



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Academic Master in Geomatic Engineering and Geoinformation WEB DEVELOP AND GEOPORTALS

Guide of developed contents in lectures.

J. Gaspar Mora Navarro.

Univesitat Politècnica de València.
Departamento de Ingeniería Cartográfica, Geodesia y Fotogrametría Department of Cartographic
Engineering and Photogrametry

Valencia, February 2017

WEB DEVELOP AND GEOPORTALES.

J. Gaspar Mora Navarro

10 de julio de 2023

Índice general

1	Evaluation	1
1.1	Evaluation	2
2	Dynamic sites with Python and WSGI	4
2.1	Introduction	5
2.2	Backend software and libraries	6
2.3	Python necessary previous knowledge	7
2.3.1	How to import Python modules	7
2.3.1.1	Practical example	9
2.3.2	Proposal exercise	11
2.3.3	Solution with Object Oriented Programing (OOP)	12
2.3.3.1	OOP proposal exercise	16
2.3.3.2	OOP proposal exercise solution	18
2.3.4	Python string formatting	20
2.3.4.1	Formatting a string using the order of the variables	20
2.3.4.2	Formatting a string using names in the formatted string	21
2.3.4.3	Proposal exercise	21
2.4	Database connection with Pyhton 3.6. Use of the <i>psycopg2</i> library	21
2.4.1	Insert rows with geometry	23
2.4.1.1	Proposed exercise: insert	24
2.4.2	Update rows	24
2.4.2.1	Proposed exercise: update	26
2.4.3	Code optimization	26
2.4.4	Delete rows	28
2.4.4.1	Proposed exercise: optimize the modules <code>building_insert.py</code> and <code>building_update.py</code>	28
2.4.5	Select rows	28
2.4.5.1	Select rows	28
2.4.5.2	Select rows getting a dictionary for each row	30
2.4.5.3	Proposed exercise	32
2.5	Managing JSON strings	32
2.5.1	Communication client - server	32
2.5.2	Creating JSON strings with Python	32
2.5.3	Decode a JSON string to obtain a Python dictionary	33
2.5.4	Decode a JSON string to obtain a JavaScript object	34
2.6	Create functions to easily insert, delete update and select buildings	34
2.6.1	Function to insert a building from a Python dictionary	34
2.6.2	Function to update a building from a Python dictionary	36
2.6.3	Function to delete a building from gid	37
2.6.4	Function to select a building from gid	37
2.7	Exercise 1.	38

2.7.1	Part 1. Test. Value 2 points	38
2.7.2	Part 2. Project. Value 1 point. Create functions to modify your own Postgis data- base with JSON strings.	38
2.8	Connect Python functions to Internet with Django 2.2	39
2.8.1	Create an Django project and app	40
2.8.2	Configuring the Django project	42
2.8.3	Initial configuration of the app urls and views	45
2.8.4	Creating the app urls and views to access to the database	47
2.8.4.1	Select a building by gid	47
2.8.4.2	Insert a building	49
2.8.4.3	Update a building	52
2.8.4.4	Delete a building	54
2.9	Sending data to the views: POST or GET	55
2.9.1	Postman	55
2.9.2	Python requests	56
2.10	Django file and functions structure	56
2.11	Users management and authentication	57
2.11.1	Create users and groups with the Django admin site	57
2.11.2	Create super-user from command	59
2.11.3	Create normal users from command	59
2.11.4	Change users password from command	59
2.11.5	Protect some views from unauthenticated users	59
2.11.6	Authenticate users	62
2.11.7	Session expiration time	66
2.11.8	Logout a user	66
2.11.9	Limit the access to views to users that belongs to some groups	67
2.11.10	Users management with Python. Official documentation	67
2.11.11	Create users with Python	67
2.11.12	Add a user to a group with Python	68
2.11.13	Active - deactive users with Python	68
2.11.14	Change users password with Python	68
2.12	Publish a Django app with Apache2. WSGI application	69
2.12.1	Give permission to Apache to be able to read the wsgi file	69
2.12.2	Change the Apache configuration	70
2.12.3	Enable the Django admin site with Apache	71
2.13	Debugging Python WSGI applications	73
2.13.1	Debugging Python code. Apache file <i>error.log</i>	73
2.13.2	Remote debugging with PyDev	75
2.14	In case of error 1	75
2.14.1	How to make a question	76
2.14.2	How to know where the Python error is	76
2.15	Geometry checks (optional)	77
2.15.0.1	Function to select the gid of the geometries nearer of a geometry	78
2.15.0.2	Use the geometry check function before insert or update	79
2.15.1	Intersection check considerations	80

3 Database update through Internet. Ajax	81
3.1 Goals in this chapter	82
3.2 Create a minimal web page	82
3.3 Visit the web page	84
3.4 Create form to get the user data in order to login	84
3.5 Styling with Bootstrap	85
3.6 Create form to insert a building	87
3.7 Add a paragraph and a div for the future map	87
3.8 Link javascript code to the page	88
3.8.1 Link a Javascript file to the web page	88
3.8.2 First JS code. Window on load event	88
3.8.3 Link button click events to a function	89
3.9 What to do in case of error 2	91
3.9.1 See the console messages	91
3.9.2 Stop the JavaScript execution	92
3.9.3 Check if all the files are being loaded	93
3.9.4 Check the order of the JavaScript files	94
3.10 Create an interactive navigation menu with Bootstrap and JavaScript	95
3.11 Get the form data	98
3.11.1 Get a form control value	98
3.11.2 Set a form control value	99
3.11.3 Get all form control values at once	99
3.12 Use Ajax to send the form data to the server and wait for its response	99
3.12.1 Login the user	99
3.13 Solve the CORS error of Google Chrome	101
3.14 Use a Javascript settings file to configure the Javascript application. <i>mySettings.js</i>	102
3.14.1 Login	104
3.14.2 Logout	105
3.14.3 Insert a building	105
3.15 What to do in case of error 3	106
3.15.1 Where I am sending the data	106
3.15.2 What I am sending to the server	107
3.15.3 What is the server responding	108
3.15.4 My JavaScript code does not refresh	108
3.16 Whether to use Django developing server or Apache server	109
3.17 Change the page content to show to the user the server answer	109
3.18 Create an Apache alias for the web page	109
3.19 Hide the JavaScript code	109
3.20 Exercise 2.	110
3.20.1 Part 1. Test. Value 2 points	110
3.20.2 Part 2. Project. Value 1 point. Create a web page to update the tables of your database.	110
4 Create a map with OpenLayers 6.1.1	112
4.1 Goals	113
4.2 Download and install the libraries	113
4.3 Create the WMS service of the layer buildings	114
4.4 Create a map with the Spanish Cadastre and The Spanish ortophoto (PNOA)	114

4.5	How to use the OpenLayers examples	118
4.6	Draw polygons in the map	118
4.6.1	Add the draw interaction to the map	118
4.6.2	Enable or disable the draw interaction	119
4.6.3	Clear the content of the vector layer	119
4.6.4	Reload a WMS Layer	120
4.6.5	Call the <i>addDrawPolygonInteraction</i> function	121
4.6.6	Add buttons to enable and disable the draw interaction	121
4.7	Send the drawn polygons in the web page to the database	122
4.7.1	Link the draw end event of the geometry with a callback function	122
4.7.2	The callback function gets the geometry coordinates and puts them into the form	123
4.8	Add draw interaction of different geometry types to the map	123
4.9	Exercise 3.	124
4.9.1	Part 1. Test. Value 2 points	124
4.9.2	Geoportal project requirements and evaluation.	124

Índice de figuras

2.1	Geoportal parts schema	5
2.2	Importing modules project	9
2.3	Virtualenv Python interpreter selection in Eclipse	11
2.4	Virtualenv Python interpreter selection in Eclipse	11
2.5	PgAdmin 4 showing the buildings database	22
2.6	Execute a Python program in Eclipse	24
2.7	The two buildings inserted seen in PgAdmin	25
2.8	Inserted buildings in in Qgis	26
2.9	Postgis connection definition in Qgis	26
2.10	Get the WKT coordinates with Qgis	27
2.11	Add some more buildings, digitizing blocks from the Spanish cadastre	29
2.12	File structure created by the command django-admin startproject djdesweb	41
2.13	File structure created by the command python manage.py startapp appdesweb	42
2.14	Django project edited with Eclipse	42
2.15	File structure created by the command python manage.py startapp appdesweb	44
2.16	Django HelloWorld view answer	46
2.17	Current project files	48
2.18	Building select result	48
2.19	You can not send a post request with the web browser	51
2.20	Create a collection with Postman	51
2.21	Set the collection name with Postman	52
2.22	Add a request to a collection with Postman	52
2.23	Set the request type with Postman	53
2.24	Set the formData variable with Postman	53
2.25	Set the formData variable with Postman	54
2.26	Django file structure	56
2.27	Django admin site. Login	57
2.28	Django admin site. Add group	58
2.29	Django admin site. Add user	58
2.30	Django admin site. Add user to a group	59
2.31	Login with PostMan	64
2.32	Session ID stored in a cookie, in the headers of the request, and in the database, in the django_session table	65
2.33	Django app running under an WSGI application, so accessible by Apache	71
2.34	Django admin site without styles	72
2.35	Django admin site collect static files	72
2.36	Django admin site published with Apache2 and been able to get the static files	73
2.37	Internal server error showed when the DEBUG setting is set to False	73
2.38	Enabling insecure access to the email Google account	74
2.39	Remote debugging with Pydev	75

2.40 Remote debugging with Pydev 2	77
3.1 Project structure for the front-end project	83
3.2 Page previsualization 1	84
3.3 Form login preview	85
3.4 Form styling with bootstrap	85
3.5 Page with the Login and Buildings forms	88
3.6 Message when the page is completely loaded	89
3.7 Show Google Chrome Developer Tools	90
3.8 Message in the console tag of the web developer tools	90
3.9 Console with the error message and the lines where the error was triggered	91
3.10 First line where the error was triggered	91
3.11 Reference error	92
3.12 Reference error	92
3.13 Stop the code execution in a line	93
3.14 Buttons to advance the execution	93
3.15 Local vars values	94
3.16 Navigation menu with Bootstrap	95
3.17 Chrome CORS error	101
3.18 Chrome CORS security disabled for development	102
3.19 Login answer, and session ID cookie	104
3.20 How to see the requests that my web does	106
3.21 How to see the where I am sending the data	107
3.22 How to see what I am sending to the server	107
3.23 How to see the server response	108
3.24 Disable Chrome cache to use the last version of the js code	108
4.1 OpenLayers map	117
4.2 Draw polygons with OpenLayers	122

CAPÍTULO 1
Evaluation

1.1 Evaluation

The evaluation consists in three exercises, each of which is also divided in two parts, a written exam and a practical exercise.

- Exercise 1. Total value 30% = 3 points. (On 30 of March)
 - Teorical written exam 2 points.
 - Practical exercise or project 1 point. In this exercise you will create Python functions to connect with PostGIS and be able manage spatial data.
- Exercise 2. Total value 30% = 3 points. (On 25 of May)
 - Teorical written exam 2 points.
 - Practical exercise or project 1 point. In this exercise you will create a basic web, and will make requests to the server with Ajax. The request will execute your Python functions of the exercise 1, to manage the database spatial data.
- Exercise 3. Total value 40% = 4 points. (On 14 of June)
 - Teorical written exam 2 points.
 - Practical exercise or project 2 point. In this exercise you will add to your basic web, made in the exercise 2, a map that publishes your database spatial data. You will add the functionality of draw new elements on the map, and add them to the spatial database in the server.
- Last chance. All course content 3 July.

Practical exercises requirements:

- There are three practical exercises in total. The practical exercises must be related forming a geoportal project. The practical exercise 2 must continue the work done in the practical exercise 1, and the practical exercise 3 must continue the exercise 2. On finishing the practical exercise 3 you must have a geoportal with some minimum requirements. This is the project specified in the subject guide.
- The practical exercises can be done individually, in pairs or in groups of three people. You must form groups at the beginning of the course, inform the teacher of the group members. This groups can not be changed.
- The exercises are the ones which appears in this document. The delivery of the exercises consists in showing the exercise to the lecturer, and to answer some questions about them. Before this, the code of the exercise must have been uploaded to the shared space in Poliformat, otherwise you will lose the corresponding point or points. The exercises consist in to make functions. They have to work and you have to have a demonstration prepared, with proper data to run these functions.
- The written exams consist in individual online written exams, where you will be able to use any document, but not your mobile. Mostly you will have to type some small functions to demonstrate you master the subject contents. The functions will be similar, but not equal, to those developed in your practical exercises. The small differences are thought to you demonstrate your understanding about important concepts.

- The three practical exercises must form a geoportal at the end of the course. The subject and data of the geoportal must be chosen by the student at the beginning, at the exercise 1. The geoportal has a minimum requirements.
- If you are doing the *Gestores de contenidos Geoespaciales y Smart Cities* subject, you will learn about web servers. You will have the opportunity of set the server to publish you geoportal in a real server, so anyone will be able to visit your geoportal. This connect also *Desarrollo Web y Geoportales* subject with the subject *Distribucion de la informacion espacial*.
- We also recommend to use the same student groups in all these three subjects, but this is not fundamental.
- The minimum number of tables to use, to make the exercises, and the geoportal are three, and at least two of them must have a geometry column of different geometry type (point, polygon or linestring).
- The geoportal will be showed to the teacher and will be evaluated at that moment. If the geoportal does not work you will not get any point.

CAPÍTULO 2

Dynamic sites with Python and WSGI

2.1 Introduction

A geoportal y composed by may parts. In the figure 2.1 you can find a schema:

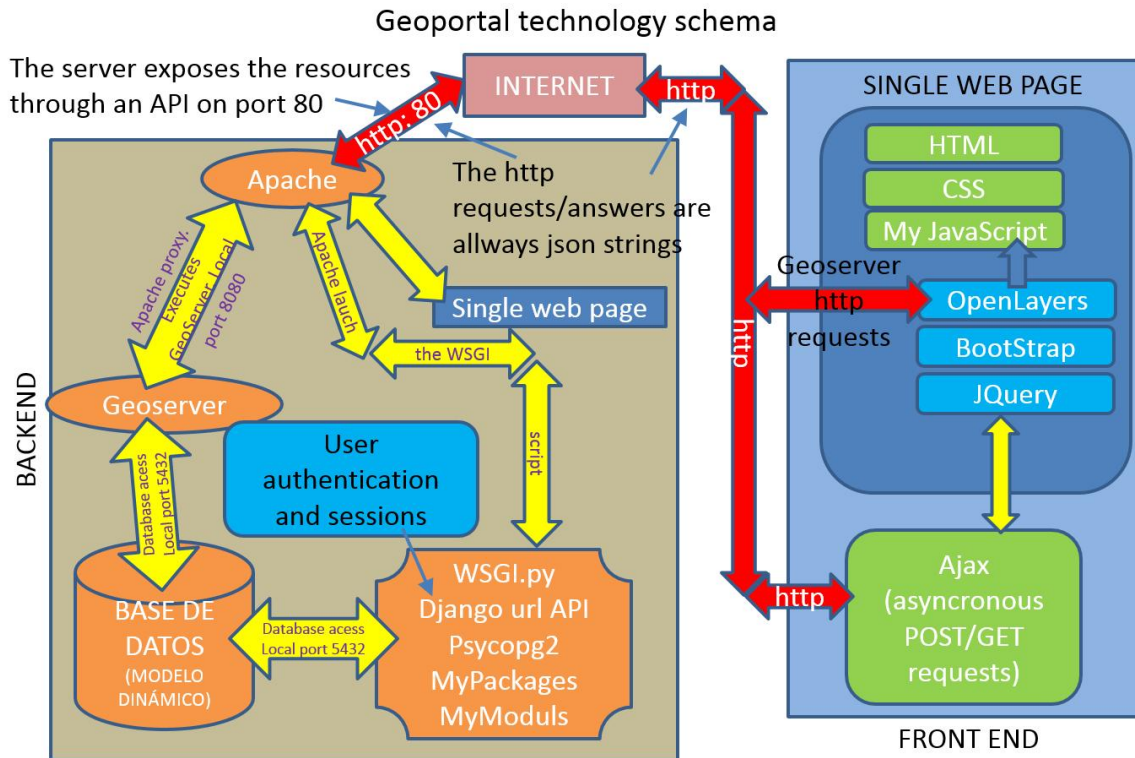


Figura 2.1: Geoportal parts schema

As you can see at first sight it is not a simple task to build a geoportal. It is composed by many parts and each part is composed by many parts, each of which needs a good level of developing skills. It's a too big task for one solely subject. The goal of the subject is to see all the parts, so that the student will have a good overview of how a geoportal works. Of course we can not see each part in deep, but knowing how is the philosophy the student will be able to go more deep if he needs it. The good thing is that once the geoportal is built all the users have access and all of them are working with the same data, so the editions of one users are immediately seen by the rest of the users.

The teacher will explain how to create a basic geoportal to add polygons to a database. To pass this subject you will have to do the same with your own tables. What the teacher is expecting about you, as minimum, is you understand the examples and you be able to modify them for your own project. These tables will come from the subject *Distribución de la Información Espacial*, in order to continue your work about INSPIRE.

To learn we are going to build a simple geoportal, but which contains all the parts showed in the schema. You can access to the geoportal with the url <https://gisserver.car.upv.es/desweb/>.

You can form groups of one, two or three people. I recommend you groups of two people, because it is good for you to be forced to join code from others. Also it is good to work in group because you can share knowledge.

You have to keep in mind that 6 out of 10 points of this subject are in written tests where you are alone, so you have to understand all the code of your project or exercises.

2.2 Backend software and libraries

In this section it is listed the software used in the server.

The server programming part, called *backend*, its going to be done with Python 3.6.8, with WSGI applications. One application WSGI is a Python program connected to Apache server with a WSGI directive. Apart of this is a normal Python program.

To help as to manage http requests and user sessions, we are going to use Django 2.2 and its dependencies if you want to deploy an app with this technology, for example in a Ubuntu Server you will have to install the following in the system:

- Python >= 3.5
- Pip3
- libpq-dev: `sudo apt-get install libpq-dev`
- Apache2: `sudo apt-get install apache2`
- mod_wsgy para python 3: `sudo apt-get install libapache2-mod-wsgi-py3`
- Enable Apache2 Headers module: `sudo a2enmod headers`

The recommended way to make a Python develop environment is to use a python virtual environment, which are isolated Python environments. The first step is to install the *virtualenv* program in the system (<https://virtualenv.pypa.io/en/latest/>):

```
sudo pip3 install virtualenv
```

You can create a virtualenv, called *env* in the current folder with:

```
virtualenv env
```

To install Django and its dependencies you must activate the virtualenv and use *pip* to install the python libraries:

```
source env/bin/activate /*Activate the virtual env*/
pip3 install Django==2.2.0
pip3 install Pillow /*dependencia de django*/
pip3 install psycopg2 /*dependencia de django*/
```

Of course you need a geospatial database and a web server. So you have to install PostgreSQL + Postgis and GeoServer, which runs under Apache Tomcat which have to be installed first.

All these software is difficult to install, so we are going to use a virtual machine which has all the software installed. You will deploy your geoportals probably in a ubuntu server. We are going to use a Ubuntu desktop 18.04 virtual machine (VM). It is exactly equal that an Ubuntu Server but has a visual environment to develop. So we are going to develop and publish at the same machine. In a real case you will develop in one machine and deploy, or publish your geoportals in other machine.

In the VM it is installed

- PostgreSQL 11 + Postgis 2,5 + Pgrouting + adminpack and PgAdmin 4
- Tomcat 9
- GeoServer 2.16

The previous programs are quite more difficult to install. Bot have detailed steps in its main pages. Also, to code, it is installed the following IDEs:

- Eclipse + PyDev
- Visual Studio Code

2.3 Python necessary previous knowledge

2.3.1 How to import Python modules

You are used to make scripts in a single file. In web developing that approach is not possible. A minimum geoportal is a very big program. You will need to divide the code in files, according a logic. The first step is to be able to create python modules and packages and know how to load them, sou that you can divide the code in small parts.

It is very important you understand the Python import system. In the following, the current folder is the folder that contains the current file, and the current file is the file that is currently being executed, imported or edited:

- A Python module is a python file, that is a file with the .py extension. A package is a folder with several python modules, and a special empty file called `__init__.py`. **Without the file `__init__.py`, any module could be imported. A folder with modules, but without the file `__init__.py` is a normal folder not a package, so python will not be able to find anything.**
- When you type `import a` Python searches for the module `a` in the current folder, if it is not, searches for the `a` package, also in the current folder, if it is not searches for the `a.py` in the `sys.path` folders list, if it is not, searches for the `a` package in the `sys.path` folders list.
- `__file__` is a global variable, always available in all the files. It is the file name. For example for the module `a.py`, the value is `a.py`. To get the absolute path of the file name you have to use the function `os.path.abspath(__file__)`. For example the function could return `/home/vagrant/apps/desweb/a.py`
- The function `os.path.dirname()` returns the folder containing the file or folder. For example. If you are editing the file `/home/vagrant/apps/desweb/a.py`

```
import os
print(os.path.abspath(__file__)) #print /home/vagrant/apps/
    desweb/a.py
fileName=os.path.abspath(__file__)
print(os.path.dirname(fileName)) #print /home/vagrant/apps/
    desweb
print(os.path.dirname(os.path.dirname(fileName))) #print /home/
    vagrant/apps
print(os.path.dirname(os.path.dirname(os.path.dirname(fileName)
    ))) #print /home/vagrant
```

- If you want to import the module `m1.py` inside the package `p1`, which is inside the current folder you can do it easily:

```
from p1 import m1
```

- But if you want to import the module *padre* in `/home/vagrant/apps/p1/padre.py`, and the current file is `/home/vagrant/apps/desweb/p2/master.py`, you must to add `/home/vagrant/apps/desweb` to the `sys.path` list **before to try the import**. Of course you could use absolute paths:

```
import sys
sys.path.append("/home/vagrant/apps/desweb")
from p1 import padre
```

The above code will work, but it is a very beginner mistake. You could not move your project, because the absolute path will change.

You always must use relative paths to the location of the current file, stored in the variable `__file__`. You must upload one level in the folder that contains the current file and add that folder to the `sys.path` list.

```
import os, sys
print(sys.path)
print(__file__)
BASE_DIR= os.path.dirname(os.path.dirname(os.path.abspath(
    __file__)))
sys.path.append(BASE_DIR)
print(sys.path)
from p1 import padre
print("padre imported")
```

The above script has the following result:

```
["/home/vagrant/apps/desweb/conf", "/home/vagrant/apps/desweb",
 "/home/vagrant/apps/env/lib/python3.6", "/home/vagrant/apps/
env/lib/python3.6/lib-dynload", "/usr/lib/python3.6", "/home
/vagrant/apps/env/lib/python3.6/site-packages", "/home/
vagrant/apps/env/lib/python36.zip"]
/home/vagrant/apps/desweb/conf/m2.py
["/home/vagrant/apps/desweb/conf", "/home/vagrant/apps/desweb",
 "/home/vagrant/apps/env/lib/python3.6", "/home/vagrant/apps/
env/lib/python3.6/lib-dynload", "/usr/lib/python3.6", "/home
/vagrant/apps/env/lib/python3.6/site-packages", "/home/
vagrant/apps/env/lib/python36.zip", "/home/vagrant/apps/
desweb"]
padre imported
```

In the above listing we added the folder `/home/vagrant/apps/desweb` to the list `sys.path`. If you realize that folder was already in the list, at the second position. This is something that only occurs when running programs with Eclipse. Eclipse adds the project folder to `sys.path`, but when you run the program without Eclipse it will fail, if you manually do not add `/home/vagrant/apps/desweb` to `sys.path`.

2.3.1.1. Practical example

Usually a Python project has many modules and packages, but the project has an unique point of entry. This is the module that executes the project, usually called *main.py* (In our case we will call it *wsgi.py* because we are going to use Django). The *main* module has a function, usually also called *main*, that start the projects. The *main* module loads an other project modules and these modules other project modules. In this section you will see the recommended way of doing that, in a real project. Follow the following steps:

- Start an Eclipse PyDev project called *importingModules*. Select the option *File ->New ->Pydev project*
- As Python interpreter you mus choose the interpreter of the virtual environment located in `/home/vagrant/apps/env` (figure 2.3, figure 2.4), otherwise you will not have all the libraries available.
- Create a Python package called *p1* (right button over the project name).
- Create a Python package called *p2*.
- Create the module *p2/main.py*
- Create the modules inside the package p1 called *padre*, *hijo* and *nieto* (figure 2.2).
- Our goal is from *main* module to load the *padre* which loads the *hijo* wich loads the *nieto* modules.

