

TUTORIAL 5 - Advancing Programming

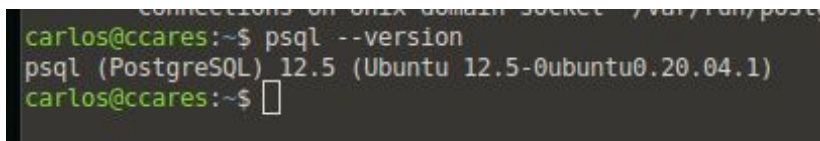
Database Connection in a Razor .NET Project

The way of answering is by reporting the results in the personnel chat using the usual way of pasting the screenshots, which must include part of your desktop background, and, at the same time, it must include the time of the screenshot. For those problems, including coding, you must also paste the images of the corresponding new code. Identify each answer with the corresponding label.

Part-1 (25%). Console access to a remote database

Step-1. Verifying psql installation

Install psql in your operating system. It is a client-side Postgres interface. In order to verify that it is running execute the following command: `psql --version`. In the case of Windows you should run CMD first (console).



```
carlos@ccares:~$ psql --version
psql (PostgreSQL) 12.5 (Ubuntu 12.5-0ubuntu0.20.04.1)
carlos@ccares:~$
```

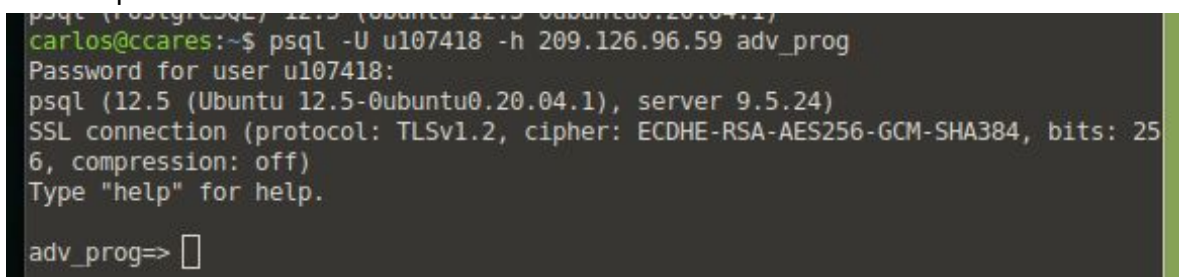
If you have other, maybe older version, do not worry, any version later than 9 will work ok.

Step-2. Connection to your database schema

The teacher has provided you, **by email**, with a Postgres account to the database `adv_prog`. You have your university number, for example, 107418, then you have the user identification `u107418` and you have the schema `s107418`. To connect to your database schema you should execute:

```
psql -U u<yourWSB_id> -h 209.126.96.59 adv_prog
```

for example:



```
psql (PostgreSQL) 12.5 (Ubuntu 12.5-0ubuntu0.20.04.1)
carlos@ccares:~$ psql -U u107418 -h 209.126.96.59 adv_prog
Password for user u107418:
psql (12.5 (Ubuntu 12.5-0ubuntu0.20.04.1), server 9.5.24)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

adv_prog=>
```

You must be sure that your firewall or the institutional firewalls are not blocking port 5432 which is the default port for Postgres. If you are at home you should not have any problem because Internet providers normally do not block ports. If you have a local installation of Postgres then you will be a collision. In this case, stop your own Postgres server or change its serving port. (file hba_conf.conf)

Step-2. Creating a table in your schema

Here we will create a table in your schema. However, this table will need a special data type that we will first create by giving the following command in the previous connection.

```
CREATE TYPE Task_Status AS ENUM  
('Running', 'Delivering', 'Revising', 'Accepted', 'Cancelled');
```

The result should be as the following lines (the command is executed only when, after the Enter key, there is a “;”. If the “;” is not found, then the database manager assumes that the command is incomplete and will wait for the rest of the command).

```
adv_prog=> CREATE TYPE Task_Status AS ENUM  
adv_prog-> ('Running', 'Delivering', 'Revising', 'Accepted', 'Cancelled');  
CREATE TYPE  
adv_prog=> █
```

