented in Figure 55.

id	name	salary	taxes	num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3
5	Richard Roe	900	0	NULL

num	name
1	Marketing
2	Sales
3	Production

```
SELECT * FROM employee JOIN department USING(num);
```

id	name	salary	taxes	num	name
1	John Doe	1000	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1200	350	2	Sales
4	Jane Roe	1000	200	3	Production

Figure 55 - SQL JOIN USING.

Sometimes, when joining tables, some of the rows are left out as they do not match any rows on the other table. If we want these rows to be present in the result, we must use what is called and *outer join*. There are three types of outer joins: 1) *LEFT* - Rows on the left table that do not match any row in the right table are kept; *RIGHT* - Rows on the right table that do not match any row in the left table are kept; *FULL* - Rows on any of the tables that do not match any row in the other table are kept.

Pre-requirements

To access PostgreSQL, you must be connected to FEUP network eduroam or connect to the eduroam via VPN. So, in the next section, you will find how can you configure a VPN connection to FEUP network. In https://sigarra.up.pt/feup/pt/web_base.gera_pagina?p_pagina=p%c3%a1gina%20est%c3%a1tica%20gen%c3%a9rica%2063 you can see several methods to use the FEUP's VPN service.

If you are working in the virtual machine, if the configurations are correct, it is probably sharing the same network adapter as the host operating system. So, you may configure a VPN connection in your host operating system, in order to have the same VPN connection available in the virtual machine's operating system. As an example for Win10, go to *Network Management -> VPN -> Add new VPN connection*. Now input the requested data, as presented in Figure 56 (you should use your own user name and password).

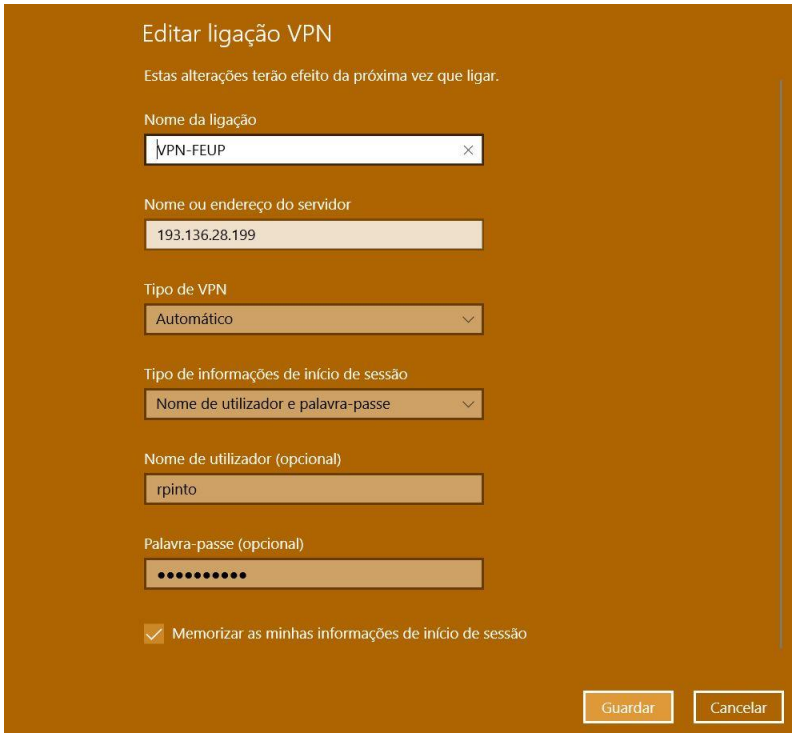


Figure 56 - FEUP's VPN Configuration on Win10.