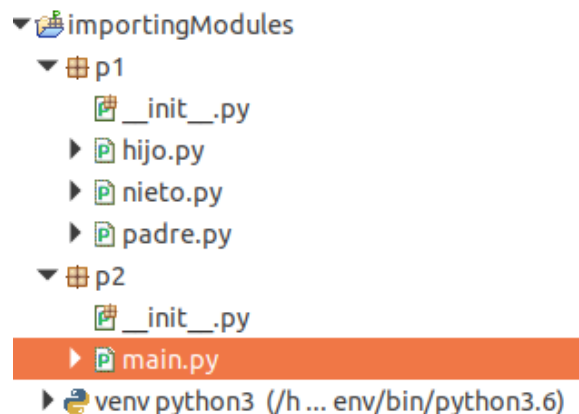


Figura 2.2: Importing modules project

- In the following listing you have the code of each module:

```
### p2/main.py ###  
  
from p1 import padre  
  
print(__file__)  
padre.printPadre()  
  
### p1/padre.py ###  
from p1 import hijo  
  
def printPadre():
```

```
print("Padre")
print(__file__)
print(__name__)

hijo.printHijo()
hijo.printNieto()

### p1/hijo.py ###
from p1 import nieto

def printHijo():
    print("Hijo")
    print(__file__)
    print(__name__)

def printNieto():
    nieto.printNieto()

### p1/nieto.py ###
def printNieto():
    print("Nieto")
    print(__file__)
    print(__name__)
```

If you run the *main.py* module with Eclipse, the module loads the module *p1.padre* and executes the function *padre.printPadre*. The module *p1.padre* loads *p1.hijo* and executes the function *hijo.printHijo* and *hijo.printNieto*. The function *printNieto* is a function of the *nieto* module, so *hijo* imports also *nieto*.

If you realise all the imports have something in common: *from p1 import something*. The question is, how Python is able to find the package *p1*. The answer is because by default Eclipse add to the Python path the project folder name (*/home/vagrant/apps/importingModules*), and inside is the package *p1*.

So the trick is to import always from the project folder, despite the modules be at the same package. That is, if *m1.py* and *m2.py* are at se same place, for example *projectFolder/p1/p2/p3*, instead of import *m2* from *m1* like this:

```
import m2
```

the right way is to specify the complete path from the PyDev project folder:

```
from p1.p2.p3 import m2
```

But if you try to execute the module *main.py* your self from the command line:

```
python3 main.py
```

you will get an import error:

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    from p1 import padre
ModuleNotFoundError: No module named 'p1'
```

This is because you are who is executing the module and not Eclipse, and is Eclipse who adds the project path to the Python path variable, so the project path is not in the Python path and the package *p1* is not found. All you have to do is to add the project path to the Python path yourself. This is only necessary once, and must be done in the project entry point, that is the module *main.py*. We will have to do the same in the file *wsgi.py* of our Django projects. So the module *main.py* code must be the following:

```
import os, sys
PROJEC_DIR=os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
)
sys.path.append(PROJEC_DIR)
print("PROJEC_DIR: " + PROJEC_DIR)

from pl import padre

print(__file__)
padre.printPadre()
```

2.3.2 Proposal exercise

You must use the IDE Eclipse, with the PyDev extension. The workspace always must be `/home/vagrant/apps`. Create a PyDev project called *prueba*. As Python interpreter you must choose the interpreter of the virtual environment located in `/home/vagrant/apps/env` (figure 2.3, figure 2.4), otherwise you will not have all the libraries available. To practice with packages and modules you must create a Python package, called *pointOperations*, and create the module *pointListOperations*. Type the following functions:

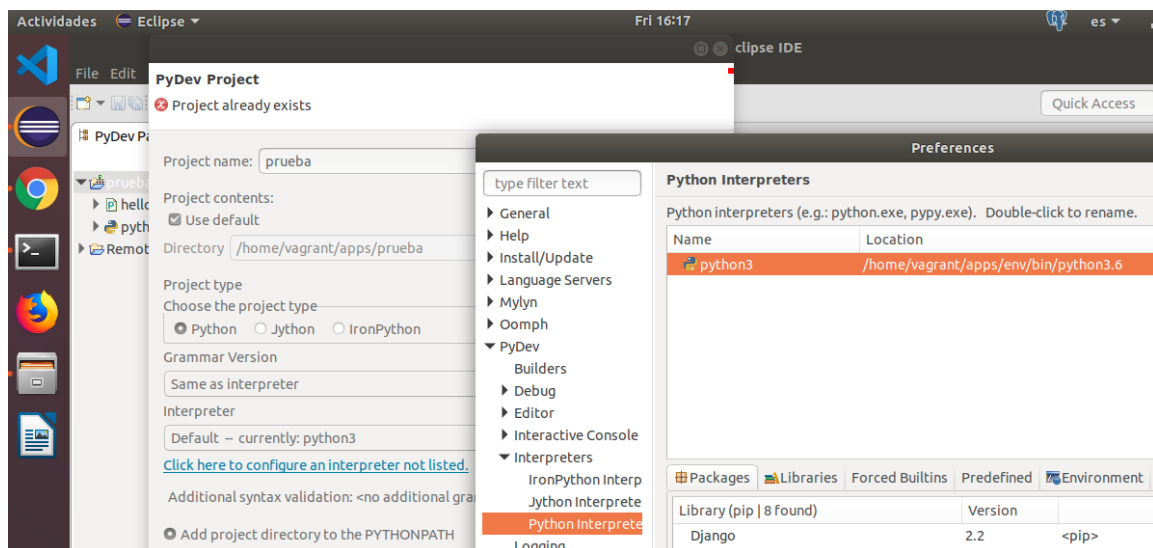


Figura 2.3: Virtualenv Python interpreter selection in Eclipse

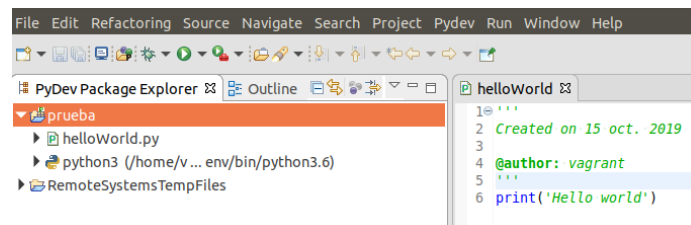


Figura 2.4: Virtualenv Python interpreter selection in Eclipse

sumXY : Receives a list of 2d points `[[X1,Y1], [X2,Y2],...]` and calculates the sum of all the X and all the Y. Returns a list `[sumX, sumY]`.

geocenterXY : Receives a list of 2d points `[[X1,Y1], [X2,Y2],...]`, and returns the a list with `[averageX, averageY]`. You must use `sumXY` in this function.

maxXY : Receives a list of 2d points `[[X1,Y1], [X2,Y2],...]` and returns a list with `[maxX,maxY]`.

minXY : Receives a list of 2d points `[[X1,Y1], [X2,Y2],...]` and returns a list with `[minX,minY]`.

traslationXY : Receives a list of 2d points `[[X1,Y1], [X2,Y2],...]` , and other list with two numbers `[dX, dY]`. The function returns a list of points moved dx and dy: `[[X1+dX, Y1+dY], [X2+dX,Y2+dY],...]`.

Create an other module, called *mainPointListOperations*, in the root project folder and execute all the *pointListOperations.py* functions.

Create another project in `/home/vagrant/apps`, called *p2*, create a module called *m2* with the following content.

```
MESSAGE = "Hello from m2 module of p2 project. It is not easy to
import me"
```

Import the module *m2* from *mainPointListOperations* and print the variable *MESSAGE* of the module *m2*. You mus not use absolute path to import.

Python challenge:

For those students who the previous exercise is easy, I propose a challenge: to do the same using *OOP* (Oriented Object Programming). Consist in to convert all the previous functions in methods without parameters in a Python class. All the methods must use the *coordList* property, set in the constructor class method. This is the way to code than probably you will be requested to use in a company. xfgdsfg

2.3.3 Solution with Object Oriented Programing (OOP)

OOP knowledge is quite important if you want to be developer, even a not professional developer, as all libraries are created using this approach. The knowledge in OOP will help you not only to arrange better your code, but also you will understand better how to use software libraries as: Django, OpenLeyers, Vue or Angular.

Algorithms are the same in lineal coding than in OOP, the difference is the organization of the code. You have to learn some more concepts, and think in a different way, but it is not that different.

The minimum concepts you have to learn are:

- How create classes.
- What is and how to use a constructor (`__init__`).
- What is *self*, that in other languages is called *this*.
- How create class variables, and the difference between public and private class variables.
- How create methods, and the difference between public and private methods.
- How to initialize the class variables from the constructor.
- How to access to the class variables from any class method.
- Difference between class variables and local variables.

- How inherit from other class and what this means.
- What is a father class, or super class, and what is a child class.
- How to initialize the constructor of the father class from the constructor of the child class.
- How to overwrite a father class method from the child class.

The main different is classes are ways fo organize data, but classes can perform much other operations: check the data before to be stored in the class and transform the data in different ways when we want to retrieve it. Therefore the class variables are accessible from all methods, so you also avoid passing parameters to the methods.

In the following example, *Point3D*, perform the following actions:

- About checking the input data:
 - Converts string into numbers.
 - Checks for negative values.
- About checking the output of the data:
 - Returns the data in several list formats.
 - Prints the coordinates in the screen.

Each 3d point can make the above operations simply with the instruction *Point3d(10,20,30)*. A list of data is created in a similar way [10,20,30], but it can not perform the same operations. A list in fact is also an object and can perform other operations as *append*, *sort*, ..., but as we can get a Poin3d as list we can do also what a list do with a Point3d.

```
class Point2d():
    """
    Variables de clase, accesibles desde todos los métodos con self.
    variable

    Las variables de clase también se pueden hacer privadas si se
    les pone el
    _ delante del nombre. Ej: _suma. Eso indicará que la variable
    _suma
    no tiene interés fuera de esta clase. Estas variables se conocen
    como variables privadas.
    """
    point_number: int = None
    x: float = None
    y: float = None
    def __init__(self,x:float, y:float, n:int=-1):
        """
        Todos los métodos de la clases reciben un primer argumento
        llamado self.
        Este argumento es obligatorio, pero es automático. No
        tenemos que pasarlo.

        El método __init__ se se llama constructor. Se le pasan los
        valores necesarios
        para inicializar las variables de clase. Ej: pt1=Point2d
        (10,20,1)
```

```
Además se aceptan valores tipo string Ej: pt1=Point2d
    ('10','20',1)
"""

#inicialización de las variables de clase
self.point_number=n
#comprobación de valores negativos
self._checkValue(x)
self._checkValue(y)
#si todo va bien se inicializan estas variables
self.x=float(x)
self.y=float(y)

def _checkValue(self,value:float):
    """
    Comprueba que el valor no es menor que cero. Si eso pasa
    genera un error
    y termina el programa.
    Se pone un guión bajo en el nombre para indicar que no tiene
    utilidad
    fuera de esta clase. Se conocen como métodos privados.
    """
    if float(value) < 0:
        mess="Coordinate values must be bigger than 0. Point
            number: {n}".format(n=self.point_number)
        raise Exception(mess)

def getXYAsList(self) -> list:
    return [self.x,self.y]
def getXYNAsList(self) -> list:
    l=self.getXYAsList()
    l.append(self.point_number)
    return l
def getAsDict(self) --> dict:
    return {'x':self.x, 'y':self.y, 'point_number': self.
            point_number}
def printAsList(self):
    print(self.getXYNAsList())

class Point3d(Point2d):
    """
    Esta clase se conoce como clase hija. Su clase padre es Point2d.
    Ha heredado todas las propiedades y métodos de la clase padre, y
    añade más cosas.
    """
    z:float=None
    def __init__(self,x:float,y:float,z:float=0,n:int=-1):
        """
        Constructor
        Se aceptan valores tipo string Ej: pt1=Point2d
            ('10','20','30',1)
        """
        Point2d.__init__(self,x, y, n)
```

```
        self._checkValue(z)
        self.z=float(z)
def getXYZAsList(self) -> list:
    return [self.x,self.y,self.z]
def getXYZNAsList(self) -> list:
    l=self.getXYZAsList()
    l.append(self.point_number)
    return l

def printAsList(self):
    """
    Este método está también en la clase padre. Al tener un
    método con el mismo
    nombre en la clase hija (esta clase) estamos
    sobrescribiendo (overrides)
    el método de la clase padre.
    Esto significa que si ejecuto printAsList() en un punto 2d,
    saldrá el resultado
    de la implementación de printAsList en Punto2d, y si ejecuto
    printAsList en
    un punto 3d saldrá el resultado
    de la implementación de printAsList en Punto3d
    """
    print(self.getXYZNAsList())

def getAsDict(self) --> dict:
    """
    También está en la clase padre (overrides)
    """
    return {'x':self.x, 'y':self.y, 'z', self.z, 'point_number':
            self.point_number}

class PointOperations():
    #variable de clase
    l:[Point3d]=None
    def __init__(self,l:list):
        #constructor
        self.l=l
    def sumXYZ(self)-> Point3d:
        #método
        sumX=0
        sumY=0
        sumZ=0
        for i in range(len(self.l)):
            sumX=sumX+self.l[i].x
            sumY=sumY+self.l[i].y
            sumZ=sumZ+self.l[i].z
        return Point3d(sumX,sumY,sumZ)

    def geocenterXY(self) -> Point3d:
        #método
        sum:Point3D=self.sumXYZ()
        n=len(self.l)
        return Point3d(sum.x/n,sum.y/n,sum.z/n)
```

```
if __name__=="__main__":

    print("Punto 2d")
    p=Point2d(10,10,1000)
    print(p.getXYAsList())
    print(p.getXYNAsList())
    p.printAsList()

    print("Punto 3d")
    p=Point3d('20','20','20',2000)
    print(p.getXYZAsList())
    print(p.getXYZNAsList())
    p.printAsList()

    l=[Point3d(10,10,10),Point3d(20,20,20), Point3d(30,30,30)]
    po=PointOperations(l)
    pt1:Point3d=po.sumXYZ()
    print("Suma")
    pt1.printAsList()
    pt2:Point3d=po.geocenterXY()
    print("Geocentro")
    pt2.printAsList()
    print("Terminado")
```

El resultado es el siguiente:

```
Punto 2d
[10.0, 10.0]
[10.0, 10.0, 1000]
[10.0, 10.0, 1000]
Punto 3d
[20.0, 20.0, 20.0]
[20.0, 20.0, 20.0, 2000]
[20.0, 20.0, 20.0, 2000]
Suma
[60.0, 60.0, 60.0, -1]
Geocentro
[20.0, 20.0, 20.0, -1]
Terminado
```

2.3.3.1. OOP proposal exercise

Create a class called *CentesimalAzimut*, with the public class property *centesimalAzimut*. The constructor receives a float , the azimuth, in centesimal mode. The constructor makes the following angle checks:

- If the centesimalAzimut is bigger than 800 the program could calculate wrong values so you should raise an error, with a proper message.
- If the centesimalAzimut is less than 0, sums 400 to the angle.
- If the centesimalAzimut is bigger than 400 then centesimalAzimut=centesimalAzimut-400.
- In none of the previous cases occur, the centesimalAzimut values is correct and must remain unchanged.

All those checks can be in the constructor, or in a public method that sets the proper value of the azimuth in the class variable. Better call a public method called *setCentesimalAzimut*, that receives the angle passed to the constructor make the previous checks, and sets proper azimuth value in the class variable *centesimalAzimut*. I recommend make the *setCentesimalAzimut* method public because, in that way can be used to change the azimuth of the object, at any moment. That is, if I have *ca=CentesimalAzimut(560)*, latter I can change its azimuth: *ca.setCentesimalAzimut(230)*, and the new azimuth for the *ca* object will be also checked.

Create the following public class methods: *getAsSexagesimal*, *getAsRadians* and *getAsCentesimal*, to return the azimuth in several formats:

- The method *getAsCentesimal* simply returns the class property *centesimalAzimut*.
- The method *getAsSexagesimal* gets the *centesimalAzimut* property multiplies by 360 and divides by 400, and returns the result.
- The method *getAsRadians* gets the *centesimalAzimut* property multiplies by *math.PI* and divides by 200, and returns the result.

I have to remark that the *centesimalAzimut* property must remain unchanged after the initialization, and the public methods simply transform the value internally and return the value that the user requested. You must aware that, if you change the *centesimalAzimut* value from any of the methods, the second time you use any of the methods will return a wrong value.

Create the class *Observation2d*, that inherits from *CentesimalAzimut*, and adds the public property *distance2d*. The constructor receives the azimuth, and the distance, both supposedly observed from a base point to a measured point with a total station. The idea is to have the *CentesimalAzimut* functionalities to check the observed azimuth, and add some more checks to the distance, all in the same class. The idea of inheritance is to be enlarging functionality in inherited classes.

The child class *Observation2d* initializes the father class, *CentesimalAzimut*, with the azimuth received by the constructor of *Observation2d*. *Observation2d* also receives a distance in its constructor. You must perform also some checks: if the distance is bigger than 0, is correct, and initializes the distance. In case the distance be less than 0 you should raise an error. Again you can implement this distance checks directly in the constructor, or outside in a private method. This time that is up to you.

Create a public method called *getDistance*, which returns the distance property.

Create a public method called *getDistanceWithOffset*, that receives an offset, and returns the distance plus the offset. If there is a systematical error in the distance, positive or negative, with the offset you can get the proper distance value. I have to recognize that this is a very strange case, but it is useful to you to know that class methods can receive extra values, to make the class method work.

Why do not put the offset variable in the *Observation2d* constructor, and store it as a class property or variable?. Good question. It is normal at the beginning not to know where is the best place to put the things. It is a matter of practice. To create a class variable, instead of require a class method parameter, you must the following two criteria:

- The property must be required for all class methods. That is, it is a common required data for all methods.
- The property must be frequently used.

In this case the class *Observation2d* has three inherited methods and his own two ones, *getDistance*, and *getDistanceWithOffset*. So one out of five require the offset. First condition is not accomplished.

But also the offset is going to be rarely used. So make sense not use it as property, and use it as parameter to get the corrected distance value with the `getDistanceWithOffset` method, if required.

Therefore, if you do not add the offset as class property, the initialization of the class will receive two parameters. e.g: `o = Observation2d(distance=200, azimuth=150)`.

But if you add the offset as class property, the initialization of the class will need three parameters: `o = Observation2d(distance=200, azimuth=150, offset=0 (or 0.05, or 0.025, whatever value))`, although you can set the offset as optional value.

Create the class `Radia2d`. That class do not inherits anything from the other classes, because it is not going to extend their functionality, but is going to use objects of previous classes, in fact is going to use a `Point2d` object, and an `Observation2d` object. Define three public class variables outside of the constructor, above, and set them to `None`: `base: Point2d`, `obs: Observation2d` and `radiatedPt2d: Point2d`.

Define the constructor that receives two parameters `base: Point2d`, `obs: Observation2d`. The constructor immediately initializes the class variables `base` and `obs`. To initialize the `radiatedPt2d` you must calculate the x, and y coordinates and create a `Point2d` object, and set the `radiatedPt2d` to that object. I recommend you to do that in a private method, outside the constructor, e.g: `_setRadiatedPt2d`.

Note: Methods that gets property values are usually named `getPropertyName`. Methods that sets property values are commonly named `setPropertyName`. They are also named as *getters* and *setters* methods.

Define the following methods to get the results: `getAsPoint2d` (returns the radiated point), `getAsList` (returns the coordinates of the radiated point as list `[x, y]`), `getAsDict`(returns the resulting point as dict), `printPoint` (prints the radiated point in list form).

2.3.3.2. OOP proposal exercise solution

Here is the solution:

```
import math
from pointOperationsObj import Point2d

PI = math.pi#global variable

class CentesimalAzimut():
    centesimalAzimut: float = None#class variable
    def __init__(self, centesimalAzimut:float):#constructor
        self.setCentesimalAzimut(centesimalAzimut)#setter de la
            propiedad centesimalAzimut

    def setCentesimalAzimut(self, value:float):
        """
        Setter de centesimalAzimut. Establece el valor correcto de
        la propiedad
        """
        if value > 800:
            raise Exception("Angulo mayor de 800")
        if float(value) < 0:
            self.centesimalAzimut = value+400
        else:
            if float(value) > 400:
                self.centesimalAzimut = value-400
            else:
                self.centesimalAzimut = value

#getters
```

```
def getAsSexagesimal(self):
    return (self.centesimalAzimut*180)/200

def getAsRadians(self):
    return (self.centesimalAzimut*PI)/200

def getAsCentesimal(self):
    return self.centesimalAzimut

class Observation2d(CentesimalAzimut):
    distance2d : float = None

    def __init__(self, centesimalAzimut:float, distance:float):
        CentesimalAzimut.__init__(self, centesimalAzimut)
        self.__checkPositiveDistance(distance)
        self.distance2d = float(distance)

    def __checkPositiveDistance(self, value:float):
        if float(value) <= 0:
            mess="Distance must be bigger than 0"
            raise Exception(mess)

    def getDistance(self):
        return self.distance2d

    def getDistanceWithOffset(self, offset:float = 0):
        return self.distance2d+float(offset)

class Radia2d():
    base:Point2d=None
    obs:Observation2d=None
    radiatedPt2d:Point2d=None

    def __init__(self, base:Point2d, obs:Observation2d):
        self.base = base
        self.obs = obs
        self.setRadiatedPt2d()

    def setRadiatedPt2d(self):
        x=self.base.x + self.obs.distance2d*math.sin(self.obs.
            getAsRadians())
        y=self.base.y + self.obs.distance2d*math.cos(self.obs.
            getAsRadians())
        self.radiatedPt2d=Point2d(x,y)

    def getAsPoint2d(self):
        return self.radiatedPt2d
    def getAsList(self):
        return self.radiatedPt2d.getXYAsList()
    def getAsDict(self):
        return self.radiatedPt2d.getAsDict()
    def printPoint(self):
        self.radiatedPt2d.printAsList()
```

```
#demostración de uso
if __name__=="__main__":

    #Comprobación de Observation2d
    obs = Observation2d(centesimalAzimut=500, distance=200)
    print("Angulo en centesimal: {0}".format(obs.getAsCentesimal()))
    print("Distancia: {0}".format(obs.getDistance()))
    print("Distancia con -0.025 de offset: {0}".format(obs.
        getDistanceWithOffset(offset=-0.025)))

    #Radiación de un punto
    ptBase=Point2d(100,100)
    rd=Radia2d(base=ptBase, obs=obs)
    print("Radiación primer punto:")
    rd.printPoint()

    ptBase2=Point2d(800,800)
    obs2=Observation2d(centesimalAzimut=300, distance=450)
    rd2=Radia2d(base=ptBase2, obs=obs2)
    print("Radiación segundo punto:")
    rd2.printPoint()
```

The result of the before program is:

```
Angulo en centesimal: 100
Distancia: 200.0
Distancia con -0.025 de offset: 199.975
Radiación primer punto:
[300.0, 100.00000000000001, -1]
Radiación segundo punto:
[350.0, 799.9999999999999, -1]
```

2.3.4 Python string formatting

In this section you are going to learn an utility that you are going to use on making sql sentences dynamically. It is a way to substitute variable values inside a string. Very useful to introduce table names and field names into the sql sentences. Python give as two ways to to this:

2.3.4.1. Formatting a string using the order of the variables

In this way you put numbers between curly brackets into the string, an later you specify the variable values in order:

```
query1 = "select {0} from {1}".format("gid, description,st_area(geom)", "d.buildings")
print(query1) #print --> select gid, description,st_area(geom) from
d.buildings
```

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

2.3.4.2. Formatting a string using names in the formatted string

In this way you put names between curly brackets into the string, and later you specify the variable names and values:

```
query2 = "select {fieldNames} from {tableName}".format(fieldNames="
    gid, description, st_area(geom)", tableName="d.buildings")
print(query2) #print --> select gid, description, st_area(geom) from
    d.buildings
```

2.3.4.3. Proposal exercise

Create a python function that receives the strings and returns one. Receives the a table name and the geometry field name of a table and returns the following string: update tableName set area = st_area(geomFieldName), where tableName and geomFieldName are substituted by the real variable values received by the function.

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

In this section you will learn how to connect with a database and how to perform any sql query. To make queries or changes in a PostgreSQL database with Python, we are going to use the *psycopg2* library, <http://initd.org/psycopg/>.

First step is to create a database, add the postgis extension and create an schema and a table. You can do this also with Python but you will do in this occasion with the console. You can use the user the linux user *vagrant* without password to create databases and to connect to databases because is also a superuser of postgres, with password *vagrant*. Open an console and type the following:

```
vagrant@linux:~$ createdb desweb
vagrant@linux:~$ psql -d desweb
psql (11.5 (Ubuntu 11.5-3.pgdg18.04+1))
Digite «help» para obtener ayuda.

desweb=# create extension postgis;
CREATE EXTENSION
desweb=# create schema d;
CREATE SCHEMA
desweb=# create table d.buildings(gid serial primary key,
    descripcion text, area double precision, geom geometry("POLYGON",
    25830 ));
CREATE TABLE
desweb=#
```

You can see the result in the figure 2.5.

Now you can insert, delete, select or update buildings with Python and Psycopg2 (<http://initd.org/psycopg/>). The workspace must be */home/vagrant/apps*. Launch Eclipse and create a PyDev project called *desweb*. As interpreter you must choose the interpreter of the virtual environment located in */home/vagrant/apps/env*.

The first you have to do is to configure the psycopg2 library, because our databases are in UTF8 coding. Create a module called *buildingsInsert.py* and write the following:

```
import psycopg2
import psycopg2.extensions
psycopg2.extensions.register_type(psycopg2.extensions.UNICODE)
psycopg2.extensions.register_type(psycopg2.extensions.UNICODEARRAY)
```

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

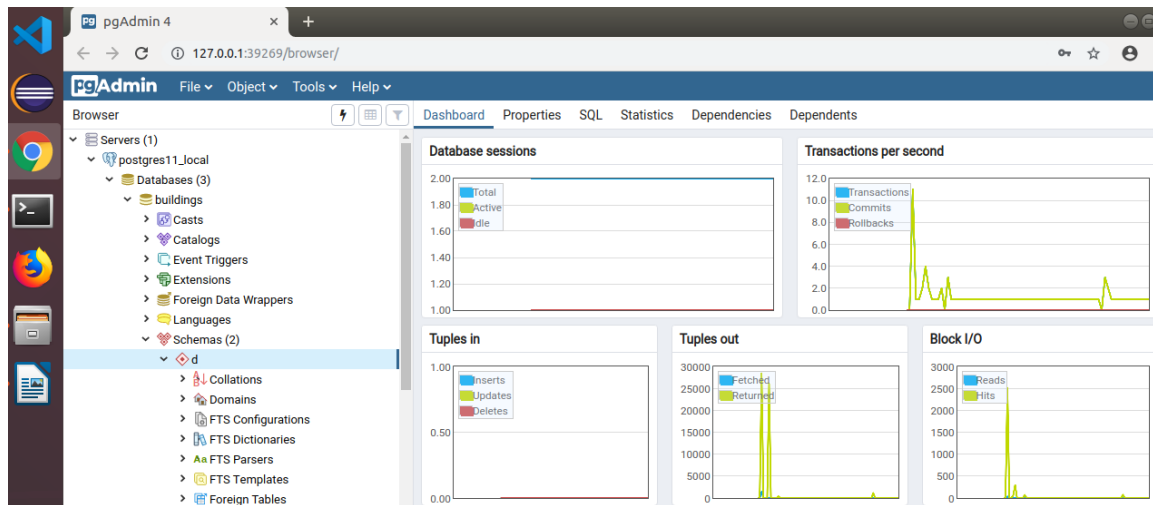


Figura 2.5: PgAdmin 4 showing the buildings database

Now you can connect with the *desweb* database:

```
database="desweb"  
user="postgres"  
password="postgres"  
host="localhost"  
port=5432  
  
#conexion  
conn=psycopg2.connect(database=database, user=user, password=  
    password, host=host, port=port)  
cursor=conn.cursor()
```

The constructor of the object *connect* created a object, which has been denominated *conn*. This object has a property, called *cursor*, which is an other object which allow to send SQL queries to the database.