After this, we will create our table using the following SQL command:

```
CREATE TABLE task (  
    task_id serial PRIMARY KEY,  
    taskname VARCHAR ( 200 ) NOT NULL,  
    supervisor VARCHAR ( 60 ) NOT NULL,  
    email_supervisor VARCHAR ( 255 ) NOT NULL,  
    responsible VARCHAR (60) NOT NULL,  
    email_responsible VARCHAR ( 255 ) NOT NULL,  
    created_on TIMESTAMP NOT NULL,  
    deadline TIMESTAMP NOT NULL,  
    first_delivering TIMESTAMP,  
    first_revising TIMESTAMP,  
    estimate_hours int NOT NULL,  
    effective_hours int,  
    status Task_Status NOT NULL  
);
```

Step-3. Inserting data by using psql

Use the following sentence for inserting a new record in the table `task`.

```
INSERT INTO task(taskname,
    supervisor,
    email_supervisor,
    responsible,
    email_responsible,
    created_on,
    deadline,
    estimate_hours,status)
VALUES ('sort method receiving array with alphanumeric key',
    'Robert Pollini',
    'rpol@highcompany.pl',
    'Carol Stern',
    'cstern@highcompany.cl',
    NOW(),
    '2020-12-24 14:00:00',
    14,
    'Running') Returning task_id;
```

For verifying the insertion execute the following command:

```
select task_id,responsible,created_on,deadline,status
from task;
```

The way of delivering this part is sending the following items having their corresponding labels.

Tut-5-P1-A. Screenshot of the version of your psql

Tut-5-P1-B. Screenshot of the output of the “select * from task;” command in your psql console

Tut-5-P1-C. Screenshot of the output of the “\d” command in your psql console

Comment: you can, in addition to the psql console, install **pgAdmin**, which is also a client-side Postgres manager having a GUI, it exists for Macintosh, Linux, and Windows operative systems.

Part-2 (25%). Reading a Postgres Database

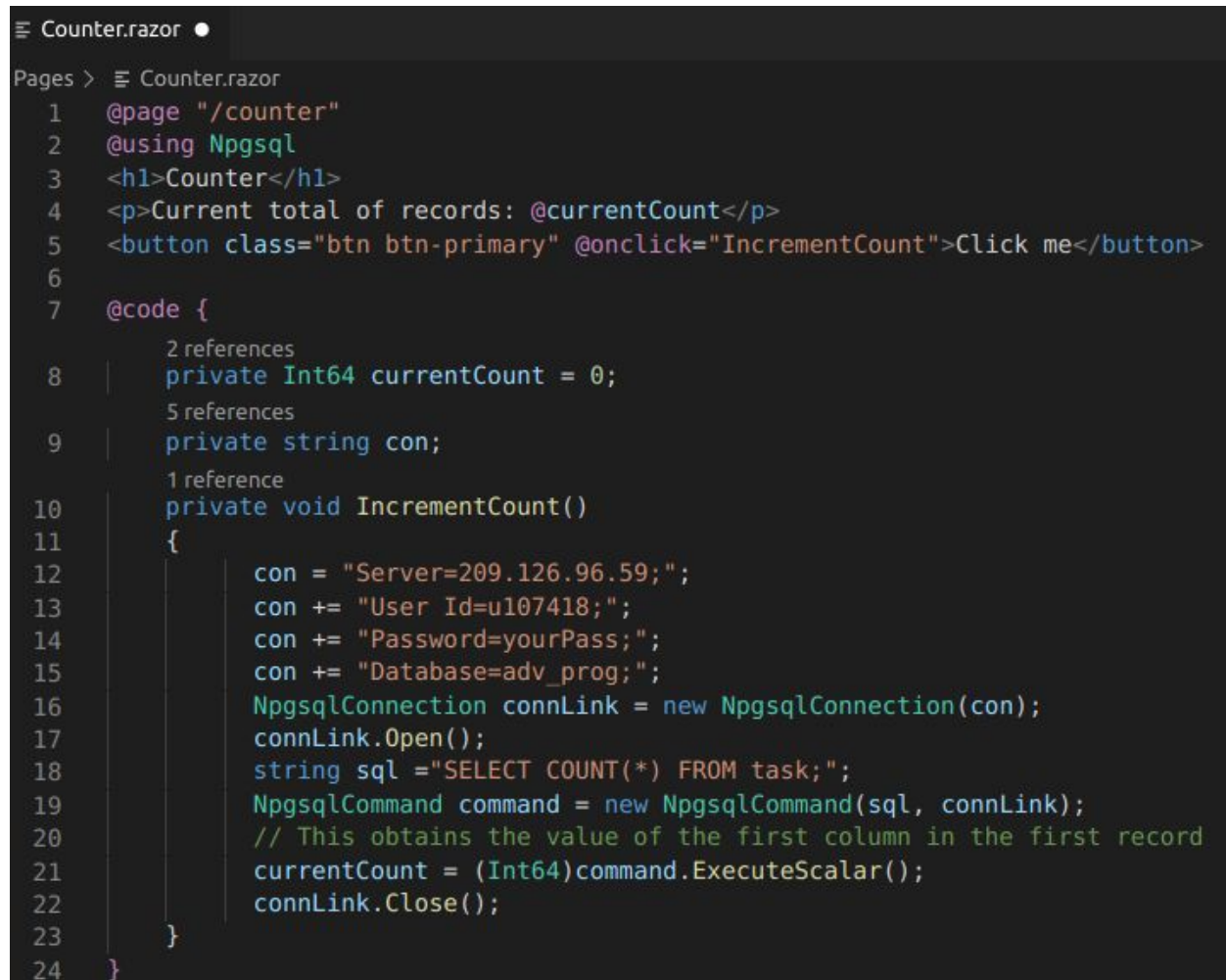
Now we will create a basic blazorserver project for testing Database connections. The command is

```
dotnet new blazorserver -o data01
```