(Only for 50% of the Grade)

If you are working with the Raspberry Pi 3, running the image provided earlier in Moodle, you can access to the eduroam network via Wi-Fi, following these steps:

1. Open the Terminal and execute the command "sudo nano /etc/wpa_supplicant/wpa_supplicant.conf";
2. Insert your email and password in the "*identity*" and "*password*" fields of the "*network*" parameter, as represented in Figure 57;

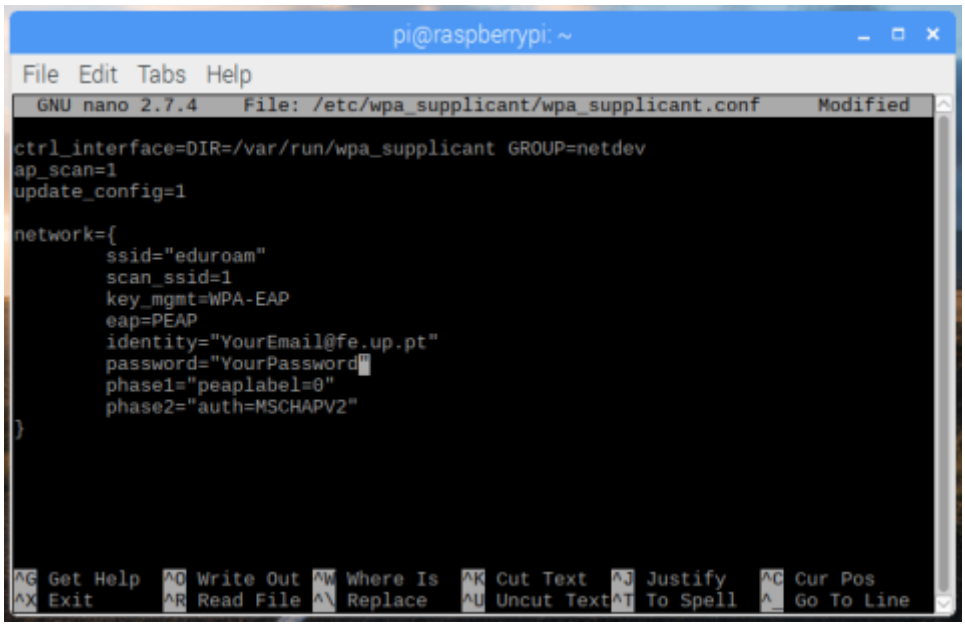


Figure 57 - Raspberry Pi - Wi-Fi Network Configuration.

3. Press “CTRL+X”, then “Y” and finally “Enter”;
4. Still in the Terminal, execute the command “*sudo nano /etc/network/interfaces*”;
5. Remove the “*#*” before “*auto wlan0*”as presented in Figure 58;

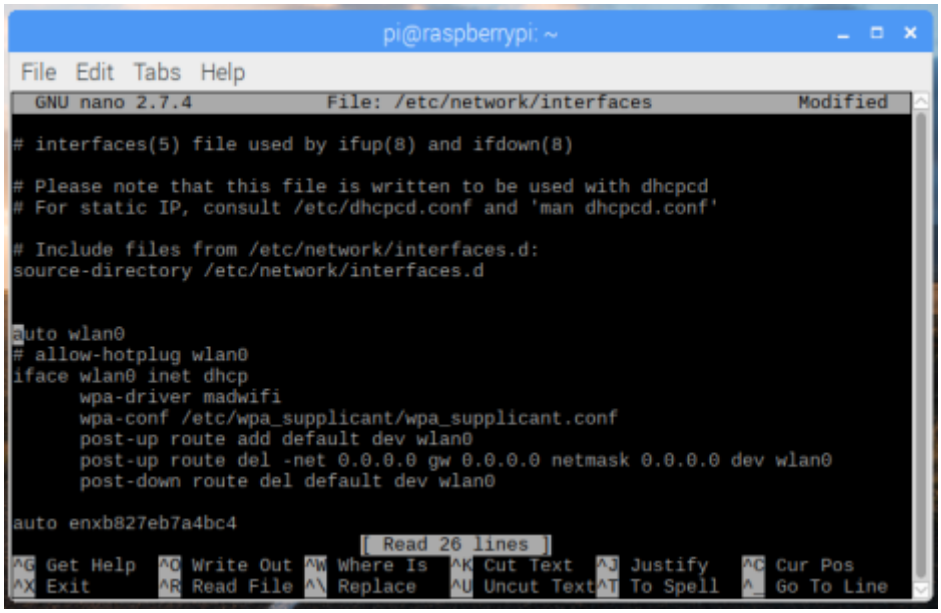


Figure 58 - Raspberry Pi - Allow Searching for a Wi-Fi Network.

6. Reboot the Raspberry Pi.

Now you have full access to the Raspberry Pi environment and internet connection.