```
cursor=conn.cursor()
```

With the *cursor*, it is already able to send SQL sentences to the database. The sentences can have parameters or not. If the sentence does not have parameters then is a normal string.

VERY, VERY, VERY IMPORTANT:

AFTER TO HAVE FINISHED WITH THE CURSOR AND CONNECTION, IT IS TOTALLY FUNDAMENTAL TO CLOSE THEM:

```
cursor.close()  
conn.close()
```

IF YOU DO NOT CLOSE THE CONNECTION, THE CONNECTIONS CONTINUE OPENED, AND THERE ARE A DEFAULT NUMBER OF CONNECTION OF 100. IN FEW MINUTES YOU WILL DO POSTGRES UNUSABLE FOR ANY OTHER USER OR PROJECT. ANY MORE WILL BE ABLE TO CONNECT TO THE DATABASE, UNTIL THE POSTGRES SERVICE BE RESTARTED.

If the cursor, or the connection is closed, you can not send any sql query.

2.4.1 Insert rows with geometry

In the following listing, a form of to insert geometries in tables with a field geometry is presented. The trick consists in to use the PostGIS function *st_geometryfromtext*, and transform the geometry, in a WKT format, to a valid column geometry type. The previous function also needs the SRC, specified in the EPSG code. Lets insert some buildings:

```
query_ins="insert into d.buildings (descripcion, geom) values (%s,
    st_geometryfromtext(%s,25830)) "

values1=["edificio 1", "POLYGON((727844 4373183,727896
    4373187,727893 4373028,727873 4373018,727858 4372987,727796
    4372988,727782 4373008,727844 4373183, 727844 4373183)) "]
values2=["edificio 2", "POLYGON((727988 4373188,728054
    4373192,728051 4373093,727983 4373093,727988 4373188)) "]

cursor.execute(query_ins, values1)
cursor.execute(query_ins, values2)

cursor.commit()

cursor.close()
conn.close()
```

We are going to use WKT format because OpenLayers is able to give its geometries in this format. The WKT representation of a polygon geometry is:

```
"POLYGON((727844 4373183,727896 4373187,727893 4373028,727873
    4373018,727858 4372987,727796 4372988,727782 4373008,727844
    4373183, 727844 4373183)) "
```

As you can see, the same query is executed twice with different parameter values. The parameter values are the included in the lists *values1* and *values2*. The executed method changes each %s by each parameter value. The first %s is replaced in the query for the first parameter value in the list, the second %s is replaced in the query for the second parameter value in the list, and so on.

The *cursor.commit* function perform the changes into the database. Without the commit any change will be done into the database, despite the code be correct.

The last thing is to close the cursor and the connection, in that order. To execute the program, right button over de code and select *Run as →Python Run* (figure 2.6). Execute the program and see the polygons in PgAdmin (figure 2.7), and in Qgis (figure 2.8).

To connect to Postgis with Qgis you need to define a connection the figure 2.9 shows the proper parameters in this case.

Now the student already know how to insert data into a database using *Psycopg2*. The same process is for any other sql (update, delete, select, ...) query. The only difficult here is to know what sql sentence to use. You only have to create the sql sentence in a string, putting %s instead of the real values, and use the *execute* method giving that string and the real values in a list.

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

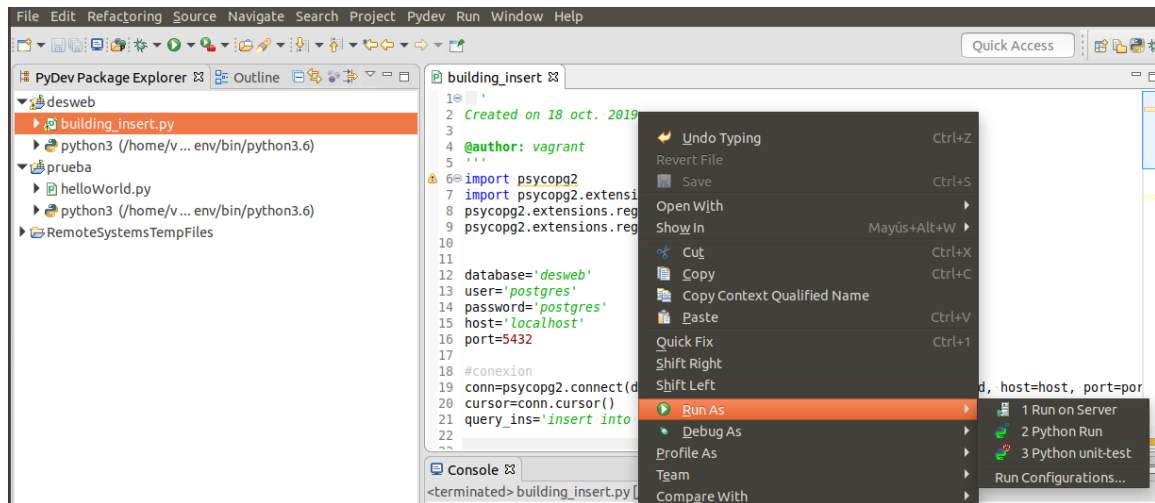


Figura 2.6: Execute a Python program in Eclipse

2.4.1.1. Proposed exercise: insert

Create a database called *training* and add the postgis extension. Create a table, inside the *training* database, called *streets*. Put the table in the schema *d*. Add three fields to the table: *gid*, *name*, *length* and *geom*. The field *gid* must be serial primary key, *name* a varchar and *geom* *linestring*, in the EPSG 25830. The *length* field must be of double precision.

Create an Eclipse PyDev project called *training*, and select the interpreter located in the virtualenv. Create a module called *streetsInsert.py* and introduce the code necessary to insert a couple of streets, in EPSG 25830 coordinates. The streets coordinates must be digitized from the cadastre, in an area near the Universitat Politècnica de València.

To get the street coordinates in 25830 you must set the SRC of the project in Qgis (settings→Options→SRC). Set the EPSG 25830 by default restart Qgis and create a new project. Later you must add the Spanish WMS service (<http://ovc.catastro.meh.es/Cartografia/WMS/ServidorWMS.aspx?>). Create a *linestring* shp layer (Layer → Create layer → New shp layer). Locate the university. Start the edition of the layer and draw 4 streets.

To get the WKT format of the digitized streets you must select a street and press the button *WKT* (figure 2.10).

2.4.2 Update rows

In the Eclipse project *desweb* create a new module called *buildingUpdate.py*, and paste the following code:

```
@author: vagrant
"""
import psycopg2
import psycopg2.extensions
psycopg2.extensions.register_type(psycopg2.extensions.UNICODE)
psycopg2.extensions.register_type(psycopg2.extensions.UNICODEARRAY)

database="desweb"
user="postgres"
password="postgres"
selecthost="localhost"
```


2.4 Database connection with Python 3.6. Use of the *psycopg2* library

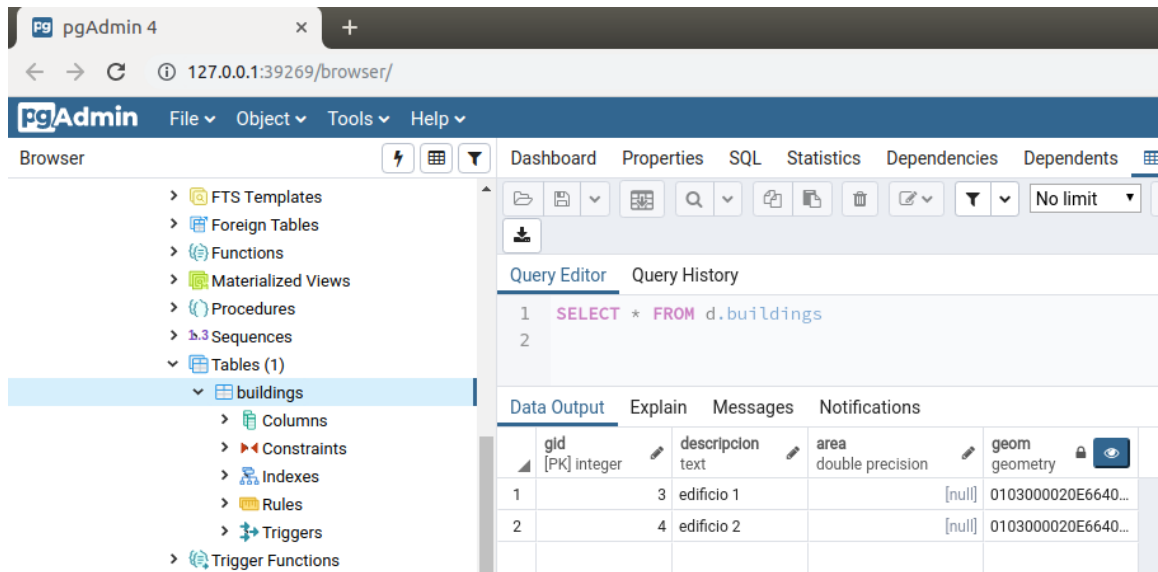


Figura 2.7: The two buildings inserted seen in PgAdmin

```
port=5432

#conexion
conn=psycopg2.connect(database=database, user=user, password=
    password, host=host, port=port)
cursor=conn.cursor()
query_update="update d.buildings set (descripcion, area, geom) = row
    (%s, st_area(geom), st_geometryfromtext(%s,25830)) where
    descripcion = %s"

values=["Edificio 2",
    "POLYGON((727988 4373188, 728054 4373192,
    728095.842977791088465601 4373142.83781164418905973,
    728051 4373093, 727983 4373093, 727988 4373188))",
    "edificio 2"
    ]

cursor.execute(query_update, values)

n = cursor.rowcount

conn.commit()

cursor.close()
conn.close()

print("Polygons updated: {0}".format(n))
```

As you can see, the update query has three %s, so the values list has to have also three values, the last of which is taken for the where condition. Only the building who *descripcion* is *edificio 2* is going to be update.

The number of rows affected is stored in the *cursor.rowcount* property.

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

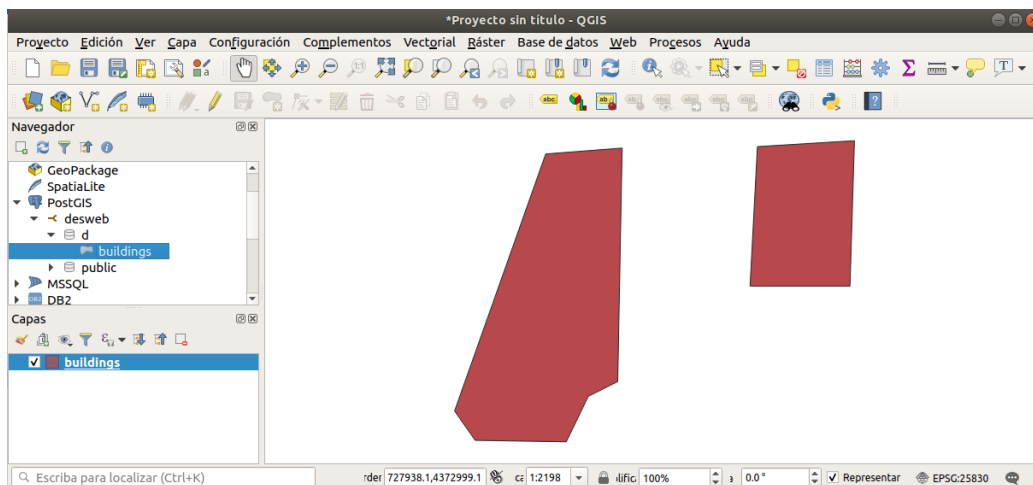


Figura 2.8: Inserted buildings in Qgis

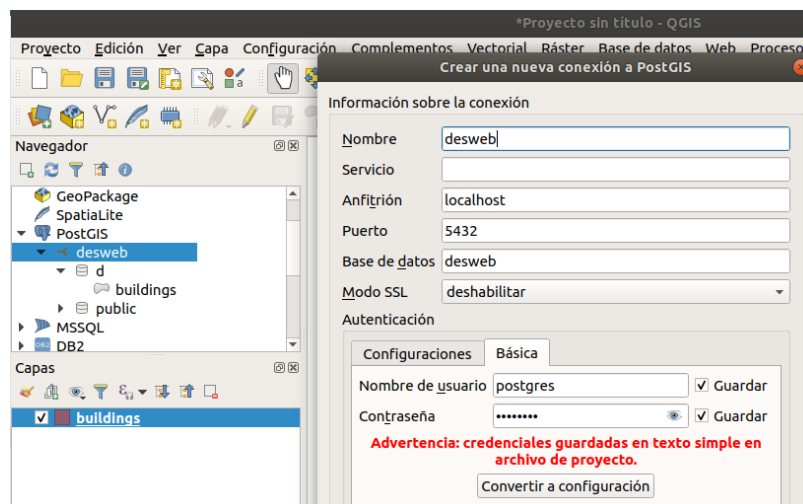


Figura 2.9: Postgis connection definition in Qgis

2.4.2.1. Proposed exercise: update

Create a module to update one street of the *cities* database. Update also the *length* field.

2.4.3 Code optimization

We have two modules and we already have repeated a lot of code. For example, the database name, user, etc are already in two files. If we continue like that with the two next operations, *delete* and *update*, we will have the same 4 times. What will happen if we change the user password?. We will have to change the password in 4 files. NEVER REPEAT CODE. IT IS A ILLNESS OF YOUR APP.

Not to repeat code allows you to change things in one point in your code, but has a price. You have to extract the common code to an other module, and import it whatever is needed. You have to know how the imports work in Python.

In the current project I will extract the database credentials to an external module, called *mySettings.py*. I also will create two functions to connect and disconnect from the database, and put it into other module called *pgUtils.py*.

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

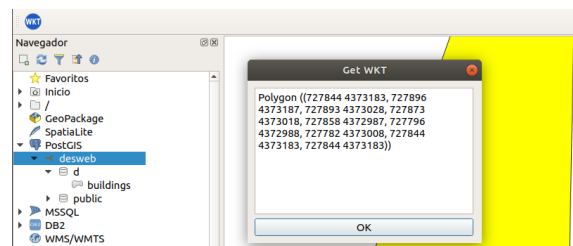


Figura 2.10: Get the WKT coordinates with Qgis

This is the content for the module *mySettings.py*:

```
DATABASE="desweb"  
USER="postgres"  
PASSWORD="postgres"  
HOST="localhost"  
PORT=5432
```

It is used to use upper letters to name the global variables

This is the code for the module *pgUtils.py*. Pay attention to the comments, the most important think here is that from *pgUtils.py* the module *mySettings.py* is imported, and they are in different packages. To be able to import one module from the other it is necessary to add the project root path to the *sys.path* list. The *sys.path* list contains the list of paths to search packages or modules.

```
import os, sys  
  
import psycopg2  
import psycopg2.extensions  
psycopg2.extensions.register_type(psycopg2.extensions.UNICODE)  
psycopg2.extensions.register_type(psycopg2.extensions.UNICODEARRAY)  
  
import mySettings #this works because mySetting and all the modules  
                  #in this project are at the same folder, the  
                  #project root folder  
  
def pgConnect():  
    """  
    Connects with the database with the library psycopg2  
    """  
    #conexion  
    conn=psycopg2.connect(database=mySettings.DATABASE, user=  
        mySettings.USER,  
                           password=mySettings.PASSWORD, host=  
                           mySettings.HOST,  
                           port=mySettings.PORT)  
  
    return conn  
  
def pgDisconnect(conn):  
    cursor=conn.cursor()  
    cursor.close()  
    conn.close()
```

2.4.4 Delete rows

In the Eclipse project *desweb* create a new module called *building_delete.py*, and paste the following code:

```
import pgUtils #this works because pgUtils and all the modules
                #in this project are at the same folder, the
                #project root folder
conn = pgUtils.pgConnect()
cursor=conn.cursor()
query_delete='delete from d.buildings where descripcion = %s'

values=['Edificio 2']

cursor.execute(query_delete, values)

n = cursor.rowcount

conn.commit()
pgUtils.pgDisconnect(conn)
```

As you can see, you do not repeat code unnecessarily.

2.4.4.1. Proposed exercise: optimize the modules *building_insert.py* and *building_update.py*

Now that you have the module *pgUtils.py*, you can use it in the modules *building_insert.py* and *building_update.py*, so that you do not repeat code. Make the necessary changes and test the new versions.

2.4.5 Select rows

To select rows is a little bit more tricky because we want to get the rows in json format. This because we are going to transfer the data from the database to a web page, in the client side, in json strings, and the same on the reverse. So we have to learn to deal with json and json strings.

Lets start adding a couple of more buildings to the table. Open Qgis and add the WMS service of the Spanish cadaster, connect to the database *desweb*, and add the table *d.building* to the canvas. Press the edit button and the add feature button. Digitize a couple of blocks figure 2.11. Save the changes and press the edit button again to stop the editing.

2.4.5.1. Select rows

The following selects the fields *gid*, *descripcion* and *geom* of all rows of the table. The *geom* field can not be selected directly because its stored in binary format. To get the geometry you can use the postgis function *st_astext*, or *st_asgeojson*. These functions returns the geometry in an adequate readable format:

st_astext : This is the format we are going to use in the HTML forms to send geometries to the server. So we can use this format also to send the geometries to the HTML form and edit the coordinates in the form.

st_asgeojson : This format is useful because OpenLayers is able to draw geometries in the map with this format. This has the advantage of avoid to use GeoServer and to be able to edit the geometries graphically in the map, using the mouse. The inconvenient is that the symbology available in OpenLayers is much more limited than in GeoServer, and, if there are millions of geometries, the performance will be slow, so many system limit the geometries transfer to 1000.

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

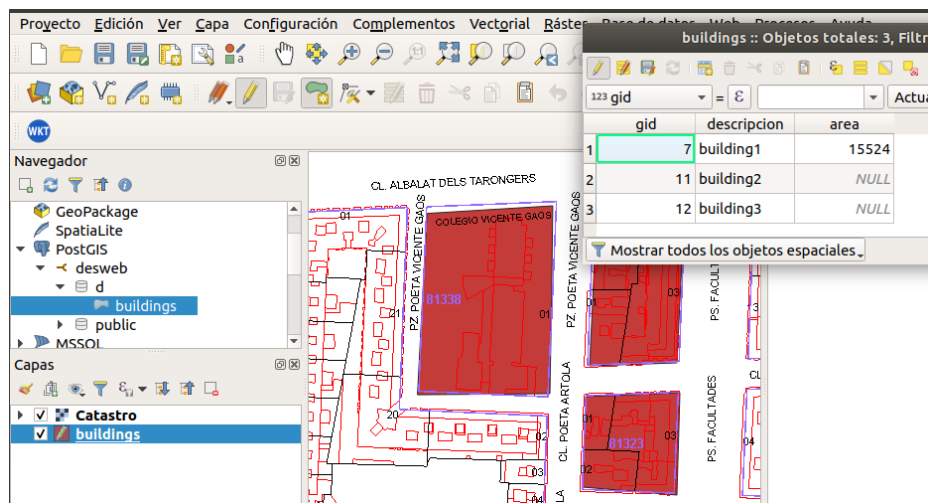


Figura 2.11: Add some more buildings, digitizing blocks from the Spanish cadastre

```
import pgUtils
conn = pgUtils.pgConnect()
cursor=conn.cursor()
query_select="select gid, descripcion, st_astext(geom) from d.
    buildings"
cursor.execute(query_select)
listOfRows=cursor.fetchall()
print(listOfRows)
pgUtils.pgDisconnect(conn)
```

The *execute* function with a select query does not have any result. The results are put into the cursor object. The cursor selects the rows and puts them into the cursor. To extract them you can use the function *fetchall*. The function *fetchall* removes the rows from the cursor, so they are not in the cursor object any more.

The above listing has the following result:

```
[ (7, "building1", "POLYGON((727988 4373188,728054
4373192,728053.399856473 4373143.69857413,728052.488432421
4373093.57025125,727983 4373093,727988 4373188))" ),
(11, "building2", "POLYGON((728077.487492143
4373196.17055886,728123.319101633
4373195.64974511,728119.152591679
4373109.71547732,728073.320982189
4373108.15303608,728077.487492143 4373196.17055886))" ),
(12, "building3", "POLYGON((728069.67528598
4373044.61375929,728071.237727212
4373092.00781002,728120.715032912
4373094.09106499,728119.152591679
4373040.44724934,728069.67528598 4373044.61375929))" ) ]
```

As you can see in the above listing, the result is a list of tuples, each of which represents a row. Each row has the field values, by order specified in the select query.

The access to the values is like in Python normal lists. For example, to print only the *descripcion* field you can the for sentence and access to the second position in each row (index 1):

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

```
for row in listOfRows:
    print(row[1])
#result:
#   building1
#   building2
#   building3
```

If you want to filter the rows you have to use the *where* sql clause and use %s to mark the position of the variable value, exactly equal than you updated a row. The following listing selects only the building who *descripcion* is *building2*.