Enter to the project by doing `cd data01`, and then add the package Npgsql that will allow you to connect, from a dotnet application to a Postgres database.

```
Dotnet add package Npgsql
```

Change the content of Counter.razor page to:

A screenshot of a code editor with a dark theme. The file name 'Counter.razor' is visible in the top left. The code is a Blazor Razor page that uses the Npgsql library to connect to a PostgreSQL database. It features a page directive, a using directive, a header, a paragraph, and a button. The code block contains a private integer for the current count, a connection string, and a method to increment the count by querying the database. The code is as follows:

```
1 @page "/counter"
2 @using Npgsql
3 <h1>Counter</h1>
4 <p>Current total of records: @currentCount</p>
5 <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
6
7 @code {
8     2 references
8     private Int64 currentCount = 0;
9     5 references
9     private string con;
10    1 reference
10    private void IncrementCount()
11    {
12        con = "Server=209.126.96.59;";
13        con += "User Id=u107418;";
14        con += "Password=yourPass;";
15        con += "Database=adv_prog;";
16        NpgsqlConnection connLink = new NpgsqlConnection(con);
17        connLink.Open();
18        string sql = "SELECT COUNT(*) FROM task;";
19        NpgsqlCommand command = new NpgsqlCommand(sql, connLink);
20        // This obtains the value of the first column in the first record
21        currentCount = (Int64)command.ExecuteScalar();
22        connLink.Close();
23    }
24 }
```

From lines 12 to 15 a connection string is defined. In lines 16-17 a connection to the database is established. In lines 18-20 a sql command is, defined, executed and its answer recovered. As the result is a number of the total amount of records in table `task`, the answer is 1 and will

continue to be 1 until more records are added. Thus, please add another record in the table by recovering the last insertion (up arrow in the psql console), change the data, and add another record. For example:

```
adv_prog=> INSERT INTO task(taskname,
supervisor,
email_supervisor,
responsible,
email_responsible,
created_on,
deadline,
estimate_hours,status)
VALUES ('search in an array the index of the first field-value pair',
'Maria Alvarez',
'malv@highcompany.pl',
'Peter Maiden',
'pmaiden@highcompany.cl',
NOW(),
'2020-12-29 12:00:00',
14,
'Running') Returning task_id;
task_id
-----
      2
(1 row)

INSERT 0 1
adv_prog=> 
```

You must show the counter before the insertion and after the insertion. Like this:



Comment: Any error in your connection string OR in your SQL command will imply that an error will occur and no effect will be on the web page. You can see the error by opening the inspect windows in your internet browser and looking for the specific error there.