C Application – Database Interaction

PostgreSQL has several interfaces available and is also widely supported among programming language libraries. One of the most used built-in interfaces in C coding language is the **libpq**, PostgreSQL's official C application interface.

libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries. In this Sprint, after database design and implementation, it is required that you adapt your C application, developed in Sprint 1, by including an interface to connect to the PostgreSQL earlier implemented, such as libpq. In you C application, you must use libpq to perform DML statements, in order to save data into the database. This data may be available sensors and actuators, sensor/actuator configuration in the house, control rules configuration, sensor data readings, actuator state, among others.

Client programs that use libpq must include the header file *libpq-fe.h* and must link with the libpq library. So, if not available, you should download and install libpq in the operating system that will run the C application (the virtual machine or the Raspberry Pi). For this, open the terminal and insert the following commands:

- sudo apt-get update
    a. Note that you may be required to introduce the password for root permissions. The password is “raspberry”.
- sudo apt-get install libpq-dev
- sudo apt-get install postgresql
- pg_config –includedir
    a. You should confirm that the location of C header files of the client interface is “*/usr/include/postgresql*”.

Now that the *libpq-fe.h* file is available to be included in the C application, you should edit your code accordingly. Figure 59 represents a simple example of a PostgreSQL C client program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char **argv)
{
    PGconn     *conn; //Database connection object
    PGresult   *res; //Query database object
    const char *dbconn; //Connection string object

    /* Make a connection to the database */
    dbconn = "host = 'db.fe.up.pt' dbname = 'dbname' user = 'user' password = 'password'";
    conn = PQconnectdb(dbconn);

    /* Check to see that the backend connection was successfully made & execute a query */
    if (!conn) {
      printf(stderr, "libpq error: PQconnectdb returned NULL.\n\n");
      PQfinish(conn);
      exit(1);
    }
    else if (PQstatus(conn) != CONNECTION_OK){
      printf(stderr, "Connection to database failed: %s", PQerrorMessage(conn));
      PQfinish(conn);
      exit(1);
    }
    else {
      printf("Connection OK\n");
      res = PQexec(conn, "INSERT INTO employee (id, name, salary) VALUES (1, 'Jonh Doe', 1000)");
      PQfinish(conn);
    }