```
query_select="select gid, descripcion, st_astext(geom) from d.
    buildings where descripcion = %s"
cursor.execute(query_select, ["building2"])
```

2.4.5.2. Select rows getting a dictionary for each row

In this section we are going to get a dictionary *fieldName : fieldValue* for each row. Also we are going to use *st_asgeojson* to get the geometry values. This is the way that you have to use in web developing:

```
#Select all the rows. Get a dictionary for each row, and use
    st_asgeoson
query_select="SELECT array_to_json(array_agg(registros)) FROM (
    select gid, descripcion, st_asgeojson(geom), st_astext(geom) from
    d.buildings as t) as registros"
cursor.execute(query_select)
listOfRows=cursor.fetchall()
print(listOfRows)
```

The result is the following:

```
[
(
{'gid': 7, 'descripcion': 'building1', 'st_asgeojson': '{"type": "
Polygon", "coordinates
": [[[727988, 4373188], [728054, 4373192], [728053.399856473, 4373143.69857413], [72
' st_astext': "POLYGON(X Y, X Y, ...)"}},
{'gid': 11, 'descripcion': 'building2', 'st_asgeojson': '{"type": "
Polygon", "coordinates
": [[[728077.487492143, 4373196.17055886], [728123.319101633, 4373195.64974511], [7
' st_astext': "POLYGON(X Y, X Y, ...)"}},
{'gid': 12, 'descripcion': 'building3', 'st_asgeojson': '{"type": "
Polygon", "coordinates
": [[[728069.67528598, 4373044.61375929], [728071.237727212, 4373092.00781002], [72
' st_astext': "POLYGON(X Y, X Y, ...)"}
],)
]
```

As you can see, the rows are inside a list, inside a tuple, inside a list. To extract the list of rows we have to get the element index 0, of the element of index 0 of the result. You have simply to add `[0][0]` after the `fetchall()`:

2.4 Database connection with Python 3.6. Use of the *psycopg2* library

```
#Select all the rows. Get a dictionary for each row, and use
st_asgeoson
query_select="SELECT array_to_json(array_agg(registros)) FROM (
    select gid, descripcion, st_asgeojson(geom) from d.buildings as t
    ) as registros"
cursor.execute(query_select)
listOfRows=cursor.fetchall()[0][0]
print(listOfRows)
```

The result is the following:

```
[
    {'gid': 7, 'descripcion': 'building1', 'st_asgeojson': '{"type": "
    Polygon", "coordinates
    ": [[[727988, 4373188], [728054, 4373192], [728053.399856473, 4373143.69857413], [72
    'st_astext': "POLYGON(X Y, X Y, ...)"}},
    {'gid': 11, 'descripcion': 'building2', 'st_asgeojson': '{"type": "
    Polygon", "coordinates
    ": [[[728077.487492143, 4373196.17055886], [728123.319101633, 4373195.64974511], [7
    'st_astext': "POLYGON(X Y, X Y, ...)"}},
    {'gid': 12, 'descripcion': 'building3', 'st_asgeojson': '{"type": "
    Polygon", "coordinates
    ": [[[728069.67528598, 4373044.61375929], [728071.237727212, 4373092.00781002], [72
    'st_astext': "POLYGON(X Y, X Y, ...)"}
]
```

As in this case each row is a Python dictionary, to access to the *descripcion* field of each row:

```
for row in listOfRows:
    print(row["descripcion"])
```

If you want to use the *where* clause to filter the result, in the following example you have an example where you only retrieve one row:

```
#Select one row. Get a dictionary for each row, and use st_asgeoson
query_select="SELECT array_to_json(array_agg(registros)) FROM (
    select gid, descripcion, st_asgeojson(geom) from d.buildings as t
    where descripcion = %s) as registros"
cursor.execute(query_select, ["building2"])
listOfRows=cursor.fetchall()[0][0]
print(listOfRows)
```

The result is the following:

```
[
    {'gid': 11, 'descripcion': 'building2', 'st_asgeojson': '{"type": "
    Polygon", "coordinates
    ": [[[728077.487492143, 4373196.17055886], [728123.319101633, 4373195.64974511], [7
]
```

2.4.5.3. Proposed exercise

Open the *training* project and add a new module called *streetsSelect*. Type a function to select one street using his name. Get all the table fields, and get the geometry field in geojson format, and in WKT format. The row selected must be a Python dictionary.

2.5 Managing JSON strings

2.5.1 Communication client - server

A server only receives and send strings. It is widely used JSON strings to send data from the server to the user, and from the user to the server. Usually the server sends to the user json strings containing table rows and the user sends json strings containing form user data.

```
COMMUNICATION CLIENT --> SERVER

CLIENT: html form --> javascript --> json --> send to the server
||Internet|| SERVER: python WSGI --> json decode --> python
dictionary --> send to the database

COMMUNICATION SERVER --> CLIENT
DATABASE --> PYTHON WSGI --> Python dictionary --> json --> send to
the client
||Internet|| CLIENT: json --> js object --> html form
```

A json string is a string that contains fields names and values. It looks like a Python dictionary but wrapped by simple quotes. Json strings can easily be transformed into Python dictionaries and vice versa. In the following listing you can see a json string example. Pay attention in the external quotes and the internal quotes format.

```
'{"gid": "10", "descripcion": "Hola", "id_trabajo": "1", "z_tapa": "10", "
  profundidad": "10", "diametro": "10", "type": "Point",
  "coordinates": "727763.05556, 4372987.48466}"'
```

2.5.2 Creating JSON strings with Python

In the following listing you can find an example of how create a json string with Python.

```
# -*- coding: utf-8 -*-

#imports the json librari to code and decode json strings
import json

#creates the dictionary d
d = {}

#introduces three keys and values
d["campo1"]="valor1"
d["campo2"]="valor2"
d["campo3"]="valor3"

#converts the dictionary to a json string
d_json = json.dumps(d)
print("First json: {0}".format(d_json))

#creates other dictionary
```



```
d2={}
#introduces some keys and values
d2["ok"]=True
d2["menssage"]="diccionariy 2"
#introduces the other json string as a value in this dictionary
d2["data"]=d
d2_json = json.dumps(d2)

print("Second json: {0}".format(d2_json))
```

The result is the following

```
First json: '{"campo1": "valor1", "campo2": "valor2", "campo3": "
valor3"}'
Second json: '{"message": "diccionariy 2", "ok": true, "data": {"
campo1": "valor1", "campo2": "valor2", "campo3": "valor3"}}'
```

As you can see in the in the second json of the above listing, you can put a dictionary into a dictionary an generate a json. This is what usually is done. In the second json there is a message, a variable to know if the request was *ok*, and a field called *data*. The data field contains, in this case, a row, but can contain several rows.

2.5.3 Decode a JSON string to obtain a Python dictionary

In the following listing you can find an example of how decode a json string with Pyhton.

```
# -*- coding: utf-8 -*-

#imports the json librari to code and decode json strings
import json

#the json string to decode
strJson = '{"message": "diccionariy 2", "ok": true, "data": {"campo1
": "valor1", "campo2": "valor2", "campo3": "valor3"}}'
#decodes the json string and creates a dictionary
d=json.loads(strJson)

#prints the keys of the dictionary
print(d.keys())
#print the values of the dictionary
print(d.values())
```

The result is the following. Pay attention in that the value for the *data* key is an other dictionary. That means that you can put several dictionaries inside other dictionaries, obtain a json string, and decoding the json string only once, you can retrieve all the dictionaries.

```
dict_keys(['message', 'ok', 'data'])
dict_values(['diccionariy 2', True, {'campo1': 'valor1', 'campo2': '
valor2', 'campo3': 'valor3'}])
```

2.6 Create functions to easily insert, delete update and select buildings

2.5.4 Decode a JSON string to obtain a JavaScript object

If a json string is received in the client, with JavaScript it is possible to convert it into an object. In the following code, an example is showed:

Listado 2.1: Convert json string to object in JavaScript

```
function functionWhichReceivesTheServerAnswer(resp_json){
    //resp_json is:
    // '{"message": "diccionariy 2", "ok": true, "data": {"campo1":
    // "valor1", "campo2": "valor2", "campo3": "valor3"}}'
    var obj_resp_json=$.parseJSON(resp_json);//$ mins JQUERY. It is
    necessary to load JQUERY library before
    alert(obj_resp_json.ok);//shows true
    alert(obj_resp_json.message);//shows the message "diccionariy 2"
    alert(obj_resp_json.data);//shows the json data, which will be a
    new json to decode again:
                                '{"campo1": "valor1", "campo2": "
                                valor2", "campo3": "valor3"}'
}
```

2.6 Create functions to easily insert, delete update and select buildings

You know how to insert, delete, update and select rows using Python and Psycopg2. You also know that we are going to send and receive json strings from the server to the client and vice versa.

To be able to communicate in that way, we need useful functions that receive a dictionaries string, decode the string and get a Python dictionary, perform the requested action (insert, delete, update or select), encode a json with the result and return this json. Django is in charged to decode the json string of the request and encode to json string the answer.

In this section you are going to get module, called *buildings* with four functions: insert, select, delete and update. All these functions will receive a dictionary, perform the operation and return a dictionary with the result. You will have to do the same with each of your database tables, at least three of them, to pass the evaluable exercise 1. So the *buildings* module can be taken as a model for your table operations.

Create a new module in the Eclipse *desweb* project called *buildings*.

2.6.1 Function to insert a building from a Python dictionary

It is necessary to have a function that receives a dictionary with the fields *descripcion* and *geomWkt* and the function inserts the building. The WKT representation of a geometry is a format that OpenLayers can generate. We will use this to send the geometries drawn in the map to the database. In the following listing you have a WKT representation of a polygon geometry:

```
POLYGON((726151.9523572231 4374499.933133851,724021.1074695279
4373503.494877015,724802.9282556607
4371571.9376406865,726627.1767566373
4372874.972284242,726151.9523572231 4374499.933133851))
```

In the next listing is the code:

```
import json
from libs import pgUtils
#example of jsonString
#jsonString= '{
#             "descripcion": "edificio 1",
```

2.6 Create functions to easily insert, delete update and select buildings

```
#         "geomWkt": "POLYGON((726151.9523572231
4374499.933133851,724021.1074695279
4373503.494877015,724802.9282556607
4371571.9376406865,726627.1767566373
4372874.972284242,726151.9523572231 4374499.933133851))"
#         }'
def insert(d):
    conn = pgUtils.pgConnect()
    cursor=conn.cursor()

    #returning gid stores into the cursor the new gid automatically
    #created by the database as gid is serial
    queryIns='insert into d.buildings (descripcion, geom) values (%s
, st_geometryfromtext(%s,25830)) returning gid'
    values=[d['descripcion'], d['geomWkt']]

    cursor.execute(queryIns, values)
    gid=cursor.fetchall()[0][0] #the new gid is stored in the first
    #field of the first column

    #for the data field we follow the same criteria than psycopg2:
    #first field of first row
    answer={"ok":"true", "message": "Building inserted", "data": [{"
        gid":gid}] }
    conn.commit()
    pgUtils.pgDisconnect(conn)
    return answer
```

In the next listing you will find an example of use:

```
import buildings
jsonString= '{"descripcion": "edificio 3", "geomWkt": "POLYGON
((728018.22892680263612419 4373075.40479189157485962,
727992.44081938592717052 4373037.91898626554757357,
727986.85782705864403397 4373058.38995813205838203,
728018.22892680263612419 4373075.40479189157485962))}'
d=json.loads(jsonString) #from json to dictionary
resp = buildings.insert(d)
print(resp)
```

And the result is:

```
{"ok": "true", "message": "Building inserted", "data": [{"gid":
15}]}
```

We always will return a json with the same fields: *ok*, *message* and *data*. The *ok* field is to indicate whether or not the things went well. The *message* field is to send the message that the user will see in the web page. The *data* field is to send the data requested, usually a list of Python dictionaries, representing each dictionary a row. All these fields are putted inside of a dictionary and Django will be in charge of transformin it into a json string, and to sending it to the client.

2.6.2 Function to update a building from a Python dictionary

In the following listing you have a function to update a building

```
def update(d):
    """
    Update the row which gid is the gid in the jsonString with the
    new field values in the jsonString
    """
    conn = pgUtils.pgConnect()
    cursor=conn.cursor()

    #returning gid stores into the cursor the new gid automatically
    #created by the database as gid is serial
    queryUpdate='update d.buildings set (descripcion, area, geom) =
    row(%s, st_area(geom), st_geometryfromtext(%s,25830)) where
    gid = %s'
    values=[d['descripcion'], d['geomWkt'], d["gid"]]

    cursor.execute(queryUpdate, values)
    n = cursor.rowcount

    #for the data field we are going to return always a list of
    #dictionaries
    answer={"ok":"true", "message": "Building updated", "data": [{"
    rowcount":n}] }
    conn.commit()
    pgUtils.pgDisconnect(conn)
    return answer
```

To execute the function:

```
import buildings
jsonString= '{"gid":"15","descripcion": "edificio 32", "geomWkt":'
POLYGON((728040.22892680263612419 4373075.40479189157485962,
727992.44081938592717052 4373037.91898626554757357,
727986.85782705864403397 4373058.38995813205838203,
728040.22892680263612419 4373075.40479189157485962))}'
d=json.loads(jsonString) #from json to dictionary
resp = buildings.update(d)
print(resp)
```

And the answer is the following:

```
{"ok": "true", "message": "Building updated", "data": [{"rowcount":
1}]}
```

2.6.3 Function to delete a building from gid

In the following listing you have a function to delete a building given a gid

```
def delete(gid):
    conn = pgUtils.pgConnect()
    cursor=conn.cursor()
    queryDelete='delete from d.buildings where gid = %s'
    values=[gid]

    cursor.execute(queryDelete, values)
    n = cursor.rowcount

    #for the data field we are going to return always a list of
    dictionaries
    answer={"ok":"true", "message": "Building deleted", "data": [{"
        rowcount":n}] }
    conn.commit()
    pgUtils.pgDisconnect(conn)
    return answer
```

To execute the function:

```
import buildings
resp=buildings.delete(gid=15)
print(resp)
```

And the answer is the following:

```
{"ok": "true", "message": "Building deleted", "data": [{"rowcount":
1}]}
```

2.6.4 Function to select a building from gid

In the following listing you have a function to select building given a gid

```
def select(gid):
    conn = pgUtils.pgConnect()
    cursor=conn.cursor()
    query_select="SELECT array_to_json(array_agg(registros)) FROM (
        select gid, descripcion, st_asgeojson(geom) from d.buildings
        as t where gid = %s) as registros"
    cursor.execute(query_select, [gid])
    listOfRows=cursor.fetchall()[0][0]

    answer={"ok":"true", "message": "Building retrieved sucessfully
        ", "data": listOfRows }
    pgUtils.pgDisconnect(conn)
    return answer
```

To execute the function:

```
import buildings
resp=buildings.select(gid=7)
print(resp)
```

And the answer is the following:

```

{"ok": "true", "message": "Building retrieved sucessfully", "data":
 [{"gid": 7, "descripcion": "building1", "st_asgeojson": "{\"type
 \": \"Polygon\", \"coordinates
 \": [[[727988, 4373188], [728054, 4373192], [728053.399856473, 4373143.69857413], [72

```

2.7 Exercise 1.

2.7.1 Part 1. Test. Value 2 points

In this exercise you can be asked about the following:

- How to import packages and modules
- How to manipulate Python lists, dictionaries and json strings
- Create functions to insert, delete, update or select in a given table in PostGIS with Python
- Any theoretical concept explained in this document, in class or any subject video.

2.7.2 Part 2. Project. Value 1 point. Create functions to modify your own Postgis database with JSON strings.

The goal of this exercise is to create functions to insert, delete, update and select data from your project tables. You have to think in which PostGIS tables are you going to manage in your own project and create them in a new database. If you are doing the Spatial Information Distribution subject, you must use a database created there, about one of the INSPIRE themes, to connect both subjects.

You can use any code from your teacher or other sources, but you have to understand it.

Create functions to insert, delete, update and select rows in each table of your database. You have to create at least four functions to manage the rows of each table. At least you have to manage three tables, and, at least, two of them have to have a geometry column, of different type of geometry.

All the functions to manage the data of each table have to be in a unique Python file, and put the file in a Python package. To create a package, you only have to create a folder, put all the python files inside, and create an empty file called `__init__.py`. Then you will have a package for manage all your tables, and a python file for each table, with four functions to insert, delete, update and select to operate with each table.

The function to insert or update must receive the row data in a dictionary. You will receive the form data json format and will transform it to a dictionary in a real web application, so this functions will be useful for you in the future.

You have to create one example of use of each of your functions in other module, from which you load the others modules, where your functions are. You will show this examples of use to the teacher.

In the following listing you have an example to manage the buildings table. This table has the fields: *gid*, *descripcion*, *area* and *geom*.

```

def buildingInsert(d):#insert a row with a dictionary
def buildingDelete(gid):#deletes the row than matches with the gid
def buildingUpdate(d)#inside the dictionary you have the gid of the
row to update, and the new row data
def buildingSelect(gid)#returns a list with the row that coincide
with the given gid. If gid is null returns a list with all the
rows. Each row has to be a dictionary.

```

All the functions have to return a dictionary with the following fields:

2.8 Connect Python functions to Internet with Django 2.2

ok : true or false

message : With a text message

data : The list of dictionaries returned. If you do not want to return anything, then is an empty list. If the operation was select, return a list of dictionaries, each dictionary represents a table row. If the operation was an insert. You must return a dictionary with a row. That row has only one field called *gid*, and the value is the *gid* of the row just inserted. In case of an update, or delete, you must return the same, a list with one dictionary, but the field must be 'numOfRowsAffected' and the value must be the number of rows affected by the operation.

You can create a dictionary answer with the following:

```
#creating a dictionary answer
answer={"ok":True, "data":list_of_dictionaries, "message":"
        Building inserted"}
```

Delivery:

- Upload the complete Eclipse project, compressed as *exercise1.zip*, to the subject shared space in Poliformat.
- Upload a file called *groupMembers.txt*, where you specify name and surname of all the members of the group.
- All the members have to upload the same two files: *groupMembers.txt* and *exercise1.zip*.
- You will show the project to the teacher. You have to have prepared a demonstration to execute all the functions, that is function calls with all the parameters ready. Of course all the functions have to work.
- You will have to answer correctly the teacher questions about the code to obtain the whole note of the exercise.

2.8 Connect Python functions to Internet with Django 2.2

At this moment you are able to modify a PostGIS database using your functions, but you need to be able to execute these functions through Internet. This is done using the HTTP protocol and HTTP requests. To manage HTTP requests you need a web server. This web server listen HTTP requests from internet, on the 80 port by default, and is able to execute scripts. The most used HTTP servers at this moment are Apache2 and Nginx. We are going to use Apache2, but not at this stage.

You need a program who be able to receive HTTP requests and execute one function or an other. Django does that and much more. We are going to use Django to execute one function or other, depending of the url.

In a real case you will have a server linked to a domain, with the Apache2 HTTP server installed. The domain will bring the request to your server by typing *http://myDomain/*. In your server you probably have several apps and web pages, each of one has an Apache alias. By typing the Apache alias after the domain Apache will execute the appropriate script. For example *myDjangoAppAlias* will execute a Python Django app. But the Django app also needs to know which function to execute, so it is necessary to write something more, for example *building_insert* could execute the *insert* function in the *buildings* module. See some more examples:

- *http://myDomain/myDjangoAppAlias/building_insert/* ->Will execute the function *buildings.insert*

2.8 Connect Python functions to Internet with Django 2.2

- `http://myDomain/myDjangoAppAlias/building_select/125/` → Will execute the function `buildings.select` selecting the building 125
- `http://myDomain/myDjangoAppAlias/building_delete/` → Will execute the function `buildings.delete`
- `http://myDomain/myDjangoAppAlias/building_update/` → Will execute the function `buildings.update`

You will configure a real server in the subject Content Managers and Smart Cities, but this is the last step on the developing process. To develop you will use a local Apache2, in second place, and the Django server in first place. The local Apache is used to check the app prior to deploy it in the real server, and the Django server is used to develop. The url to execute the Django app with the local Apache are like the following:

- `http://localhost/myDjangoAppAlias/building_insert/` → Will execute the function `buildings.insert`
- `http://localhost/myDjangoAppAlias/building_select/125/` → Will execute the function `buildings.select` selecting the building 125
- `http://localhost/myDjangoAppAlias/building_delete/` → Will execute the function `buildings.delete`
- `http://localhost/myDjangoAppAlias/building_update/` → Will execute the function `buildings.update`

To execute the Django app with the Django server (only for developing purposes) the url are like the following:

- `http://localhost:8000/building_insert/` → Will execute the function `buildings.insert`
- `http://localhost:8000/building_select/125/` → Will execute the function `buildings.select` selecting the building 125
- `http://localhost:8000/building_delete/` → Will execute the function `buildings.delete`
- `http://localhost:8000/building_update/` → Will execute the function `buildings.update`

The Django server serves and listen by the port 8000. You do not use `myDjangoAppAlias` in the url because that is only for Apache.

The usual flow is to use first the Django server, latter the local Apache server and at the end the Apache server in the real server.

2.8.1 Create an Django project and app

Django is installed not in the global Python, but in a Python virtualenv. A virtualenv is an isolated Python installation. The virtualenvs are used to be able to have different Python libraries in the same computer. For example, you can not have Django 2.2 and Django 2.1 at the same Python installation, but you do if they are in different Python virtualenvs, as they are different isolated Python installations.

A virtualenv is a folder with all the Python files. To use a virtualenv Python you must to activate it. Once the virtualenv is activated you can execute the Python interpreter by typing `python`, or install new libraries in it with `pip`.

The virtualenv where all libraries are installed is in `/home/vagrant/apps/env`. To activate the virtualenv open a console and go to `/home/vagrant/apps` folder. To activate the virtualenv execute the following commands:

2.8 Connect Python functions to Internet with Django 2.2

```
vagrant@linux:~/apps$ dir
desweb  env  prueba  pyupvp3  RemoteSystemsTempFiles  vueapp
vagrant@linux:~/apps$ source env/bin/activate /* comando to
activate the virtualenv */
(env) vagrant@linux:~/apps$ /* the (env) prompt indicates that the
virtualenv is activated */
```

Now we can test if Django is installed trying importing it:

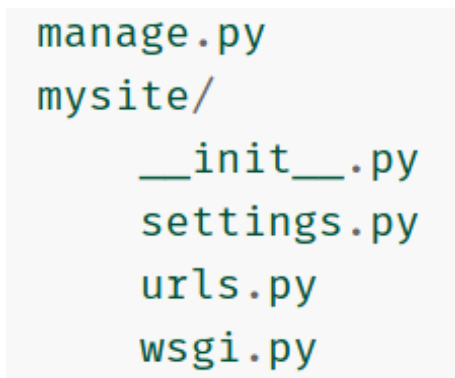
```
(env) vagrant@linux:~/apps$ python /* get into Python interpreter
*/
Python 3.6.8 (default, Oct 7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import django
>>> /* no error, so success */
>>> exit() /* exit from Python interpreter */
(env) vagrant@linux:~/apps$
```

Now, with the virtualenv activated we are ready to start a Django project and an Django app. A Django project can contain several apps. An app is a web page.

To create a Django project in the /home/vagrant/apps folder, with the virtualenv activated, type the following:

```
(env) vagrant@linux:~/apps$ django-admin startproject djdesweb
(env) vagrant@linux:~/apps$
```

The command creates a folder named djdesweb with the files showed in the figure 2.12⁽¹⁾.



```
manage.py
mysite/
  __init__.py
  settings.py
  urls.py
  wsgi.py
```

Figura 2.12: File structure created by the command `django-admin startproject djdesweb`

Now we can create an app:

```
(env) vagrant@linux:~/apps$ cd djdesweb/
(env) vagrant@linux:~/apps/djdesweb$ python manage.py startapp
appdesweb
```

The command creates the folder appdesweb with the files of the figure 2.13.

⁽¹⁾Source <https://docs.djangoproject.com/es/2.2/intro/tutorial01/>

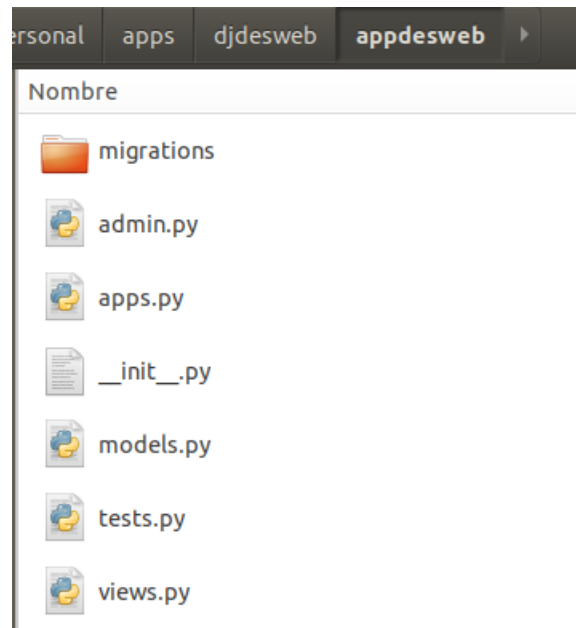


Figura 2.13: File structure created by the command `python manage.py startapp appdesweb`

2.8.2 Configuring the Django project

We have to modify the files created automatically by Django. To do this we are going to create a Eclipse project with the same name than the Django project. As the Django project is in the Eclipse workspace, Eclipse is going to add all the files to the project.

Open Eclipse and start new Pydev project, named *djdesweb*. Select the interpreter in the *env* folder.

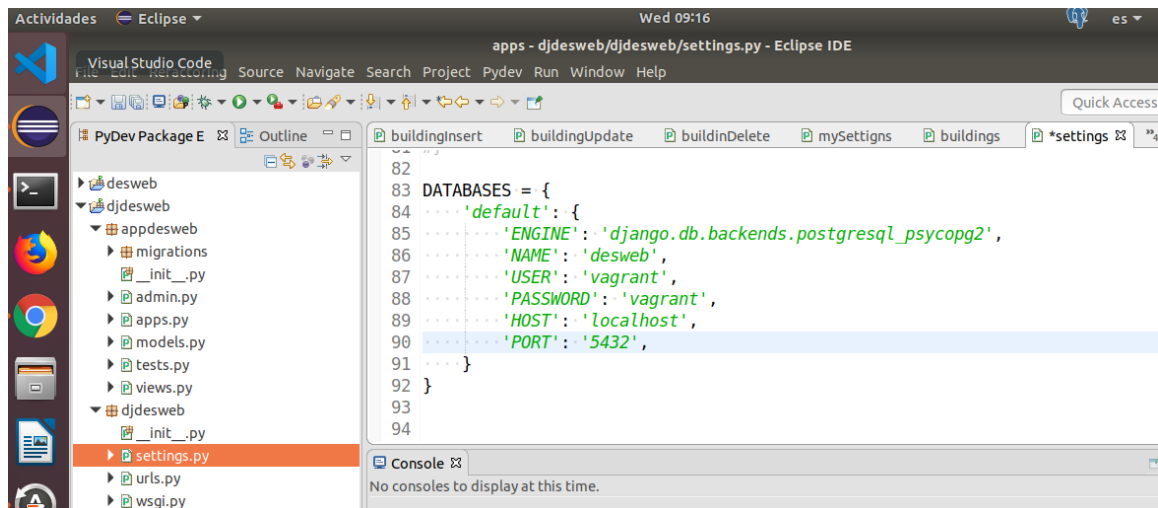


Figura 2.14: Django project edited with Eclipse

Once the project and the app is created you need to configure the project. All the configurations are made in the module `djdesweb/settings.py`. This file configures the entire project. You have to change the following:

Change