Tut-5-P2-A. Screenshot of the version of your new Counter.razor page

Tut-5-P2-B. Screenshot of the output of the new insertion. Better if you try your own additional insertion.

Tut-5-P2-C. Screenshot of the counter before and after the insertions.

Part-3 (25%). Displaying data from a table

Now we will display the data from our table. However, in order to use the connection in the displaying section of the code, it is needed to define the connection variables out of the function. We define the variable `data` of the type `NpgsqlDataReader`, which contains the answer to the SQL command. This kind of variable can know if there are some returning rows (`HasRows` property), it can iterate reading the rows (`Read()` method), and get the corresponding column (`GetInt32()` or `GetString()` methods) of the current row. Copy the next code as the displaying part of the page.

```
Counter.razor X
Pages > Counter.razor
1 @page "/counter"
2 @using Npgsql
3 <h1>Counter</h1>
4 <button class="btn btn-primary" @onclick="IncrementCount">
5 Running Tasks</button>
6 <hr>
7 @if(data!=null) {
8     @if(data.HasRows) {
9         int tot=0;
10        <table border="1">
11            <tr><td>Id</td><td>Task</td><td>On charge</td><td>Deadline</td></tr>
12            @while (data.Read()) {
13                tot++;
14                <tr style="font-size:9">
15                    <td>@(data.GetInt32(0))</td>
16                    <td>@(data.GetString(1))</td>
17                    <td>@(data.GetString(2))</td>
18                    <td>@(data.GetString(3))</td></tr>
19            }
20        </table>
21        <p>Total of Running Tasks: @tot</p>
22    }
23    else{
24        <h3>No Running Tasks</h3>
25    }
26 }
27
```

Basically, line 7 verifies if there is a SQL answer (the first time there is no). Line 8 verifies either it has rows or not. If there are some rows a table is generated. In line 11 the head of the table is generated. In line 12 to 19 each cell is generated. The methods `GetInt32` and `GetString` obtain the columns 1 to 4 of the answer. Therefore, for generating the display is necessary to know what was exactly the SQL command. In this case, the SQL command is:

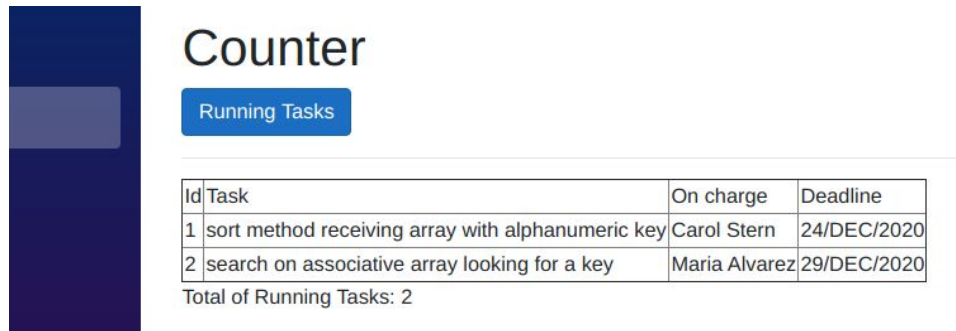
```
SELECT task_id, taskname, responsible, to_char(deadline,'DD/MON/YYYY')
FROM task WHERE status='Running';
```

There is a known issue about recovering timestamp data types because this type of data has several forms to be converted to a string. Therefore, a solution is to ask for the specific format of each timestamp, in this case, we recover the field `deadline` in a readable form by using the `to_char` Postgres function.