    return 0;
}
```

Figure 59 - Sample of a PostgreSQL C client program.

Connecting to the database server is done via the *PQconnectdb()* function, with the configuration being passed as a string. In this case, you should specify the *host*, which is always "*db.fe.up.pt*", *dbname* and *user*, which will probably be the same, and *password*. User and password will be provided to you by the teacher. To check if the connection to the database was successful, you need to test the status of the connection, via the *PQstatus()* function. If not, print out the error for better debugging. If the connection was ok, then you can execute SQL statements, via the *PQexec()* function, by passing as parameters the connection object and the query string.

Finally, when you are finished, it is good practice to close the connection, via the *PQfinish()* function, in order to avoid memory leaks.

Note that, in order to compile you C application successfully, you should include additional arguments. In this case, you should include the -lpq argument, as represented in Figure 60.

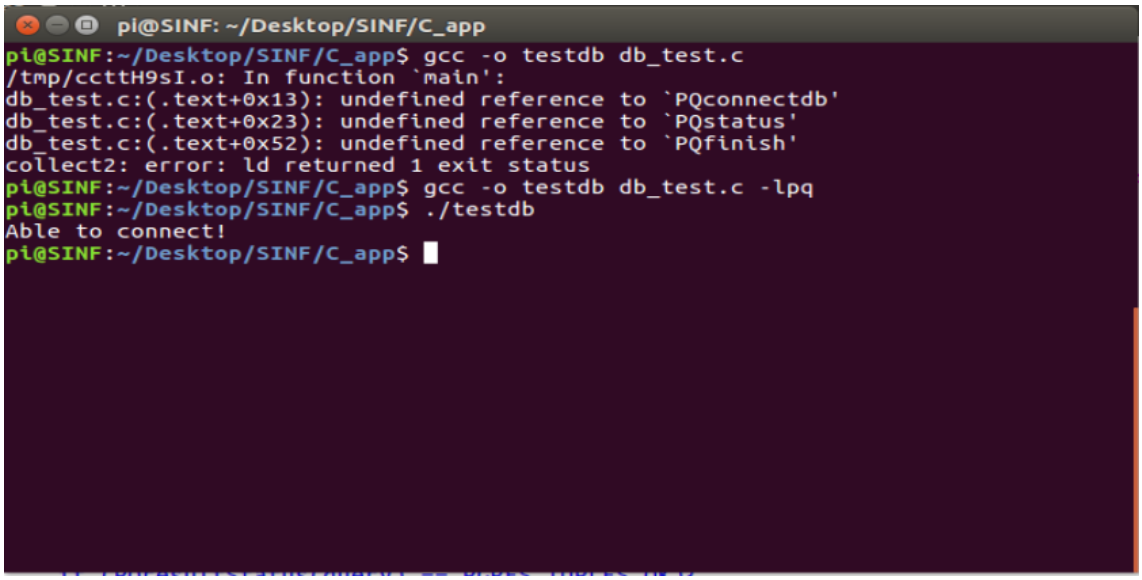


Figure 60 - Compile C code using -lpq arguments.

Sprint 2 Main Goals

According to the Projects Dashboard, the main goals for Sprint 2 are represented in Figure 61.

Attendance Grade	Goal	Appreciation
6 points	Information model (UML Class Diagram)	10%
	Project Planning (Sprint Backlog)	5%
	DB Implementation (Relational model + DDL)	8%
	Get configuration and rules (DML)	8%
	Store sensor information and actuator state (DML)	5%
8 points	Store system configuration (rooms, sensors and actuators) from C Application	
	Store all rules from C Application	
10 points	Sensors can be moved inside the house; the application needs to keep a record of the places where the sensor has been operating and the corresponding period	
	Different rules can be active only in certain periods of the day (e.g. rule 1 from 9:00 to 14:00, and rule 2 from 14:00 to 18:00)	

Figure 61 - Project's Dashboard Goals.

All goals are related to several technical tasks (except the project planning goal). Next, more detail is given regarding the technical tasks of each goal.

1) **Information Model** - Develop the information model regarding your system implementation. For this, you should design the conceptual information model, by developing an UML Class Diagram. You can use the draw.io web tool. You can see more information in Section 2.a**Database Design & Modelling**.
2) **DB Implementation** - Develop the logical database design, based on the UML Class Diagram. For this, you should map the UML Class Diagram to a relation model. Having the relation model, next you should define and build the DB schema on PostgreSQL based on the relation model. For this, you should use DDL in the web tool phpPgAdmin. You can see more information in Sections **Database Design & Modelling**, 1.a**Managing Databases with phpPgAdmin** and **Data Definition Language**.
3) **Get Configurations and Rules** - Populate your PostgreSQL database with information regarding sensor/actuator configuration, house configuration and control rules. For this, you can choose one of three options: manually input this information in the phpPgAdmin, use DML in the phpPgAdmin or use DML statements in your C application. Later, you should retrieve the stored

information in your C application using DML. You can see more information in Sections **Managing Databases with phpPgAdmin** and **Data Manipulation Language**.

4) **Store Sensor Information and Actuator State** - Connect your C application with the PostgreSQL database. For this you should edit you C code, in order to implement DML statements. These statements can be used to insert in the database sensor/actuator information, such as sensor measurements and actuator state. You can see more information in Sections **Data Manipulation Language** and **C Application – Database Interaction**.
5) **Store System Configuration & all Rules from C Application** - Connect your C application with the PostgreSQL database and, based on DML statements, store into the database information such as sensor/actuator configuration, house configuration, control rules and others, such as user information. You can see more information in Sections **Data Manipulation Language** and **C Application – Database Interaction**.
6) **Keep Record of Sensor Location & Rule Activation Periods** – Adapt the structure and populate your PostgreSQL database with information regarding sensor location and activation periods of rules, based on DDL and DML. Assume that sensor motes can be in different locations at different periods of time. Also, assume that control rules can be activated only in some periods of time. Connect your C application with the PostgreSQL database and, based on DML statements, you should be able to retrieve the history of relations between sensors, locations, control rules and periods of time. You can see more information in Sections **Data Definition Language**, **Data Manipulation Language** and **C Application – Database Interaction**