```
ALLOWED_HOSTS = []
```

by the following, to allow other ip addresses to use the app:

```
ALLOWED_HOSTS = ['*']
```

Add to the `INSTALLED_APPS` list the app `appdesweb`. You have to add a string in the way: `appname.apps.AppnameConfig`.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'appdesweb.apps.AppdeswebConfig'  
]
```

Comment the following lines

```
#DATABASES = {  
#    'default': {  
#        'ENGINE': 'django.db.backends.sqlite3',  
#        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
#    }  
#}
```

And add the below of the before lines the following, to configure the database credentials.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql_psycopg2',  
        'NAME': 'desweb',  
        'USER': 'vagrant',  
        'PASSWORD': 'vagrant',  
        'HOST': 'localhost',  
        'PORT': '5432',  
    }  
}
```

Now that the postgres database and credentials are set, you can create all the internal Django tables into desweb database, to allow Django to manage users and sessions. Type the following:

```
(env) vagrant@linux:~/apps/djdesweb$ python manage.py migrate  
Operations to perform:  
  Apply all migrations: admin, auth, contenttypes, sessions  
Running migrations:  
  Applying contenttypes.0001_initial... OK  
  Applying auth.0001_initial... OK  
  Applying admin.0001_initial... OK  
  Applying admin.0002_logentry_remove_auto_add... OK  
  Applying admin.0003_logentry_add_action_flag_choices... OK  
  Applying contenttypes.0002_remove_content_type_name... OK  
  Applying auth.0002_alter_permission_name_max_length... OK  
  Applying auth.0003_alter_user_email_max_length... OK  
  Applying auth.0004_alter_user_username_opts... OK  
  Applying auth.0005_alter_user_last_login_null... OK  
  Applying auth.0006_require_contenttypes_0002... OK
```

2.8 Connect Python functions to Internet with Django 2.2

```
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying sessions.0001_initial... OK
```

You can see the tables added to the *desweb* database, in the schema *public*, figure 2.15. Note that thanks to we created a schema called *d*, our tables are not mixed with the Django tables.

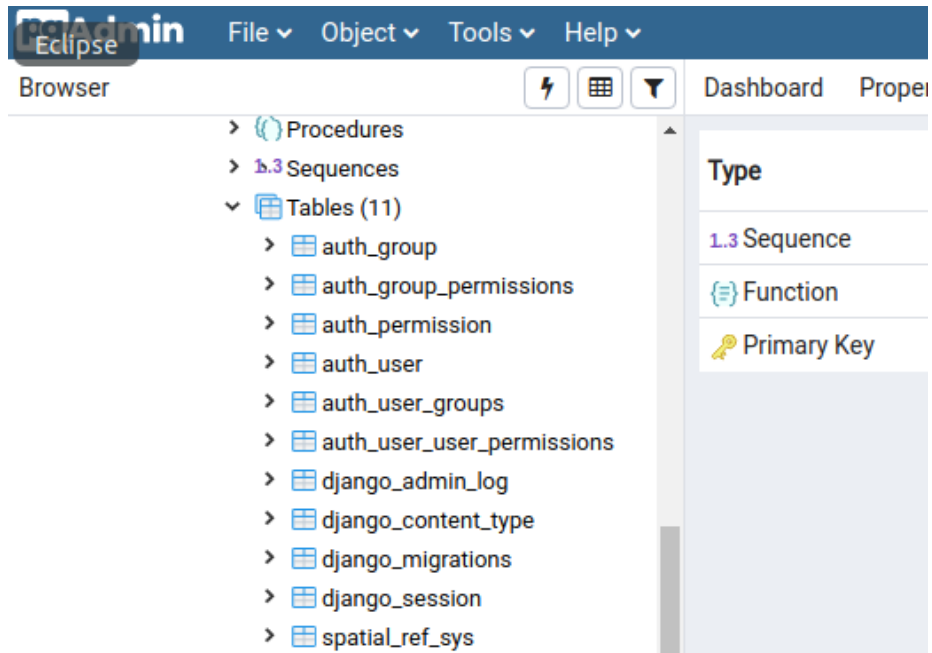


Figura 2.15: File structure created by the command `python manage.py startapp app-desweb`

Now we can create the superuser to manage the Django admin site. **Please note that we are using the default passwords in all the settings. This is potentially dangerous in a real deploy. In that case you must change all the passwords.** This step is not necessary if you are not going to use the Django admin site.

```
(env) vagrant@linux:~/apps/djdesweb$ python manage.py
createsuperuser
Username (leave blank to use 'vagrant'): admin
Email address: admin@admin.com
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

You can run the Django server:

```
(env) vagrant@linux:~/apps/djdesweb$ python manage.py runserver
Watching for file changes with StatReloader
```

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
October 23, 2019 - 08:51:16
Django version 2.2, using settings 'djdesweb.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

You can visit the default page in <http://localhost:8000>. You can visit the Django admin page in <http://localhost:8000/admin> (user admin, psw admin).

Now you are ready to configure the urls and views of your app, in order to be able to execute them.

2.8.3 Initial configuration of the app urls and views

To execute a function from a url, you need tree files:

- `urls.py`. This file links the url with a function in the file `views.py`. You have to load `views.py` from `urls.py` to use its functions.
- `views.py`. A view is a function who has an object called *request* which contains all the data that the user sent to this url. Form `views.py` you will load the modules where you have your functions to access to the database, in order to use them. We are going to put these modules in a new package called *pyCode* inside *appdesweb* package.

On typing a url on the web browser Django search for a coincidence in the file `urls.py` and executes the adequate view in `views.py`, who uses the database access functions, stored unto the modules of the *pyCode* package.

Create the module `appdesweb/urls.py`, and write the following in it:

```
from django.urls import path
from appdesweb import views

urlpatterns = [
    path("hello_world/", views.HelloWord.as_view(), name="hello_world")
]
```

When Django read *hello_world* will execute the method *as_view()* from the class *HelloWorld* of the *views* module. The *HelloWorld* class does not exist yet. Do not worry if you do not understand this syntax it is normal. You only have to repeat the same structure. Open the `appdesweb/views.py` file and write the following:

```
import json

#Django imports
from django.http import JsonResponse, HttpResponse
from django.views import View
from django.contrib.auth import logout
from django.contrib.auth.mixins import PermissionRequiredMixin,
    LoginRequiredMixin
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator

class HelloWord(View):
    def get(self, request):
```

2.8 Connect Python functions to Internet with Django 2.2

```
return JsonResponse({"ok": "true", "message": "Hello world", "
                    data": ""})
```

We have not used all the imports yet, but we will do.

Do not matter if you do not understand what is happening, you only have to know how to repeat this structure in order to manage to execute your functions. I am going to try to explain what is happening. This has to do with Oriented Object Programming.

The class *HelloWorld* inherits of the class *View*, imported above. This is called a *view class*. The class *HelloWorld* is a children of the *View* class. All the children of the class *View* has to define the method *get*, for *get* requests, or the method *post*, for *post* requests, or both. In this case it is only defined the *get* method. All the methods of a Python class receive as first argument the parameter *self*. In the case of the *View* class also receive a second argument called *request*, which contains the information of the user who called the *HelloWorld* view.

You will do the same for each operation in your database:

1. To add a new url in the file `urls.py` to link a url with a view class in the file `views.py`
2. Add a new view class to the file `views.py` to manage the post or get request
3. From the view class, use the functions in the modules of the `pyCode` package, that you already have done.

Now you have to see to Django that load the `appdesweb/urls.py` file. This is something that only have to be done once. Open the file `djdesweb/urls.py` and write the following:

```
from django.contrib import admin
from django.urls import path
from django.urls import include

urlpatterns = [
    path('', include('appdesweb.urls')),
    path('admin/', admin.site.urls),
]
```

Run the Django server, if it have stopped and visit the url `http://localhost:8000/hello_world/`. The Django server recompiles the project on saving a change. If there is a fatal error the server can not continue and it stops. In this case you have to restart it: `python manage.py runserver`. Remember to be in the `/home/vagrant/apps/djdesweb` folder and to have the `virtualenv` activated.

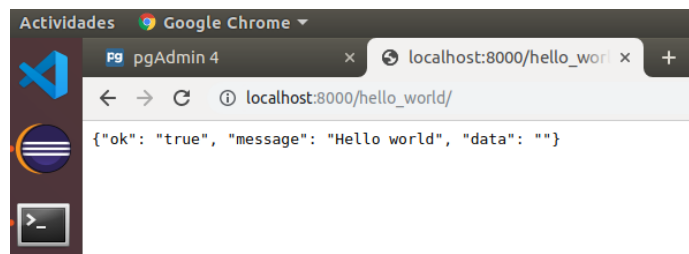


Figura 2.16: Django HelloWorld view answer

2.8.4 Creating the app urls and views to access to the database

For request that only gets information from the server, without modify the server data, it is used *get* requests. *Post* requests are used to modify the database. So for select operations you can use *get*. Files and form data must always be sent by *post* method.

In this section you are going to configure the following urls:

- `http://localhost:8000/building_insert/` →Will execute the function `buildings.insert`, sending a json to the server and receiving a json from the server. Its a post request.
- `http://localhost:8000/building_select/125/` →Will execute the function `buildings.select` selecting the building 125. We will send the building gid in the same url. Its a get request.
- `http://localhost:8000/building_delete/` →Will execute the function `buildings.delete`. Its a post request.
- `http://localhost:8000/building_update/` →Will execute the function `buildings.update`. Its a post request.

The server exposed operations, *building_insert*, *building_select*, *building_delete*, *building_update* form the server HTTP Application Server Interface (API). An API is a set of operations that someone did, and the user only have to know what each operation needs to work and what is going to be the result.

The first step is to create a new package in *appdesweb*, called *pyCode*, and copy inside the `buildings.py` module. As `buildings.py` imports `pgUtils.py` and this imports `mySettings.py`, we have to copy these modules too. In the figure 2.17 you can see the project files.

Remember, you always must import the modules starting the path from the project folder. For example, if you have a module in `appdesweb/pyCode` and you want to import the module `appdesweb/pyCode/mySettings`, instead of

```
import mySettings
```

You must use

```
from appdesweb.pyCode import mySettings
```

2.8.4.1. Select a building by gid

1. Import the module `buildings` from `appdesweb/views.py`, and add the `BuildingSelect` class to the `appdesweb/views.py`:

```
from appdesweb.pyCode import buildings /* The imports must  
appear in the beginning of the module */  
  
class BuildingSelect(View):  
    def get(self, request, gid):  
        r= buildings.select(gid)  
        return JsonResponse(r)
```

2. Add a url to the `appdesweb/urls.py`

```
from django.urls import path  
from appdesweb import views  
  
urlpatterns = [
```

2.8 Connect Python functions to Internet with Django 2.2

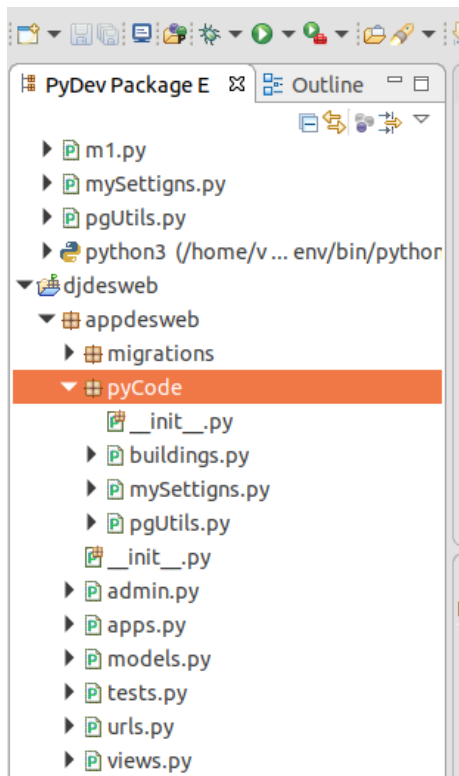


Figura 2.17: Current project files

```
path("hello_world/", views.HelloWord.as_view(), name="
    hello_world"),
path('building_select/<gid>/', views.BuildingSelect.as_view
    (), name='building_select')
]
```

The above listing defines a variable, called `gid`, that must be passed in the url. E.g: `http://localhost:8000/building_select/7/` will search for the building which `gid = 25`.

3. Restart the Django server, if it is stopped, and check the new url `http://localhost:8000/building_select/7/`, figure 2.18

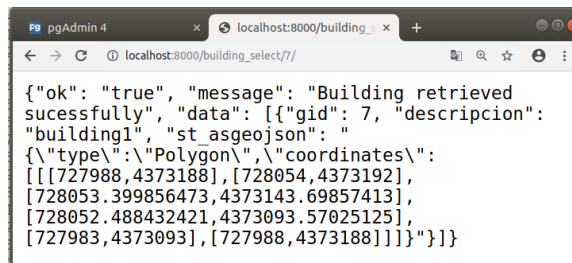


Figura 2.18: Building select result

2.8.4.2. Insert a building

Insert implies to modify the database so that you need to use the post request. You have to imagine a web form than the user fulfils and then press the *Send* button, then a JavaScript routine generates a json with the form values and sends, by post, a json with all the form data. In the server, you have to get the form data, transform it into a Python dictionary, and send it to your buildings.insert function. The JavaScript code will send something like that to the server:

```
'{"descripcion": "edificio 5", "geomWkt": "POLYGON
((728155.94273754325695336 4373095.60990698169916868,
728217.62150992138776928 4373098.00261797849088907,
728200.6066761618712917 4373038.98241337575018406,
728154.34759687830228359 4373040.84341081790626049,
728155.94273754325695336 4373095.60990698169916868))}'
```

Perform the following steps:

1. Create the BuildingInsert class in the module appdesweb/views.py

```
@method_decorator(csrf_exempt, name='dispatch')
class BuildingInsert(View):
    def post(self, request):
        d=general.getPostFormData(request)
        r= buildings.insert(d)
        return JsonResponse(r)
```

Here you have some new things. The *method_decorator* is necessary always in all the views than are going to manage a post request. The information sent by the user is stored in the *request* object. The *request* object contains the *POST* object, which is similar to a Python dictionary.

In the previous listing the function *getPostFormData* from the module *general* is used to get the POST data. The function returns a Python dictionary with the form data. Create a new module called *general* in *deswebapp/pyCode* and paste the following function definition:

```
import json
def getPostFormData(request):
    """
    Depending of who sends the post data is in request.POST or
    request.body.
    If it is in request body it is also y binary and you have to
    decode
    The function searches for the key formData, and if exists
    returns a dictionary
    If formData does not exist return also a ditionary. In this
    way you can use the same
    function if you are sending the data with Postman, Ajax or
    Angular
    """
    js=request.POST.get("formData","")
    if js != "":
        d=json.loads(js)
    else:
        js=request.body.decode('utf-8')
        d=json.loads(js)
        d2=d.get("formData","")
        if d2 != "":
```

```
        d=d2
    return d
```

To use the previous function you have to import it:

```
from appdesweb.pyCode import general
```

It is not necessary to modify the function *buildings.insert* but as is Django who is going to execute it we can avoid to use our psycopg2 connection and use the Django connection, which is exactly equal. To do this has some advantages. Django opens and close the connections according to its needs. Its automatic, so we do not have to worry about to open or close them. The new code for the function *buildings.insert* is:

```
from django.db import connection as conn

def insert(d):
    cursor=conn.cursor()
    #returning gid stores into the cursor the new gid
    #automatically created by the database as gid is serial
    queryIns='insert into d.buildings (descripcion, geom) values
              (%s, st_geometryfromtext(%s,25830)) returning gid'
    values=[d['descripcion'], d['geomWkt']]

    cursor.execute(queryIns, values)
    gid=cursor.fetchall()[0][0] #the new gid is stored in the
    #first field of the first column

    #for the data field we are going to return always a list of
    #dictionaries
    answer={"ok":"true", "message": "Building inserted", "data":
           [{"gid":gid}] }
    conn.commit()
    return answer
```

As you can see in the previous listing, we import the connection from Django and do not close it at the end of the function. The rest is all equal. You must use the Django connection in your code.

2. Add a url to the appdesweb/urls.py

```
urlpatterns = [
    path("hello_world/", views.HelloWord.as_view(), name="
        hello_world"),
    path('building_select/<gid>/', views.BuildingSelect.as_view
        (), name='building_select'),
    path('building_insert/', views.BuildingInsert.as_view(),
        name='building_insert')
]
```

3. Restart the Django server, if it is stopped, and check the new url http://localhost:8000/building_insert/, figure 2.18. It will not work, figure 2.19.

In the future you will perform post request with JavaScript, but now we need a software to do it, Postman. Run postman by double clicking in its icon. You will find it in the /home/vagrant/desktopApps/Postman folder

Once Postman is opened, select New→Collection figure 2.20, which is a request collection.

2.8 Connect Python functions to Internet with Django 2.2



Figura 2.19: You can not send a post request with the web browser

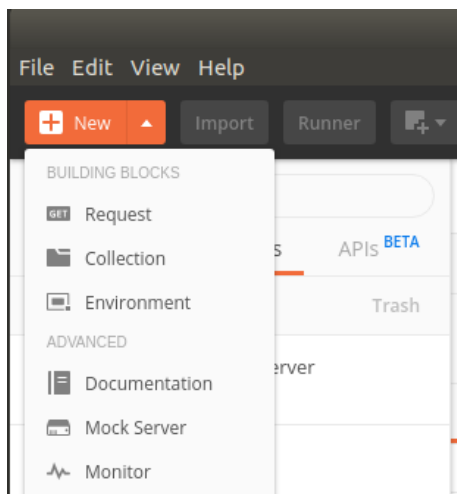


Figura 2.20: Create a collection with Postman

Set the name *appdesweb Django server*, figure 2.21 to indicate that is a set of requests to be sent do the *appdesweb* application and to the Django server.

Select *New*→*Request*. Type the name *building_insert* and be sure that it is going to be added to the *appdesweb Django server* collection,figure 2.22.

Change the default *GET* request to *POST*, figure 2.23.

Select the *body* tab and in the *KEY* column create a variable called *formData*, and in the same row, and in the *VALUE* column type the following, figure 2.24:

```
{"descripcion": "edificio 5", "geomWkt": "POLYGON
((728155.94273754325695336 4373095.60990698169916868,
728217.62150992138776928 4373098.00261797849088907,
728200.6066761618712917 4373038.98241337575018406,
728154.34759687830228359 4373040.84341081790626049,
728155.94273754325695336 4373095.60990698169916868))" }
```

Pay attention that it is a Python dictionary, not a json, but it is sent as json.

Press *Save* and *Send*. Next time you open Postman the collection and the request will be available to try again. The answer must be the showed in the figure 2.25.

CREATE A NEW COLLECTION

Name

appdesweb Django server

Description Authorization Pre-request Scripts Tests Variables

This description will show in your collection's documentation, along with the descriptions of its folders and requests.

Make things easier for your teammates with a complete request description.

Figura 2.21: Set the collection name with Postman

SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).
[Learn more about creating collections](#)

Request name

building_insert

Request description (Optional)

Make things easier for your teammates with a complete request description.

Descriptions support [Markdown](#)

Select a collection or folder to save to:

Search for a collection or folder

appdesweb Django server + Create Folder

POST building_insert

Cancel Save to appdesweb Django server

Figura 2.22: Add a request to a collection with Postman

2.8.4.3. Update a building

You have a function that receives a json with a row, extracts the gid and updates the database row which match with that gid with the rest of data in the json row. So the data to send to the server is very similar than the for the url *building_insert*. As you are going to change the database, you must use the post method.

Perform the following steps:

1. Create the BuildingUpdate class in the module appdesweb/views.py

```
@method_decorator(csrf_exempt, name='dispatch')
class BuildingUpdate(View):
    def post(self, request):
        d=general.getPostFormData(request)
        r= buildings.update(d)
        return JsonResponse(r)
```

2.8 Connect Python functions to Internet with Django 2.2

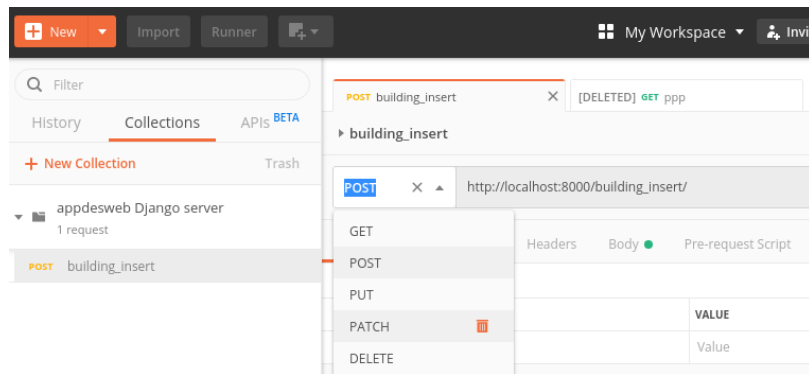


Figura 2.23: Set the request type with Postman

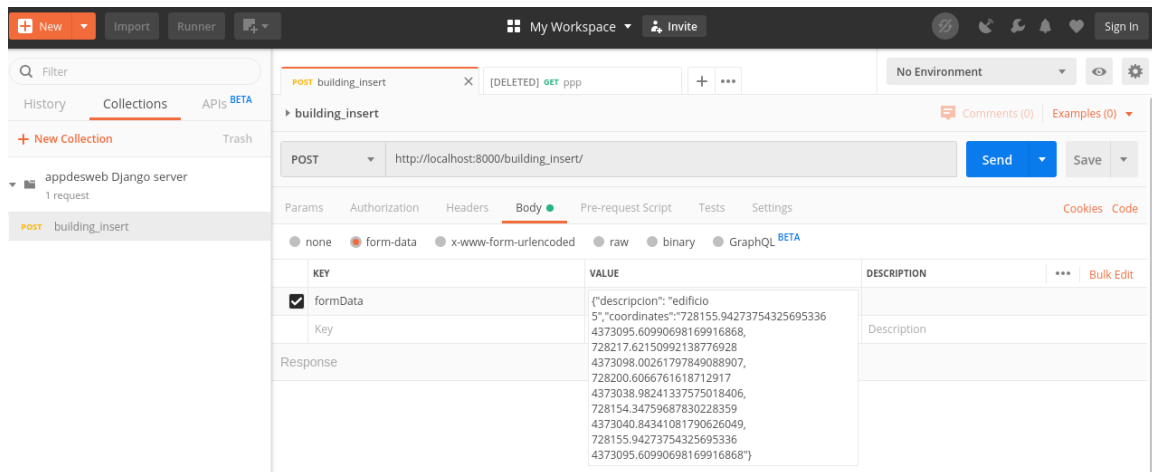


Figura 2.24: Set the formData variable with Postman

Nothing new to explain.

2. Add a url to the appdesweb/urls.py

```
urlpatterns = [  
    path("hello_world/", views.HelloWord.as_view(), name="hello_world"),  
    path('building_select/<gid>', views.BuildingSelect.as_view(), name='building_select'),  
    path('building_insert/', views.BuildingInsert.as_view(), name='building_insert'),  
    path('building_update/', views.BuildingUpdate.as_view(), name='building_update')  
]
```

3. Restart the Django server, if it is stopped.

As it *building_update* expects a post request you have to use Postman. Postman allow you to copy collections or requests. Right button on the *building_insert* request and select *Duplicate*. Set the name to *building_update*, change the url to *building_update*, change the building description, and add the gid field specifying the gid of the building to update.

2.8 Connect Python functions to Internet with Django 2.2

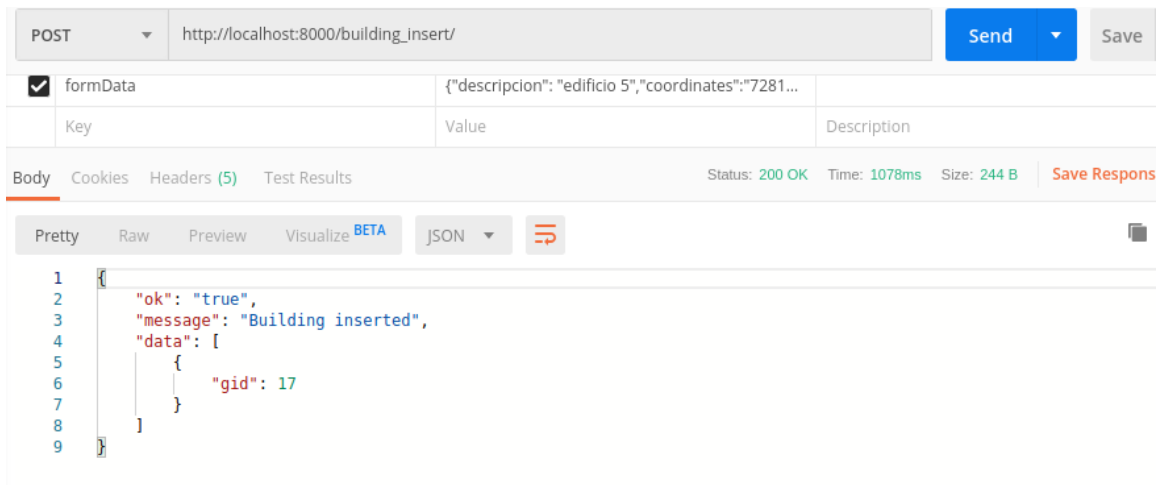


Figura 2.25: Set the formData variable with Postman

```
{"gid": "13", "descripcion": "edificio 5", "geomWkt": "POLYGON
((728155.94273754325695336 4373095.60990698169916868,
728217.62150992138776928 4373098.00261797849088907,
728200.6066761618712917 4373038.98241337575018406,
728154.34759687830228359 4373040.84341081790626049,
728155.94273754325695336 4373095.60990698169916868))" }
```

Press *Save* and *Submit*. The result is showed in the next listing.

```
{
  "ok": "true",
  "message": "Building updated",
  "data": [
    {
      "rowcount": 1
    }
  ]
}
```

2.8.4.4. Delete a building

You have a function that receives `gid` and deletes the building who `gid` matches with it. As it is a operation that change the database, it must be sent by post http request. To follow the same criteria you are going to send from Postman a dictionary like the following:

```
{"gid": "13"}
```

but the server will receive a json like:

```
'{"gid": "13"}'
```

Perform the following steps:

1. Create the `BuildingDelete` class in the module `appdesweb/views.py`

```
@method_decorator(csrf_exempt, name='dispatch')
class BuildingDelete(View):
    def post(self, request):
        d=general.getPostFormData(request)
```

```
r= buildings.delete(gid=d["gid"])
return JsonResponse(r)
```

Nothing new to explain.