In the code section, we have defined both necessary variables for the connection, the object representing the connection itself (`connLink` variable) and the data obtaining from the query (`data` variable).

```
Pages > Counter.razor
28 @code {
    5 references
29     private string con;
    8 references
30     private NpgsqlDataReader data=null;
    3 references
31     private NpgsqlConnection connLink;
    1 reference
32     private void IncrementCount()
33     {
34         con = "Server=209.126.96.59;";
35         con += "User Id=u107418;";
36         con += "Password=yourPasswd;";
37         con += "Database=adv_prog;";
38         connLink = new NpgsqlConnection(con);
39         connLink.Open();
40
41         string sql ="SELECT ";
42         sql += " task_id, taskname, responsible, ";
43         sql += " to_char(deadline,'DD/MON/YYYY') ";
44         sql += " FROM task WHERE status='Running'";
45         NpgsqlCommand command = new NpgsqlCommand(sql, connLink);
46         data = command.ExecuteReader();
47         //connLink.Close();
48     }
49 }
```

Although this version works, it requires that the connection do not be close because the `NpgsqlDataReader` object requires an open connection to work. The corresponding output for this version is:



Please as evidence of your work deliver the following items:

- Tut-5-P3-A. Screenshot of the version of your new Counter.razor page separated in the two sections as it has been shown.
- Tut-5-P3-B. Screenshot of the output in the browser.
- Tut-5-P3-C. Screenshot of the output in the psql for the given SQL command (SELECT ...).

Part-4 (25%). Architecting an object-oriented approach

Although the previous version works it has several problems. For example, the way of referring fields depending on the order of the column means that you have high coupling elements because the way of displaying depends on the way data has been recovered. Another problem is that for sure, in a real system, a page will not be the only code section where the database will be connected. Therefore, in order to avoid an aspect, i.e. a crosscutting concern, we need a unique part of the code where the connection is established. Another problem is that “Tasks” seems to be a relevant business concept that has and will have, for sure, business rules to follow. Therefore, a class Task is needed and, will be needed. However we have a collision of names here because “Task” is also a relevant class in the dotnet framework, thus, we will call it DBTask.

Please replicate the following files in order to have a better architecture of objects (not that I say better just because the previous version is a bad approach from [Software Engineering principles of low coupling](#)). However, it does not mean that the proposed architecture be the best. It still has some problems.

The first file to copy is class MyPostgresDB. This class opens the connection in the constructor and keeps open the connection during its cycle of life, but, on the contrary to the previous solution, it closes the connection because the destroyer has been implemented. The destroyer is automatically called when the object reaches the final of its scope (line 17).

Class MyPostgresDB in MyPostgresDB.cs


```

Data > MyPostgresDB.cs > {} data01.Data > data01.Data.MyPostgresDB > MyPostgresDB()
1  using System;
2  using Npgsql;
3  namespace data01.Data
4  {
5      1 reference
6      public class MyPostgresDB
7      {
8          2 references
9          private NpgsqlDataReader data=null;
10         4 references
11         private NpgsqlConnection lnk;
12         1 reference
13         public MyPostgresDB() {
14             string con = "Server=209.126.96.59;";
15             con += "User Id=u107418;";
16             con += "Password=yourPasswd;";
17             con += "Database=adv_prog;";
18             this.lnk = new NpgsqlConnection(con);
19             lnk.Open();
20         }
21         0 references
22         ~MyPostgresDB(){ //destroyer
23             this.lnk.Close();
24         }
25         1 reference
26         public NpgsqlDataReader executeSQL(string sql) {
27             NpgsqlCommand command = new NpgsqlCommand(sql, this.lnk);
28             data = command.ExecuteReader();
29             return data;
30         }
31     }
32 }

```

Finally and obviously, a change on the database name or user will imply only to change this class and not several files in the system.

Note that the method in line 20 is generic enough for several SQL commands such as INSERT or UPDATE.

We follow by showing the class for the DBTask

```

Data > C# DBTask.cs > {} data01.Data > data01.Data.DBTask
1  using System;
2  using Npgsql;
3  namespace data01.Data
4  {
5      3 references
6      public class DBTask
7      {
8          2 references
9          public long task_id=-1;
10         2 references
11         public string taskname="";
12         1 reference
13         public string supervisor="";
14         1 reference
15         public string email_supervisor="";
16         2 references
17         public string responsible="";
18         1 reference
19         public string email_responsible="";
20         0 references
21         public string created_on="";
22         2 references
23         public string deadline="";
24         0 references
25         public string first_delivering="";
26         0 references
27         public string first_revising="";
28         0 references
29         public int estimate_hours=-1;
30         0 references
31         public int effective_hours=-1;
32         0 references
33         public string status="Running";

```

In this part of the class, only the default values are set. Note that timestamps are handled as strings which would not be necessarily a good decision.