2. Add a url to the appdesweb/urls.py

```
urlpatterns = [
    path("hello_world/", views.HelloWord.as_view(), name="
        hello_world"),
    path('building_select/<gid>/', views.BuildingSelect.as_view
        (), name='building_select'),
    path('building_insert/', views.BuildingInsert.as_view(),
        name='building_insert'),
    path('building_update/', views.BuildingUpdate.as_view(),
        name='building_update'),
    path('building_delete/', views.BuildingDelete.as_view(),
        name='building_delete')
]
```

3. Restart the Django server, if it is stopped.

Add a new request to the *appdesweb* Postman collection called *building_delete*. You have to send the following data in the variable *formData*:

```
{"gid": "13"}
```

Press *Save* and *Submit*. The result is showed in the next listing.

```
{
  "ok": "true",
  "message": "Building deleted",
  "data": [
    {
      "rowcount": 1
    }
  ]
}
```

2.9 Sending data to the views: POST or GET

2.9.1 Postman

Postman program allow you to store the requests in a *collection* to use them again and again, until you be sure the back-end works properly. You can check you API without to have the front-end finished.

In the virtual machine you can execute Postman by double clicking into the file `/home/vagrant/desktopApps/Postman/Postman`.

2.11 Users management and authentication

In a real project you only will allow authenticated users to perform some API operations. In our Buildings project, for example, the usual way would be to allow all users to select buildings, but only authenticated users to insert, delete or update buildings, that is to modify the database.

Django provides very useful tools to manage and authenticate users. Usually you will have three types of user: administrators, editors and users:

- administrators: they can get into the administration site and manage other users.
- editors: they can modify data in the database.
- users: they can see the data.

2.11.1 Create users and groups with the Django admin site

You can create and manage users and groups with Python and with the Django administration site. This is a pre-installed application in all the Django projects. You can access to the admin site by typing `http://your-server/admin`. In the case of the development server `http://localhost:8000/admin` figure 2.27. You can authenticate with the user *admin*, password *admin*. You created this user in a previous step with the command `python manage.py createsuperuser`.

To use Django users framework, you should have:

- Migrated the database first, executing `python manage.py migrate`, only once. See the section 2.8.2.
- In the `INSTALLED_APPS` variable, in the module `settings.py`, the following apps `django.contrib.auth`, and `django.contrib.admin`. These apps are installed by default.
- In the `urls.py` of the project folders have imported `admin` from `django.contrib`, and to add `path('admin/', admin.site.urls)` to the `urlpatterns` list. This is also done by default.

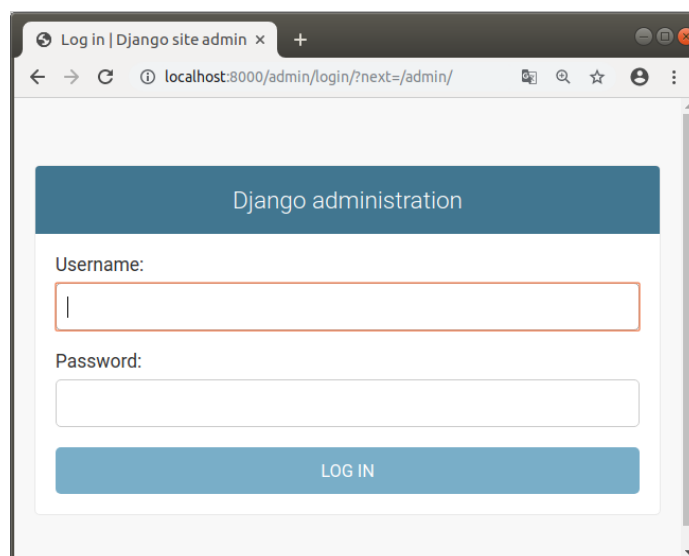


Figura 2.27: Django admin site. Login

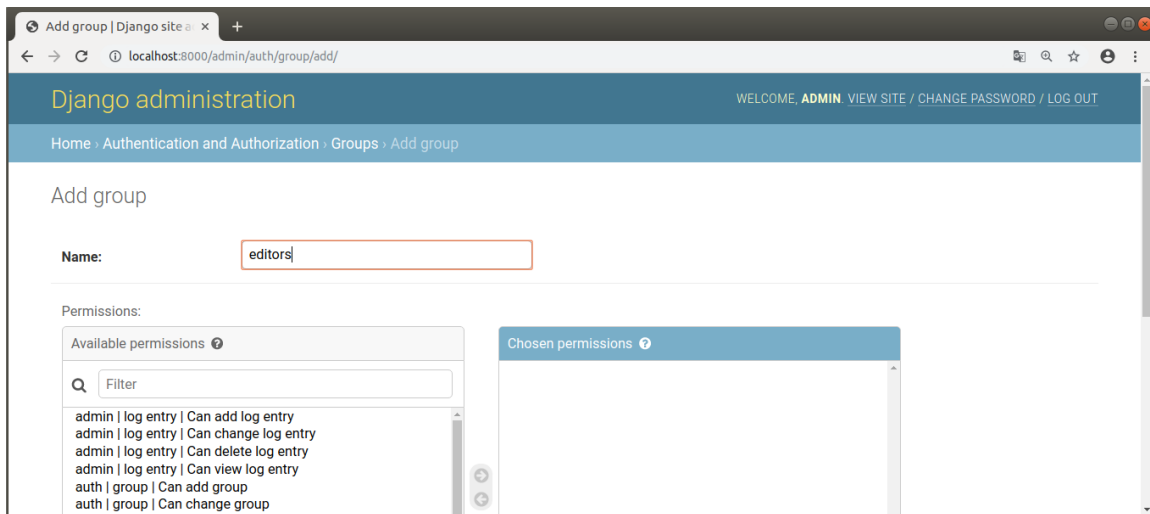


Figura 2.28: Django admin site. Add group

Select the *Groups* link and create the groups needed, by clicking in the button *Add ... +*, figure 2.28.

Return to the main page and select the link *Users*. In the new window add a user by clicking in the button *Add ... +*. Add all the necessary user details, figure 2.29. In the example, the username is *desweb@desweb.com* and the password is *adminadmin*.

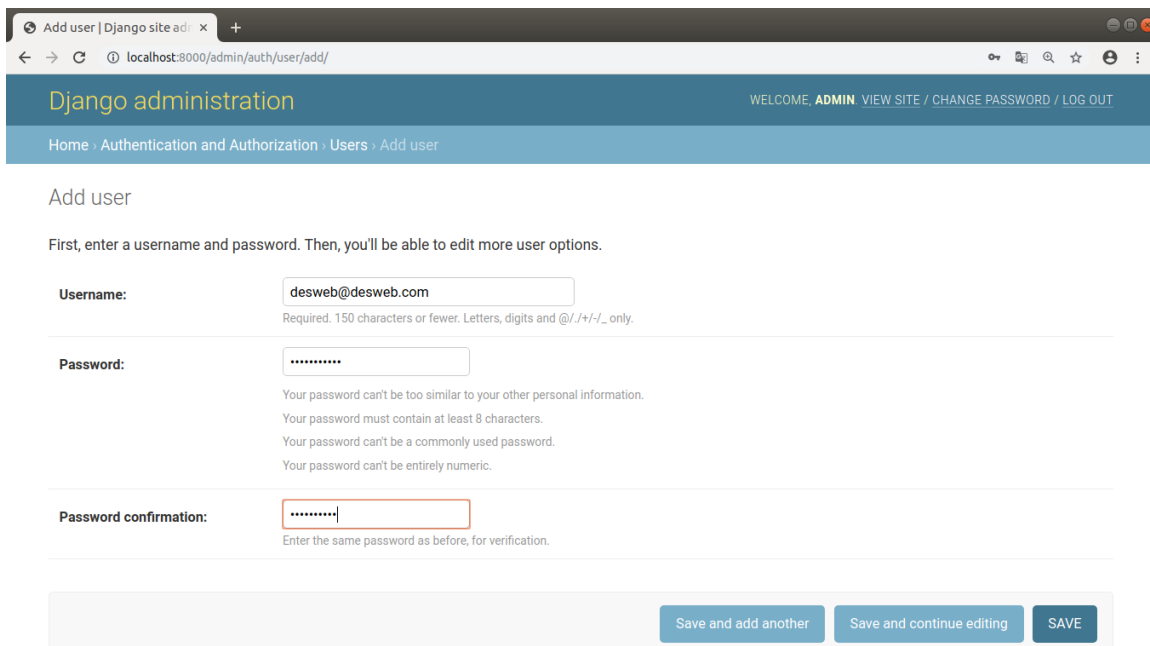


Figura 2.29: Django admin site. Add user

To add a user to a group, go back to the main page and select the user, select the group on the left list and pass it to the right list, figure 2.30.

A user can belong to several groups.

Permissions

Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status
Designates whether the user can log into this admin site.

Superuser status
Designates that this user has all permissions without explicitly assigning them.

Groups:

Available groups ?

administrators

users

Chosen groups ?

editors

Figura 2.30: Django admin site. Add user to a group

2.11.2 Create super-user from command

You can create a super-user by typing:

```
python manage.py createsuperuser
```

2.11.3 Create normal users from command

You can not add a normal user by executing any shell command. To create normal users, you have to use the Django Admin Site, see the section 2.11.1, or with Python, see the section 2.11.11.

2.11.4 Change users password from command

You can change user password with the following command:

```
python manage.py changepassword <user_name>
```

2.11.5 Protect some views from unauthenticated users

Now you have some users you can check if the user is authenticated or not, before to use any view.

To forbid not authenticated users to access to a view you have to import the class *LoginRequiredMixin* from *django.contrib.auth.mixins*, and make your views to inherit from it. Lets change the code of the module *buildings.py* to forbid non authenticated users to the views *BuildingInsert*, *BuildingDelete* and *BuildingUpdate*. See the next listing:

```
#appdesweb.views.py

#standard imports
import json
```

```
#Django imports
from django.http import JsonResponse, HttpResponse
from django.views import View
from django.contrib.auth import logout
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator

from appdesweb.pyCode import buildings

class HelloWord(View):
    def get(self, request):
        return JsonResponse({"ok":"true","message": "Hello world", "
            data":""})

class BuildingSelect(View):
    def get(self, request, gid):
        r= buildings.select(gid)
        return JsonResponse(r)

@method_decorator(csrf_exempt, name='dispatch')
class BuildingInsert(LoginRequiredMixin, View):
    def post(self, request):
        formData=request.POST.get("formData", "")
        if formData=="":
            return JsonResponse({"ok":"false", "message": "Error it
                is necessary to set the formData variable", "data":
                [] })
        r= buildings.insert(jsonString=formData)
        return JsonResponse(r)

@method_decorator(csrf_exempt, name='dispatch')
class BuildingUpdate(LoginRequiredMixin, View):
    def post(self, request):
        formData=request.POST.get("formData", "")
        if formData=="":
            return JsonResponse({"ok":"false", "message": "Error it
                is necessary to set the formData variable", "data":
                [] })
        r= buildings.update(jsonString=formData)
        return JsonResponse(r)

@method_decorator(csrf_exempt, name='dispatch')
class BuildingDelete(LoginRequiredMixin, View):
    def post(self, request):
        formData=request.POST.get("formData", "")
        if formData=="":
            return JsonResponse({"ok":"false", "message": "Error it
                is necessary to set the formData variable", "data":
                [] })
        d=json.loads(formData)
        r= buildings.delete(gid=d["gid"])
        return JsonResponse(r)
```

In the previous listing the class based views *BuildingInsert*, *BuildingDelete* and *BuildingUpdate* inherit from the *LoginRequiredMixin* class first and the *View* class latter. The order is important. Only with this any unauthenticated user are able to use these views. Lets try to insert a building with PostMan. You will get the following answer:

```
Page not found (404)
Request Method: GET
Request URL: http://localhost:8000/accounts/login/?next=/
             building_insert/
Using the URLconf defined in djdesweb.urls, Django tried these URL
patterns, in this order:

hello_world/ [name='hello_world']
building_select/<gid>/ [name='building_select']
building_insert/ [name='building_insert']
building_update/ [name='building_update']
building_delete/ [name='building_delete']
admin/
The current path, accounts/login/, didn't match any of these.
```

Django forbids the access but do not know what to answer if the user is not authenticated. To solve this you have to create a function view that will be called automatically each time a non authenticated user tries to use a protected view. The view will return a json. This is made in three steps:

1. Add the following function view to the file *appdesweb.viewsUser.py* module. You have to create it. This module will manage all the views that have something to do with users management:

```
#appdesweb.viewsUsers.py
from django.http import JsonResponse, HttpResponse

def notLoggedIn(request):
    return JsonResponse({"ok":"false","message": "You are not
                       logged in", "data":""})
```

2. Add a url to the file *appdesweb.urls.py* to execute the *notLoggedIn* view function:

```
#appdesweb.urls.py

from django.urls import path
from . import views, viewsUser

urlpatterns = [
    path('not_logged_in/', viewsUsers.notLoggedIn, name='
         not_logged_in'),
    path("hello_world/", views.HelloWord.as_view(),name="
         hello_world"),
    path('building_select/<gid>/', views.BuildingSelect.as_view
         (), name='building_select'),
    path('building_insert/', views.BuildingInsert.as_view(),
         name='building_insert'),
    path('building_update/', views.BuildingUpdate.as_view(),
         name='building_update'),
    path('building_delete/', views.BuildingDelete.as_view(),
         name='building_delete')
]
```

3. Set the variable `LOGIN_URL` to `'/not_logged_in/'` in the `desweb.settings.py` file. This is the url to change when a user tries to do something that requires to be logged in, and he is not. That variable is not set, you have to write it. You can add the following at the end of the file `desweb.settings.py`:

```
LOGIN_URL = '/not_logged_in/' #url to the view to use when a
    user tries to do something that requires to be logged in,
    and he is not logged in
```

With these configuration if you try to use any of the protected views you will get the following message:

```
{
    "ok": "false",
    "message": "You are not logged in",
    "data": ""
}
```

Try it with Postman.

2.11.6 Authenticate users

To authenticate users you have to have a function view. This view will receive a the a json with the content:

```
'{"formData": {"username": "the_username", password: "the_password
"}}
```

This is exactly the same format that the views to insert and update buildings receive the data. To authenticate a user you must create a new module dedicated to manage users: `appdesweb.pyCode.users.py`, and paste the following code (see the comments to understand the code. More detailed information about authentication in the official Django site).

First you have to create the following two functions in the module `appdesweb.pyCode.libs.general.py`

```
#appdesweb.pyCode.libs.general.py

import json

from django.contrib.auth.models import User

def getPostFormData(request):
    #...
    #already existing function

def getUserGroups(user: User):
    """
    Gets a lists with the user groups that the user belongs. The
    user is an object of the
    django.contrib.auth.models.User class
    """
    l = user.groups.values_list('name', flat = True) # QuerySet
    Object
    return list(l)

def getUserGroups_fromUsername(username):
```

```
"""
Gets a lists with the user groups that the user belongs. The
username is the username,
usually an email
"""
user=User.objects.get(username=username)
return getUserGroups(user)
```

Now you can create the following function in the module *appdesweb.pyCode.users.py*:

```
#appdesweb.pyCode.users.py

import random, time

from django.contrib.auth.models import User
from django.contrib.auth import authenticate, login
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator

from appdesweb.pyCode.libs import pgUtils, general

def appLogin(request):
    #django puts in every request the object 'user',
    #which is of the class from django.contrib.auth.models.User
    #this object is used to get the user data
    if request.user.is_authenticated:
        return {"ok":"true","message": "You where already
            authenticated", "data":[{"username": request.user.
            username}]}

    #to make thinks difficult to hackers, you make a random delay,
    between 0 and 1 second
    seconds=random.uniform(0, 1)
    time.sleep(seconds)

    #get the form data
    d=general.getPostFormData(request)
    username=d.get("username","")
    password=d.get("password","")

    #If user is not None, the credentials where correct
    user = authenticate(username=username, password=password)
    if user:
        login(request,user)#introduce into the request the user data
        #in order, in the the following requests, know the user
        is authenticated
        return {"ok":"true","message": "User {0} logged in".format(
            username), "data":[{"userame": username, 'userGroups':
            general.getUserGroups(user=request.user)}]}
    else:
        return {"ok":"false","message": "Wrong user or password", "
            data":[]}
```

Now you need a view to be able to execute the above function. Create a new module called *appdesweb.viewsUsers.py*, and paste the following code:

```
#appdesweb.viewsUsers.py
```

```

from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator
from django.views import View
from django.contrib.auth import logout

from appdesweb.pyCode import users

def notLoggedIn(request):
    return JsonResponse({"ok": "false", "message": "You are not logged
        in", "data": ""})

@method_decorator(csrf_exempt, name='dispatch')
class AppLogin(View):
    def post(self, request):
        r= users.appLogin(request)
        return JsonResponse(r)

```

Now you need an url in the file `appdesweb.urls.py` to be able to execute the above view:

```

#appdesweb.urls
from django.urls import path
from appdesweb import views, viewsUsers

urlpatterns = [
    path('not_logged_in/', viewsUsers.notLoggedIn, name='
        not_logged_in'),
    path('app_login/', viewsUsers.AppLogin.as_view(), name='
        app_login'),
    path("hello_world/", views.HelloWord.as_view(),name="hello_world
        "),
    path('building_select/<gid>/', views.BuildingSelect.as_view(),
        name='building_select'),
    path('building_insert/', views.BuildingInsert.as_view(), name='
        building_insert'),
]

```

Use the PostMan application to execute the `app_login` view figure 2.31.

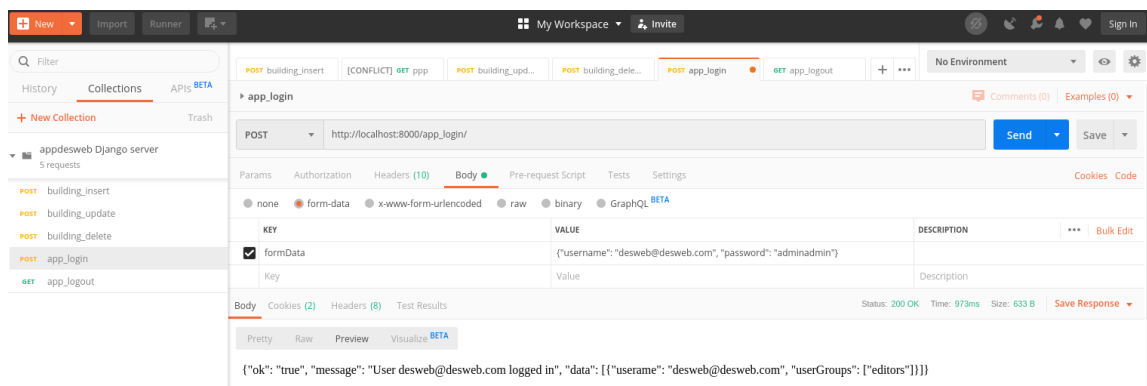


Figure 2.31: Login with PostMan

2.11 Users management and authentication

If you try to login again you will receive the message:

```
{"ok": "true", "message": "You where already authenticated", "data": [{"username": "desweb@desweb.com"}]}
```

This means the system remembers that you where already logged in. But, how?. Internet is an stateless system. This means that once the server sends the answer forgets everything about the user. If we where not using Django, next request of the authenticated user will be rejected, as the server forgets that the user where already authenticated. What Django does to remember the user?. When you call the function `login(request, user)`, the `login` function creates a random session ID and it is stored in the database, together the encrypted session data (field `session_data`). But this is not enough, the `login` function also stores the session ID in the request, in a part called `headers`, in something called cookie (figure 2.32). Wen the server answer is received by the client, usually a web browser, but in this case PostMan, the cookie is stored in the client. The cookies are automatically sent to the server in every request, so Django gets the session ID, search for it in the database and, if it exists, retrieves from the database the information. These information is encrypted, but Django is able to decrypt it. In this way, only with the session ID Django can retrieve the username, if he where logged in, etc.

The image shows two screenshots. The top one is from Postman, displaying the headers of a POST request to `http://localhost:8000/app_login/`. The headers include `Postman-Token`, `Host`, `Content-Type`, `Accept-Encoding`, `Cookie`, and `Content-Length`. The `Cookie` header contains `csrftoken=TTUddjUxixwHo6nKM76XIV5dEMS8VOha2RNA0BJMkn29r1L2elYrSdWbUmQBxTDm; sessionid=wzx39399mkr0xw8ghz9bj7lezg6v6ck`. The bottom screenshot is from the Django Admin interface, showing a query in the Query Editor: `SELECT * FROM public.django_session`. The Data Output table shows two rows of session data:

session_key	session_data	expire_date
pk4juk4bizme87utv60a6vufjo7juloi	ODU2MWNhZT14YTU0ZDY0NmI0NWl4MzZhMzQyM2FhZ...	2021-04-28 21:01:14.850925+00
wzx39399mkr0xw8ghz9bj7lezg6v6ck	ZGE20DA5ZTJKMWRjYQwNTY1MTBkODUwZjEwYjU3NjB...	2021-04-28 21:43:28.918418+00

Figure 2.32: Session ID stored in a cookie, in the headers of the request, and in the database, in the `django_session` table

Try to insert a building, in order to check that now, as you are logged in, you are able to insert.

2.11.7 Session expiration time

Although the user close the client (the web browser, or PostMan), the session ID is stored, and each time the user visits our page, the cookie is sent to the server, so Django is able to retrieve the user data: if he is authenticated or not, the username, etc. You can test this by closing PostMan, restarting it again, and sending an insert request. You will be able to do it because the cookie is sent in the header of the request, and the session expire date is not expired, see the field *expire_date* in the figure 2.32.

By default the session expire date is set 14 days latter of the session initialization. You can change this. For example you can force session expiration on closing the client (the web browser, with PostMan it does not work). To do that you have to set the variable *SESSION_EXPIRE_AT_BROWSER_CLOSE* to *True*, in the *settings.py* module.

2.11.8 Logout a user

To logout a user you need an other view linked to an url. You have to use the `django.contrib.auth.logout` function. Lets create the view:

```
#appdesweb.viewsUsers
...

from django.views import View
from django.contrib.auth import logout
...

class AppLogout(View):
    def get(self, request):
        if request.user.is_authenticated:
            username=request.user.username
            logout(request) #removes from the header of the request
                            the user data, stored in a cookie
            return JsonResponse({"ok":"true","message": "The user
                                {0} is now logged out".format(username), "data":[]})
        else:
            return JsonResponse({"ok":"false","message": "You where
                                not logged in", "data":[]})
```

Now you need to create an url in *appdesweb.urls*:

```
#appdesweb.urls

from django.urls import path
from appdesweb import views, viewsUsers

urlpatterns = [
    path('not_logged_in/', viewsUsers.notLoggedIn, name='
        not_logged_in'),
    path('app_login/', viewsUsers.AppLogin.as_view(), name='
        app_login'),
    path('app_logout/', viewsUsers.AppLogout.as_view(), name='
        app_logout'),
    path("hello_world/", views.HelloWord.as_view(),name="hello_world
        "),
    path('building_select/<gid>/', views.BuildingSelect.as_view(),
        name='building_select'),
```

```
    path('building_insert/', views.BuildingInsert.as_view(), name='
        building_insert'),
]
```

Test the logout operation with Postman and check if after two consecutive logouts you receive different messages. Check, if after to be logout, you can insert a building, you must not be able to do it.

2.11.9 Limit the access to views to users that belongs to some groups

In most cases it is not enough that a user be authenticated to allow him to use a view. For example *users*, and *editors* should not be able to perform some operations that only administrators must perform, despite to be authenticated. So it is not enough to be authenticated to be able to use some views, also it is necessary, in most cases, to check the group that the user belongs.

This is easy to check as you have the function `appdesweb.pyCode.libs.general.getUserGroups_fromUsername`, that receives the username which is always stored in the request, in `request.user.username`. This function returns a list with the name of the groups that the user belongs. To check if a group name is in that list is very easy. For example, in the following listing is checked if the logged in user is administrator:

```
l=general.getUserGroups('user@user.com')
if not 'administrator' in l:
    #is not able to continue
    return {"ok": "false", "message": "You have to be administrator
        to use this view", "data": []}

#the user is administrator
#the logic of the view continues
...
```

2.11.10 Users management with Python. Official documentation

See the official documentation at <https://docs.djangoproject.com/en/3.2/topics/auth/default/>.

2.11.11 Create users with Python

You probably will need to create new users from a view, without using the Django administration site. You can create a new user by using the Django *User* model

```
from django.contrib.auth.models import User

...
#checks whether or not the user exists
if User.objects.filter(username=d['username']).exists():
    return {"ok":"false","message": "The user {0} already exists
        ".format(d["username"]), "data":[]}

#creates the user and returns a User object to manipulate the
    user
user = User.objects.create_user(email=d["username"], username=d
    ["username"], password=d["password"])
...
```

2.11.12 Add a user to a group with Python

You usually will create the users groups manually with the Django administration site, but, if you create users from Internet, you probably will need to classify them in the already created Django groups. In the next listing you have an example of how to do it:

```
from django.contrib.auth.models import Group
from django.contrib.auth.models import User

...
#user = request.user
#or
#user = User.objects.create_user(email=d["username"], username=d
    ["username"], password=d["password"])
#or
#user = User.objects.get(username='the_username')

normal_users = Group.objects.get(name='normal_users')
normal_users.user_set.add(user)
...
```

2.11.13 Active - deactive users with Python

If you want temporally disable the login for a user, the best way is to de-active him. By default all new users are active, see the selected check in the figure 2.30, in the *Active* field. By unselecting this check, the user is inactive. This means that the user, and all his data, exists but he is unable to login. The user will be able to login again at the moment an administrator activates the user.

You can active-deactive users also with Python. In the next listing you have an example of how to deactivate an user:

```
from django.contrib.auth.models import User

...
#user = request.user
#or
#user = User.objects.create_user(email=d["username"], username=d
    ["username"], password=d["password"])
#or
#user = User.objects.get(username='the_username')

user.is_active=False
user.save()
...
```

2.11.14 Change users password with Python

You can change the user password like in the following example:

```
from django.contrib.auth.models import User
u = User.objects.get(username='the_username')
u.set_password('new password')
u.save()
```

2.12 Publish a Django app with Apache2. WSGI application

Until now, we have used the Django server, to check our back-end operations, but this is only for develop mode. To manage that a Django application answers real HTTP requests in a production environment, we have to connect the Django application with Apache HTTP server. Apache HTTP server can execute special Python applications called *wsgi* applications. To run wsgi applications you have to have installed the *Apache wsgi module*. This is already installed in the virtual machine. To connect Apache with your Django app, you have to link an Apache alias with the `wsgi.py` file in your configuration Django project. In the case of the *djdesweb* project, this file is in `/home/vagrant/apps/djdesweb/djdesweb`. You have to perform two steps:

1. Give permission to Apache to be able to read the wsgi file
2. Change the Apache configuration
3. Open the `/home/vagrant/apps/djdesweb/djdesweb/wsgi.py` module and change

```
import os
```

by

```
import os, sys
```

```
PROJEC_DIR=os.path.dirname(os.path.dirname(__file__)) # is /home  
/vagrant/apps/djdesweb  
sys.path.append(PROJEC_DIR)
```

so that the following line in the same file

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'djdesweb.  
settings')
```

could work, because the previous line looks for *djdesweb.settings*, and, if the folder `/home/vagrant/apps/djdesweb` is not in the Python path, it is not going to be found.

2.12.1 Give permission to Apache to be able to read the wsgi file

Apache belongs to the group of users *www-data*. To give Apache all permissions over the file `/home/vagrant/apps/djdesweb/djdesweb/wsgi.py`. You only have to type:

```
chmod 775 /home/vagrant/apps/djdesweb/djdesweb/wsgi.py  
sudo sudochgrp www-data /home/vagrant/apps/djdesweb/djdesweb/wsgi.py
```

The first line gives read/write/execute permission to the grup who owns the file. The second line changes the owner group of the file to *www-data*. A good explanation about what is happening is given here⁽²⁾:

Summary: 775 changes the permissions mode of a file or directory so that the owner and group has full read/write/(execute or search) access and all others have read and execute/search access. The concept of owner/group/everyone else is fundamental to unix-based files and permissions are part of this approach to basic file security.

Each number refers to the total bits of the separate parts of owner/group/other that you can have, from 0 to 7.

⁽²⁾Source: <https://www.quora.com/What-does-chmod-775-mean>

- 0 is no read/write/execute or search access
- 1 is execute/search
- 2 is write
- 3 is write/execute or search
- 4 is read
- 5 is read/execute or search
- 6 is read/write
- 7 is read/write/execute or search

So putting that all together you can see that with the 3 numbers the owner has a full 7, so does the group, and everyone else gets 5.

Being able to execute or search means different things depending on whether its a file or a directory. A directory almost always has 7, giving full access to the owner. The execute permission on a directory does not means you can execute anything but you can get into it to list its files. You can always change that if you're the owner but its very rare. A full 7 for a file usually means the owner wants to use that file to execute a process, which could be a compiled file or a script file. Execute/search access for groups and anyone else for files and directories is usually granted according to the owner's wishes.

Most directories you create yourself are 755 giving you full access and everyone else read and search access but not write access. Most ordinary files you create are 644 giving you read/write access and everyone else read access. Most executable files are also 755.

On modern Linux systems the group usually defaults to the user's own group, but this isn't always the case. Groups are used to share or restrict access so their permissions on a file or directory are important. Having the group the same as your owner's name protects you by default from others being able to change files in your directories.

For example, in this situation if its a directory, and the group is different to your group, you can still read and access the directory. If the permission was just one bit different, say 765, you couldn't access or read the directory or even edit a file in it that you owned. If the directory was 755 you could edit a file that you owned but you couldn't delete it!

On the other hand, if you took all the group privileges away and gave outsiders all privileges, so the directory was 707, you could do whatever you wanted with the files you had access to in that directory!

File and directory permissions aren't the perfect answer to system security, but they can be subtle and surprising if you don't understand their interactions with users and groups.

2.12.2 Change the Apache configuration

Apache has, by default, two configuration files: *000-default.conf* and *default-ssl.conf*. The former file is for normal http requests through the port 80. The latter file is for secure connection with https, trough the port 443. You are going to work with http connections, so you have to modify the former file. Both files are located in the folder */etc/apache2/sites-available*. You need sudo permission to be able to edit the file. In a console type `sudo gedit /etc/apache2/sites-available/000-default.conf` and add the following:

```
Header always set Access-Control-Allow-Origin "*" 
```

2.12 Publish a Django app with Apache2. WSGI application

```
####DJDESWEB####

#create a Python process with the appropriated Python interpreter
WSGIDaemonProcess desweb python-home=/home/vagrant/apps/env python-
  path=/home/vagrant/apps/djdesweb:/home/vagrant/apps/env/
  lib/python3.6/site-packages

#sets the alias who will executes the django wsgi application
WSGIScriptAlias /djdesweb/ /home/vagrant/apps/djdesweb/djdesweb/wsgi
  .py/ process-group=desweb

#gives permission to apache to read the file wsgi.py in the folder /
  home/vagrant/apps/djdesweb/djdesweb
<directory /home/vagrant/apps/djdesweb/djdesweb>
#  Options Indexes FollowSymLinks
#  AllowOverride None
#  Require all granted
  <Files wsgi.py>
    Require all granted
  </Files>
</Directory>

</VirtualHost>
```

Unlike the Django server, every time you change the Python code, or the Apache configuration, you must restart the Apache service, otherwise you will not see the changes.

```
sudo service apache2 restart
```

Open the web browser and check if the *building_select* operation works, figure 2.33. Pay attention we have changed the :8000 port by the Apache alias used in the *WSGIScriptAlias* directive (*djdesweb*) used in the Apache 000-default.conf file.

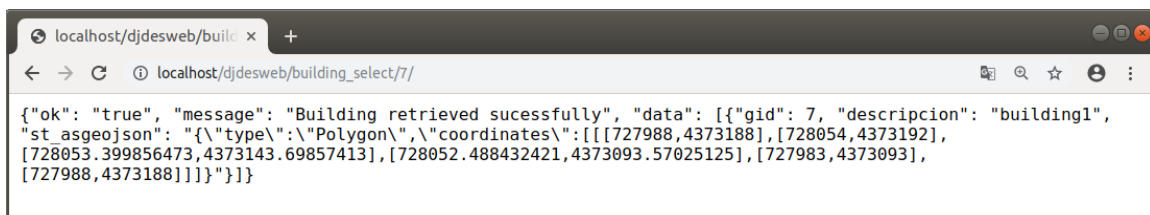


Figura 2.33: Django app running under an WSGI application, so accessible by Apache

2.12.3 Enable the Django admin site with Apache

If you try to show the Django admin site visiting the page without the Django developing server, you will get the web page but without not styles figure 2.34.

To get the admin styles you need to perform three steps:

1. Gather the static files of the site. To to this you have to perform two steps:

- a) Set the folder where to put the static files. This is done with the variable *STATIC_ROOT* in the *settings.py* module:

2.12 Publish a Django app with Apache2. WSGI application

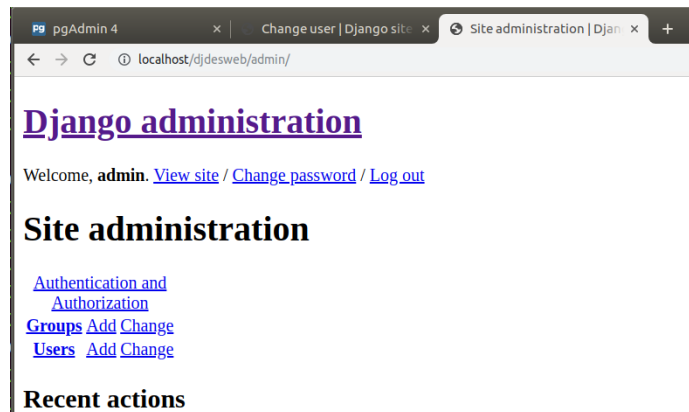


Figura 2.34: Django admin site without styles

```
STATIC_ROOT = os.path.join(BASE_DIR, 'djdesweb_static')
```

BASE_DIR is the project folder, so you need to create the folder *djdesweb_static* in */home/vagrant/apps/djdesweb*.

- b) Stop the developing server and execute the command `python manage.py collectstatic`. Django collects all the files and puts them in the subfolder *admin* figure 2.35.

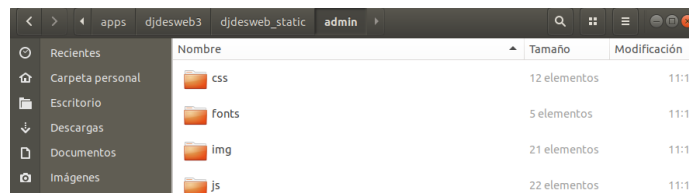


Figura 2.35: Django admin site collect static files

2. Put the static files where Apache2 can read them. This must be outside of the Django project folder, in order any user could read the raw Python code. Lets create the folder */home/vagrant/www*, and paste there the folder *djdesweb_static*.
3. Change the Apache configuration file. You have to:
 - a) Create an alias (*djdesweb_static*) that redirects to */home/vagrant/www/djdesweb_static*.
 - b) Give read permission to Apache to read the */home/vagrant/www* folder.

In the next listing you have the Apache configuration (file */etc/apache2/sites-available/000-default.conf*).

```
#configuration to get the django admin static files
alias /djdesweb_static/ /home/vagrant/www/djdesweb_static/
<directory /home/vagrant/www>
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
```

4. Set the url in the *settings.py* module to get the static files, in the variable *STATIC_URL*. To get it with the local installation of Apache:


```
STATIC_URL = 'http://localhost/djdesweb_static/'
```

Visit the admin page again and check the result, figure 2.36.

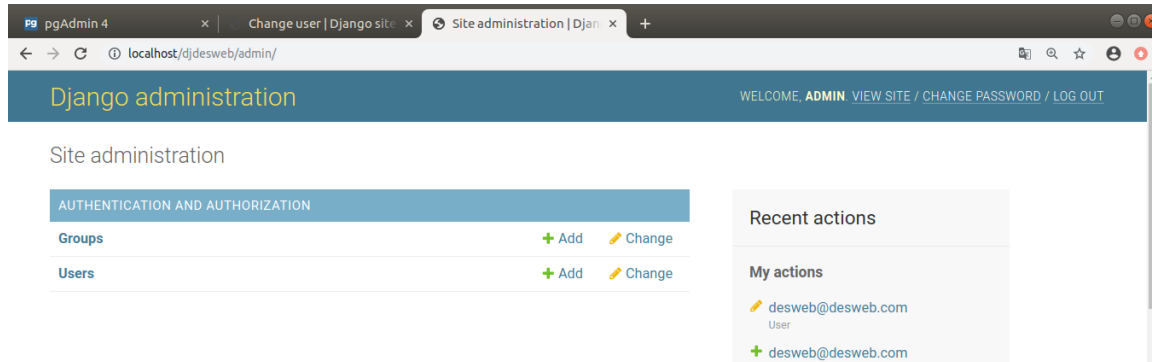


Figura 2.36: Django admin site published with Apache2 and been able to get the static files

2.13 Debugging Python WSGI applications

2.13.1 Debugging Python code. Apache file *error.log*

You must to test your Django application very well before to upload and publish in the real server. To do that you must use the Django development server by executing `python manage.py runserver` command. Nevertheless probably you will overlook some error in the developer server. If you are running a WSGI application with Apache, if the navigator shows 500 server error, means that there is a error in the Python server code (figure 2.37). In this cases it is necessary to see the file `/var/log/apache2/error.log`, where, in the last lines, the error description is written.

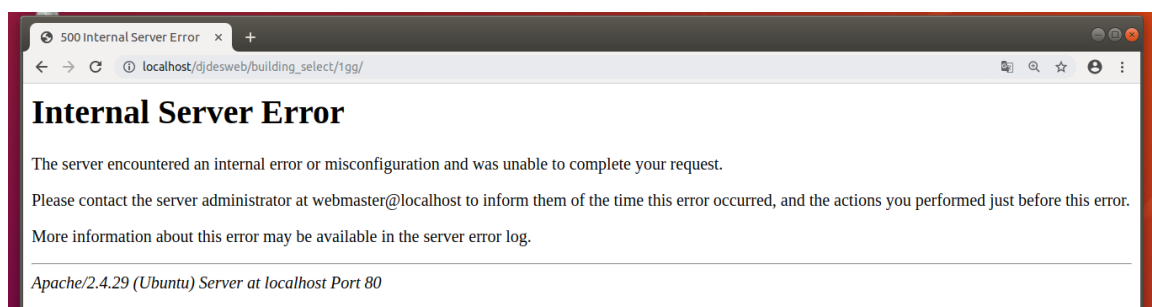


Figura 2.37: Internal server error showed when the `DEBUG` setting is set to `False`

For example, if you type in the web browser the url `http://localhost/djdesweb/building_select/1gg/`, you will get the error *Internal server error*. This is because you set in the `settings.py` of the Django project the parameter `DEBUG` to `false`. If you set in the `settings.py` of the Django project the parameter `DEBUG` to `true` you will get in the web browser the complete error description, which is a several security fail, as any user will be able to see your server configuration. If `DEBUG` is set to `false`, the correct way, you will get in the web browser the message *Internal server error*, and you will not have any clue of what happened. In this case yo have to watch the file `/var/log/apache2/error.log`, where you will have the complete error description.

2.13 Debugging Python WSGI applications

```
[Tue Apr 13 14:44:15.020725 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764]      return handler(request, *
args, **kwargs)
[Tue Apr 13 14:44:15.020730 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764]    File "/home/vagrant/apps/
djdesweb/appdesweb/views.py", line 20, in get
[Tue Apr 13 14:44:15.020734 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764]      r= buildings.select(gid)
[Tue Apr 13 14:44:15.020739 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764]    File "/home/vagrant/apps/
djdesweb/appdesweb/pyCode/buildings.py", line 76, in select
[Tue Apr 13 14:44:15.020742 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764]      cursor.execute(
query_select, [gid])
[Tue Apr 13 14:44:15.020758 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764] pycopg2.errors.
InvalidTextRepresentation: invalid input syntax for integer: "lgg
"
[Tue Apr 13 14:44:15.020762 2021] [wsgi:error] [pid 3948:tid
140322179979008] [remote ::1:54764] LINE 1: ...asgeojson(geom)
from d.buildings as t where gid = 'lgg') as ...
```

It is also possible to configure Django to send to the administrator the error description. This is good practice as you will realise of the error at the moment they have happened, and you avoid to visit the file `/var/log/apache2/error.log`.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_USE_TLS = True
EMAIL_PORT = 587
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'your account?s password'
ADMINS=[('Gaspar Mora', 'admin@upv.es')]
```

To use the Google smtp server you must enable the access to insecure applications in your Google account (figure 2.38).

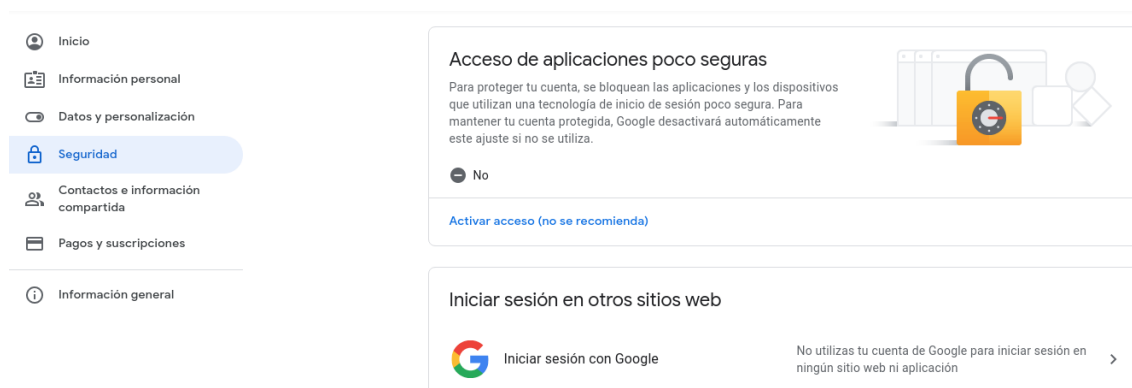


Figura 2.38: Enabling insecure access to the email Google account

2.13.2 Remote debugging with PyDev

In the developing process, you will use the Django developing server. In this case you will receive detailed error description when something is going wrong, in the web browser and in the terminal window, where the Django server is being executed. Despite that, in some cases you will not know what is going wrong, despite the error messages, probably because an algorithm is very complicated. In this cases you will need to execute the code step by step, and check the variable values as the execution is progressing. This is the remote debugging.

To remote debug with PyDev:

- Add, where you want stop the execution, the instructions of the following listing:

```
#import sys;sys.path.append(r"/opt/liclipse/plugins/org.
python.pydevd_4.5.5.201603221237/pysrc")
#import pydevd;pydevd.settrace()
```

Simply writing *pydevd*, Liclipse writes the rest of the sentences.

- In the *Debug* perspective, run the PyDev server.
- Execute the page with a web browser.

With the before steps, the execution is stopped in the line under *pydevd.settrace()* sentence, and it is possible to advance line by line in the code, seeing the variable values, (figure 2.39).

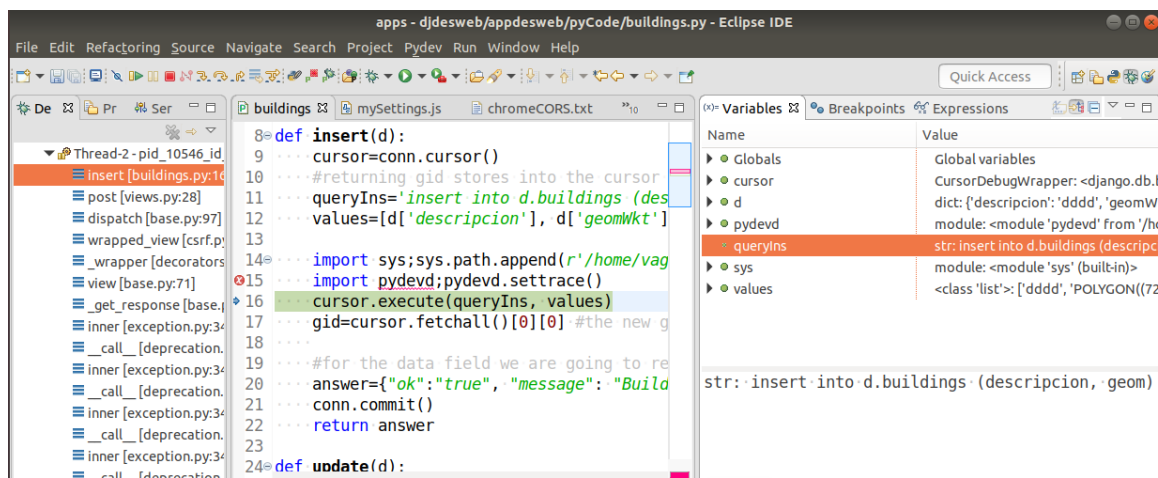


Figura 2.39: Remote debugging with Pydev

2.14 In case of error 1

You must try to solve your own problems before call anyone. This is an important part of your learning process. If you are in a company you can not call your superiors at every step. Problems are normal and you have to try to learn to solve them. At the beginning you maybe will not understand the error messages, or will not know what is happening, but it is better for you to investigate first. Errors are always the same and, when it will had happened a couple of times, you will know how to solve them. In this section you will find what to do in the case of a Python error.

2.14.1 How to make a question

Asking questions properly takes effort. If you do not make this effort, the person who you are asking the question will not do the effort to answer.

When you ask, the person who are you asking expects:

- Ask politely
- You already have investigated and tried some things. No one is going to spend their time on something that you have not spent time on yet.
- You must give all the necessary details to be able to be answered. What is happening, what you have already tried, the code necessary to be able to study the problem, and the complete error description.

In all the list of users you will be told off if you do not follow these steps.

2.14.2 How to know where the Python error is

When you are using the Django developing server, Django reports all is happening on the console. If you get an error, you will find the error description, the file and the line where the error was raised. See the next listing:

```
Quit the server with CONTROL-C.
Internal Server Error: /building_insert/
Traceback (most recent call last):
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/core/handlers/exception.py", line 34, in inner
    response = get_response(request)
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/core/handlers/base.py", line 115, in _get_response
    response = self.process_exception_by_middleware(e, request)
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/core/handlers/base.py", line 113, in _get_response
    response = wrapped_callback(request, *callback_args, **
        callback_kwargs)
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/views/generic/base.py", line 71, in view
    return self.dispatch(request, *args, **kwargs)
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/utils/decorators.py", line 45, in _wrapper
    return bound_method(*args, **kwargs)
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/views/decorators/csrf.py", line 54, in wrapped_view
    return view_func(*args, **kwargs)
  File "/home/vagrant/apps/env/lib/python3.6/site-packages/django/views/generic/base.py", line 97, in dispatch
    return handler(request, *args, **kwargs)
  File "/home/vagrant/apps/djdesweb/appdesweb/views.py", line 28, in
    post
    r= buildings.insert(d)
  File "/home/vagrant/apps/djdesweb/appdesweb/pyCode/buildings.py",
    line 12, in insert
    values=[d['descripcion'], d['geomWktt']]
KeyError: 'geomWktt'
[16/Dec/2019 17:18:35] "POST /building_insert/ HTTP/1.1" 500 92566
```

Try to solve it studying the lines above of the error line. Most of times will be enough to know what happened. For example, in the previous listing, you can see that the error was raised in the `/building_insert/` view, in the file `/home/vagrant/apps/djdesweb/appdesweb/pyCode/buildings.py`, line 28. The error is `KeyError: geomWktt`. That means that the key `geomWktt`, does not exist in the dictionary `d`. The error will be solved if you write a proper key for the dictionary `d`.

If you are unable to know what is happening, try to stop the execution in the line that caused the error an execute the call to the view again, as it is explained in the section 2.13.2, 75. In the example of the above listing, may be you do not know which are the keys of the dictionary `d`. So you can write the magic lines:

```
#import sys;sys.path.append(r"/opt/liclipse/plugins/org.python.pydev_4.5.5.201603221237/pysrc")
import pydevd;pydevd.settrace()
```

Start the Pydev debug server, in the Debug perspective, and recall the view with the same data. You can see the `d` content in the figure 2.40. The error is you have to change `geomWktt` by `geomWkt`, because is the real key name in the dictionary `d`.