The class continues with the method `set_from_reader`. This method parses the `DataReader` in their corresponding fields. This way we can make transparent the order of fields in the `SELECT` commands. However, note that it is not complete for all the fields existing in the table `task` of the database `adv_prog`.

```

20     public bool set_from_reader(NpgsqlDataReader r) {
21         if(r.IsOnRow) {
22             for(int i=0; i<r.FieldCount; i++) {
23                 string fname = r.GetName(i);
24                 if(fname=="task_id")
25                     this.task_id=r.GetInt64(i);
26                 else if(fname=="taskname")
27                     this.taskname=r.GetString(i);
28                 else if(fname=="supervisor")
29                     this.supervisor=r.GetString(i);
30                 else if(fname=="email_supervisor")
31                     this.email_supervisor=r.GetString(i);
32                 else if(fname=="responsible")
33                     this.responsible=r.GetString(i);
34                 else if(fname=="email_responsible")
35                     this.email_responsible=r.GetString(i);
36                 else if(fname=="deadline")
37                     this.deadline=r.GetString(i);
38             }
39             return true;
40         }
41         return false;
42     }
43 }
44 }

```

Finally, we show the @code section first in order to explain their components but you know that this section goes after the display section on the razor page. Here we use the MyPostgresDB class for keeping the connection to the database and also we keep a list of DBTasks that is updated each time the button is pressed. Although this version is better than the first one, it has problems yet. For example, we have an SQL command and we parse the general reader in the code of a displaying page. That is because we create the class DBtask but we do not create the class DBTaskCollection in order to handle sets of DBTasks, therefore, we still have coupling between data management and the view system. The code section of the razor page follows:

```

28  @code
29  | 1 reference
   | private MyPostgresDB myDB = new MyPostgresDB();
30  | 4 references
   | private List<DBTask> myTasks = new List<DBTask>();
   | 1 reference
31  | private void GetRunningTasks()
32  | {
33  |     myTasks.Clear();
34  |     string sql = "SELECT ";
35  |     sql += " task_id, taskname, responsible, ";
36  |     sql += " to_char(deadline, 'DD/MON/YYYY') as deadline ";
37  |     sql += " FROM task WHERE status='Running'";
38  |     DBTask tsk;
39  |     NpgsqlDataReader data = myDB.executeSQL(sql);
40  |     while(data.Read()) {
41  |         tsk = new DBTask();
42  |         tsk.set_from_reader(data);
43  |         myTasks.Add(tsk);
44  |     }
45  | }
46  | }
47

```

Finally, the displaying section remains this way:

```

Pages > Counter.razor
1  @page "/counter"
2  @using Npgsql
3  @using System.Collections.Generic
4  @using Data
5  <h1>Counter</h1>
6  <button class="btn btn-primary" @onclick="GetRunningTasks">
7  Running Tasks</button>
8  <hr>
9  @if(myTasks.Count>0) {
10     int tot=0;
11     <table border="1" class="table">
12         <tr><td>Id</td><td>Task</td><td>On charge</td><td>Deadline</td></tr>
13         @foreach (var tsk in myTasks) {
14             tot++;
15             <tr style="font-size:9">
16                 <td>@(tsk.task_id)</td>
17                 <td>@(tsk.taskname)</td>
18                 <td>@(tsk.responsible)</td>
19                 <td>@(tsk.deadline)</td></tr>
20         }
21     </table>
22     <p>Total of Running Tasks: @tot</p>
23 }
24 else{
25     <h3>No Running Tasks</h3>
26 }
27
28 @code {

```

Note that we can use the name of attributes for generating the view (lines 16 to 19) which makes the code more maintainable and legible.

Please as evidence of your work deliver the following items:

- Tut-5-P4-A. Screenshot of the code of your MyPostgresDB class.
- Tut-5-P4-B. Screenshot of the code of your DBTask class.
- Tut-5-P4-C. Screenshot of the new version of your new Counter.razor page separated in the two sections as it has been shown.
- Tut-5-P4-D. Screenshot of the output in the browser.