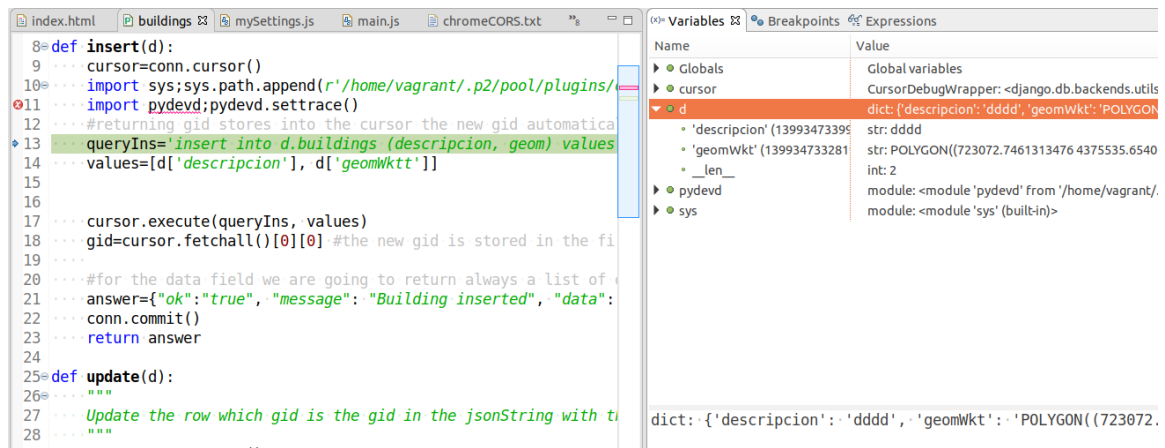


Figura 2.40: Remote debugging with Pydev 2

2.15 Geometry checks (optional)

Before to insert, or update a geometry it is very common to have to check some geometry conditions: if intersects with an other existing geometry, if it is nearer of other geometry than x distance, etc. You can check geometry conditions by selecting the geometries that accomplish that conditions with the PostGIS functions. If this selections has any result, you know before to insert or update that the geometry does not accomplish the criteria needed, there fore you can reject the geometry. In this section you will see an example of how to reject geometries than are nearer of x distance of the geometry of other layer.

2.15.0.1. Function to select the gid of the geometries nearer of a geometry

Usually you will need several geometry checks, so you should create a new module to store these functions together. In this case has been created the module `appdesweb/pyCode/libs/geometry-Checks.py`

```

from appdesweb.pyCode.libs.pgUtils import djConnect

def checkMinimumDistance(minimumDistance, layerName, geomWkt, epsg):
    """
    Selects the gid of the geometries of a layer (layerName) nearer
    than minimumDistance from the geometry geomWkt
    Executes the query:
    select gid from {layerName} where st_distance(geom,
    st_geometryFromText(...)) < minimumDistance
    @param minimumDistance: the required distance
    @param layerName: the layer which geometries are going to be
    checked
    @param geomWkt: the geometry un wkt format
    @param epsg: the epsg code of the src of the geometry and the
    layer. They must use the same epsg.
    Returns:
    - A list with a tuples with the gid of the geometries nearer
    [(10,), (25,), ...]].
    - If not geometries nearer returns an empty list []
    """
    q="""select gid
    from {layerName}
    where
    st_distance(geom,st_geometryFromText(%s,%s)) < {
    minimumDistance}""".format(
    layerName=layerName, minimumDistance=minimumDistance
    )

    djConn=djConnect()
    cursor=djConn.cursor()

    cursor.execute(q, [geomWkt, epsg])
    r=cursor.fetchall()
    print(r)
    return r

```

The first line (`from appdesweb.pyCode.libs.pgUtils import djConnect`) is a new function in the module `appdesweb.pyCode.libs.pgUtils.py`

```

from django.db import connection as djConn
def djConnect():
    #Returns the Django connection
    #The advantage is that we don't have to worry about closing the
    connection
    #Use this function instead of pgConnect, and do not use
    pgDisconnect any more
    return djConn

```

2.15.0.2. Use the geometry check function before insert or update

In the `appdesweb.pyCode.buildings.py` module, in the `insert` function, you can use the `checkMinimumDistance` function before to insert:

```
from appdesweb.pyCode.libs import pgUtils
from appdesweb.pyCode.libs import geometryChecks

def insert(d):
    conn = pgUtils.djConnect()
    cursor=conn.cursor()

    rows=geometryChecks.checkMinimumDistance(minimumDistance=20,
        layerName='d.buildings', geomWkt=d['geomWkt'], epsg=25830)
    #rows can be [] or [(10,),(20,),...]
    if len(rows)>0:
        gids=[]
        message="The new building is nearer than 20 meters from some
            other buildings: "
        for row in rows:
            message = message + str(row[0]) + ', '
            gids.append(row[0])
        message = message[:-2]
        answer={"ok":"false", "message": message, "data": [{"nearer":gids}] }
        print(answer)
        return answer

    #returning gid stores into the cursor the new gid automatically
    #created by the database as gid is serial
    queryIns=""insert into d.buildings (descripcion, geom) values (
        %s
    ,st_geometryfromtext( %s,25830)) returning gid""
    values=[d['descripcion'], d['geomWkt']]
    cursor.execute(queryIns, values)
    gid=cursor.fetchall()[0][0] #the new gid is stored in the first
    #for the data field we follow the same criteria than psycopg2:
    #firts field of first row
    answer={"ok":"true", "message": "Building inserted", "data": [{"gid":gid}] }
    conn.commit()
    #pgUtils.pgDisconnect(conn) <--You do not need to disconnect if
        you are usin the Django connection
    return answer
```

If you try to insert a building that is nearer of 20 meters to other building, the function `geometryChecks.checkMinimumDistance` will return something like this:

```
[(215,),(216,),(219,),(220,),(225,),(227,),(228,)]
```

And the function `insert` will return something like this:

```
{'ok': 'false', 'message': 'The new building is nearer than 20
    meters from some other buildings: 215, 216, 219, 220, 225, 227,
    228', 'data': [{'nearer': [215, 216, 219, 220, 225, 227, 228]}]}
```

2.15.1 Intersection check considerations

This is the most difficult concept to understand about PostGIS. You have to know:

- If you check if two geometries intersects or not with the function `st_intersects`, if the geometries are adjacent, they share a part of their perimeter, and the intersection function returns true, when you probably expect false (no intersection)
- If instead of using the function `st_intersects` you use the function `st_relate`, to check if the interior of the geometries intersects or not:

```
st_relate (geom1, geom2, 'T*****');
```

You are doing it well, this is the way to check it, but you probably will get a wrong result. Why?. Because the decimal places of the coordinates may be do not match in every decimal.

For example PostGIS the X coordinate 100.0000000000 of the geometry A is different from the X coordinate 100.0000000001 of the geometry B. There is a distance between them, if this distance is in one side, there will be a separation, and if the distance is to the other side there will be an intersection.

Of course 0.0000000001 does not make sense, X_a and X_b should be consider the same. The solution is to round all the coordinates to the same number of decimals. You should round to 1/100 times your coordinates accuracy. If you coordinates have 0.001 m of accuracy, you should round to 0.00001 m all the coordinates.

To round the coordinates PostGIS has the function `st_snapToGrid`: https://postgis.net/docs/ST_SnapToGrid.html

Before insert anything in your database you should apply this function to all the geometries the following will return the geometry but all the coordinates rounded to 0.00001 decimals:

```
st_snapToGrid(geom, 0.00001)
```

The complete

```
insert into
  tablename (field1, field2, geom)
values
  (value1, value2, st_snapToGrid(st_geometryfromtext(geomWkt,epsg)
    , 0.00001));
```

If you already have geometries you can round them with:

```
update tablename set geom = st_snapToGrid(geom, 0.00001);
```


CAPÍTULO 3

Database update through Internet. Ajax

3.1 Goals in this chapter

In this chapter the goal is that you learn how to send web form data to the server, in order to insert, delete, update or select a building, and how to manage the server answer to change the page content. Here you will learn how to:

- Create a minimal web page
- Create a form
- Link javascript code to the page
- Start the javascript code on finishing the page loading
- Connect click events on buttons to execute javascript functions, usually to send the form data
- Get the form data
- Use Ajax to send the form data to the server and wait for its response
- Change the page content to show to the user the server answer

The front-end development is the most complicated part of a geoportal. You must to type a lot of code in a lot of files. The result is a very big projects difficult to manage. As a consequence professional developers use web frameworks to develop front-ends. The most famous are Reactjs (<https://es.reactjs.org/>), Vuejs (<https://vuejs.org/>) and Angular (<https://angular.io/>). If you like web develop you must use one of these frameworks after you understand web developing philosophy, which is explained in this subject.

3.2 Create a minimal web page

A web page has two main sections:

- HEAD: Where you put the title, key words, description and link Javascript and CSS file.
- BODY: Where there is the page content. You divide the page content in sections (SECTION) or divisions (DIV)

To edit HTML JavaScript and CSS, you are going to use an other editor called Microsoft Visual Studio Code, already installed in the virtual machine.

You must use the Apache DocumentRoot folder in `/home/vagrant/www/html` to store the main page (`index.html`). Create the subfolders `css`, `externalLibs`, `js`, e `img` inside. Create a new HTML file, and call it `index.html` (figure 3.1). Paste the following code to the `index.html` file:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="description" content="Buildings management" />
  <meta name="author" content="Gaspar Mora-Navarro" />
  <meta name="keywords" content="Web develop, geoportal, postgis"
  />
  <title>Buildings management</title>
```

3.2 Create a minimal web page

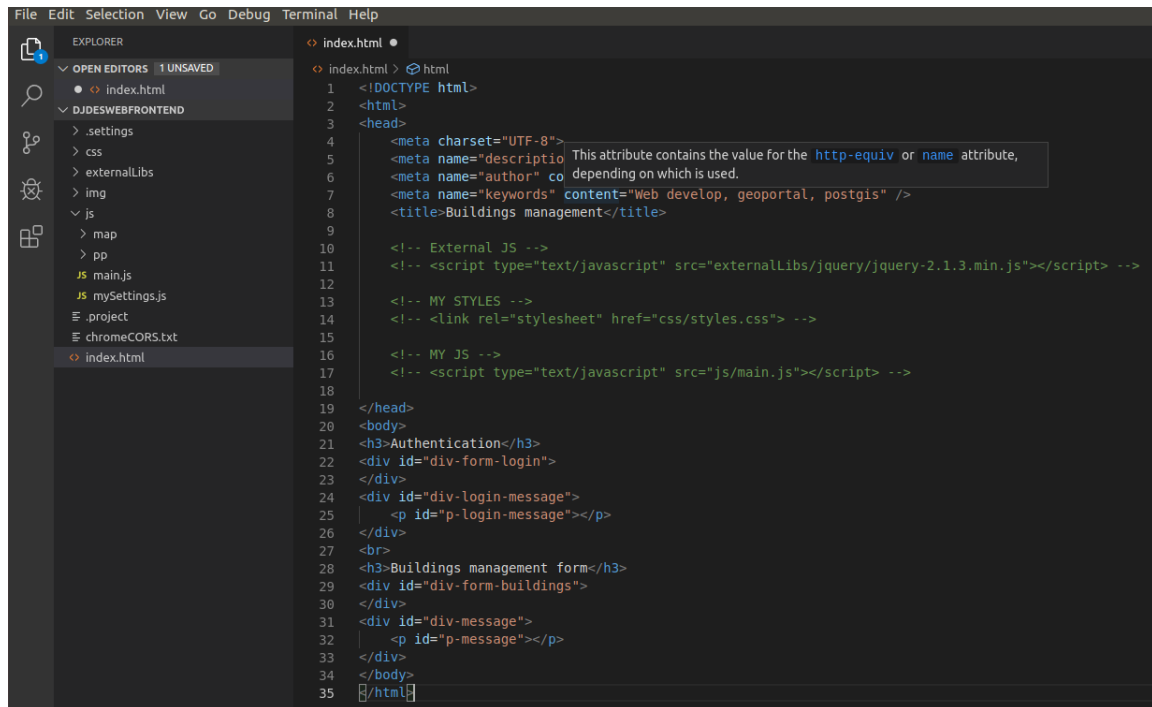


Figure 3.1: Project structure for the front-end project

```
<!-- External JS -->
<!-- <script type="text/javascript" src="externalLibs/jquery/
      jquery-2.1.3.min.js"></script> -->

<!-- MY STYLES -->
<!-- <link rel="stylesheet" href="css/styles.css"> -->

<!-- MY JS -->
<!-- <script type="text/javascript" src="js/main.js"></script>
-->

</head>
<body>
<h3>Authentication</h3>
<div id="div-form-login">
</div>
<div id="div-login-message">
  <p id="p-login-message"></p>
</div>
<br>
<h3>Buildings management form</h3>
<div id="div-form-buildings">
</div>
<div id="div-message">
  <p id="p-message"></p>
</div>
</body>
</html>
```

In the above listing you have the head, with some lines commented, and the body with a title and two empty div. This divs are identified by the keys *div-form-login* and *div-form-buildings*.

3.3 Visit the web page

Visit the page <http://localhost/>, figure 3.2.

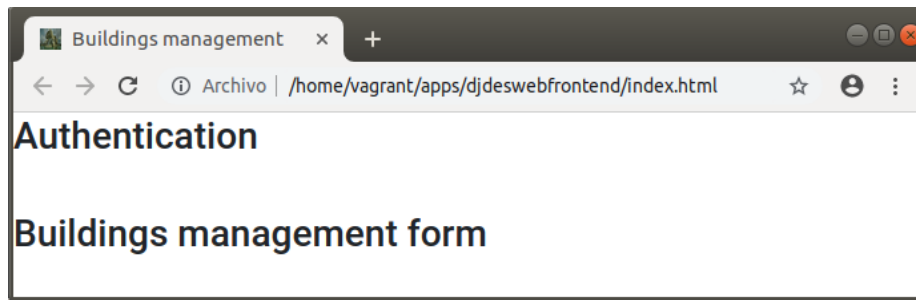


Figura 3.2: Page previsualization 1

3.4 Create form to get the user data in order to login

Now introduce the code for the form to get the user credentials. You must put the following code in the *div-form-login* div:

```
<form id="form-login" onsubmit="event.preventDefault();">
  <!-- Prevent default avoids the form be sent on pressing a
  form button
  This is necessary because we want on click a button
  -->
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" required
    minlength="4" maxlength="50" size="50">
  <br>
  <label for="password">Password:</label>
  <input type="password" id="password" name="password"
    required minlength="4" maxlength="50" size="50">
  <br>
  <button id="button-login">Login</button>
  <button id="button-logout">Logout</button>
</form>
```

In the above listing you created a form with couple of text inputs, and two buttons. The *BR* tags introduce a new blank line figure 3.3.

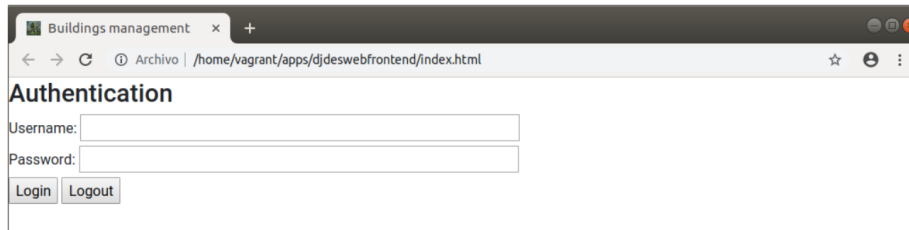


Figura 3.3: Form login preview

3.5 Styling with Bootstrap

Bootstrap (<https://getbootstrap.com/docs/5.0/getting-started/introduction/>) is a library to facilitate styling. One of the hardest task in web developing.

You have to study the Bootstrap grid first: <https://getbootstrap.com/docs/5.0/layout/grid/>

An the column alignment: <https://getbootstrap.com/docs/4.0/layout/grid/#horizontal-alignment> (despite the example is of the version 4, it works with the version 5).

In the following example you will see how it works (figure 3.4:

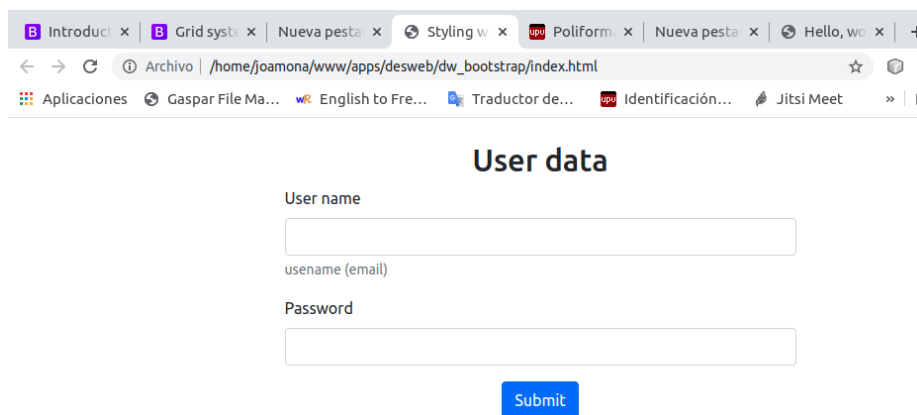


Figura 3.4: Form styling with bootstrap

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale
      =1">

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/
      css/bootstrap.min.css" rel="stylesheet" integrity="sha384-
      wEmeIV1mKuiNpC+
      IOBjI7aAzPcEZeedi5yW5f2yOq55WWLwNGmvvx4Um1vskeMj0"
      crossorigin="anonymous">
```

```

<title>Styling with Bootstrap</title>
</head>
<body>

<br>

<div class="container-fluid">
  <!-- class = container has some padding left and right
  class = container-fluid uses all the width of the
  screen
  -->

  <div class="row text-center">
    <!--class = text-center to center text inside the row-->
    <h2>User data</h2>
  </div>
  <div class="row justify-content-center">
    <!--Every col must be inside a row
    justify-content-center centers the columns on the
    screen
    -->
    <div class="col-sm-12 col-md-8 col-lg-6 col-xl-4">
      <!--12 cols are the complete width screen-->
      <!-- The class = col-sm-12 col-md-8 col-lg-6 col-xl-4 means:
      col-sm-12: if the sreen is small use 12 columns
      (complete screen)
      col-md-8: if the screen is medium sice use 8
      columns out of 12
      col-lg-6: if the screen is large use 6 columns
      col-xl-4: if the screen is extra large use 4
      columns.

      So the with of the form is variable, in function
      of the screen sice

      -->
      <form form id="form-login" onsubmit="event.
      preventDefault ();">
        <div class="mb-3">
          <label for="username" class="form-label">User
          name</label>
          <input type="email" class="form-control" id="
          username" aria-describedby="Email">
          <div id="usernameHelp" class="form-text">
          username (email)</div>
        </div>
        <div class="mb-3">
          <label for="password" class="form-label">
          Password</label>
          <input type="password" class="form-control" id
          ="password">
        </div>

```

```
        <div class="text-center">
            <!--Center the content of the div-->
            <button type="submit" class="btn btn-primary
                ">Submit</button>
        </div>
    </form>
</div>
</div>
</div>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/
    js/bootstrap.bundle.min.js" integrity="sha384-
    p34f1UUtsS3wqzfto5wAAmdvj+osOnFyQFpp4Ua3gs/
    ZVWx6o0ypYoCJhGGScy+8" crossorigin="anonymous"></script>
</body>
</html>
```

3.6 Create form to insert a building

Now introduce the code for the form to insert a building. You must put the following code in the *div-form-building* div:

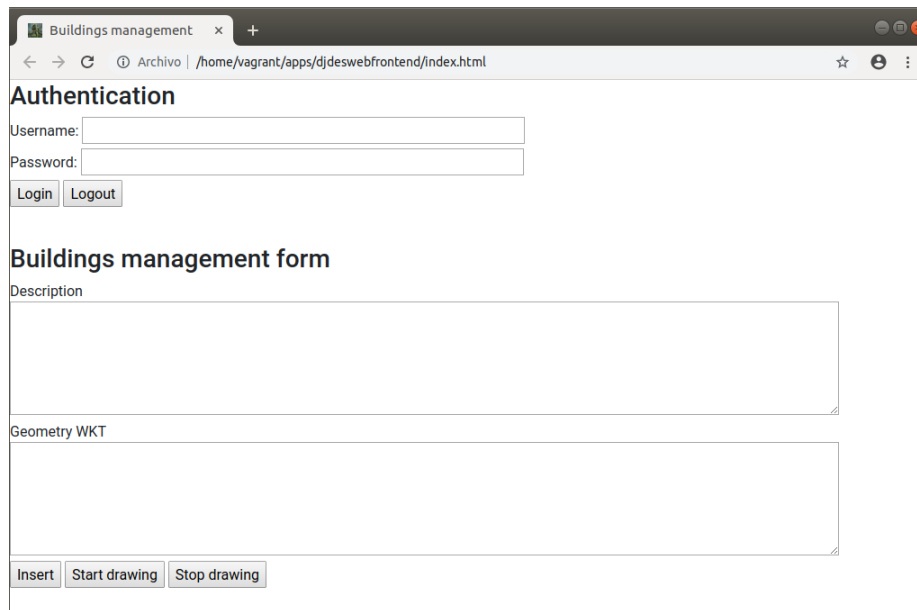
```
<form id="form-buildings" onsubmit="event.preventDefault();">
    <!-- Prevent default avoids the form be sent on pressing a
        form button
        This is necessary because we want on click a button
        -->
    Description<br>
    <textarea rows="5" cols="100" id="descripcion" name="
        descripcion"></textarea>
    <br>
    Geometry WKT<br>
    <textarea rows="5" cols="100" id="geomWkt" name="geomWkt"></
        textarea>
    <br>
    <button id="building-insert">Insert</button>
    <button id="building-start-drawing">Start drawing</button>
    <button id="building-stop-drawing">Stop drawing</button>
</form>
```

In the above listing you created a form with couple of text area inputs, and a button. See the result (figure 3.5).

3.7 Add a paragraph and a div for the future map

In the future we will need a paragraph to show the server answers and a div to show the OpenLayers map. Add the following before the end of the body of the page.

```
<div id="div-message">
    <p id="p-message"></p>
</div>
<div id="map" class="map"></div>
```



The screenshot shows a web browser window with the title 'Buildings management'. The address bar shows the URL `/home/vagrant/apps/djdeswebfrontend/index.html`. The page content is divided into two sections. The first section, 'Authentication', has a 'Username:' label followed by an input field, a 'Password:' label followed by an input field, and two buttons labeled 'Login' and 'Logout'. The second section, 'Buildings management form', has a 'Description' label followed by a large text area, a 'Geometry WKT' label followed by another large text area, and three buttons labeled 'Insert', 'Start drawing', and 'Stop drawing'.

Figura 3.5: Page with the Login and Buildings forms

3.8 Link javascript code to the page

If you press the login or insert button nothing happens. We need to link the click event of the button to a function. This is made in the following steps:

3.8.1 Link a Javascript file to the web page

The Javascript code must be in a separate files. All you js files must be organized in the `js` folder of the project. To link the `js/main.js` file to the web page you must use the following line in the `HEAD` page, you only have to uncomment it:

```
<script type="text/javascript" src="js/main.js"></script>
```

As you can see, you must reference the file location in its relative position to the html file location.

3.8.2 First JS code. Window on load event

Create the file `js/main.js` and put the following code in it:

```
/**
 * This file starts all the Javascript functionality
 */

function mainInit(){
    alert("I am loaded and ready to work");
}

window.onload = function() {
    mainInit();
};
```

Save the file and reload the page figure 3.6.

The explanation is the following. The navigator gets the page and starts rendering its components (paragraphs, divs, sections, ...). Also ask for the js, css, and img files. The sentence `window.onload`

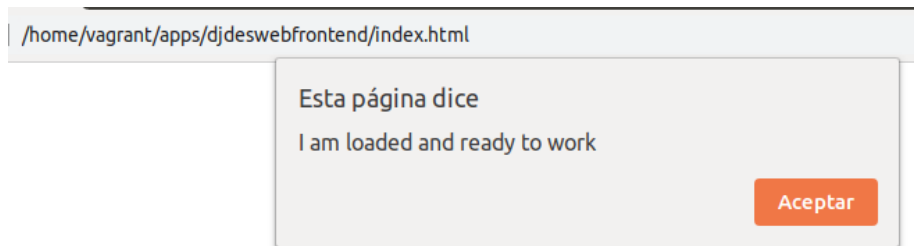


Figura 3.6: Message when the page is completely loaded

`= function() {mainInit();}` ensures that any Javascript code is going to be executed until the page is completely loaded. This prevent the Javascript code tries to change something that is not still in the page. You always must do this in all your projects, if do not, you will get an error and the page will stop loading.

So that line, when the page is completely loaded, and all the page elements are currently created, executes a function called *mainInit*, who shows a message window.

3.8.3 Link button click events to a function

You need to execute a function to send the form data only when the user clicks the *building-insert* button. The *id* and *name* properties set in the html page, are useful to identify the elements with Javascript or CSS. The *id* and *name* properties must be unique in all the web page.

To link the click event on the button *building-insert* to a function you must wait until the page be totally loaded. The best pace to put the sentence is in the *mainInit* function as it is not going to be executed until the page be loaded.

Change the *js/main.js* code to the following:

```
function login(){
}
function logout(){
}
function buildingInsert(){
    console.log("I promess. I will send the form data");
}

function mainInit(){
    console.log("I am loaded and ready to work");
    document.getElementById("button-login").addEventListener("click", login);
    document.getElementById("button-logout").addEventListener("click", logout);
    document.getElementById("building-insert").addEventListener("click", insertBuilding);
}

window.onload = function() {
    main_init();
};
```

3.8 Link javascript code to the page

Reload the page and click the insert button tree times. Nothing seems to happen but if extract the Google Chrome Developer Tools (figure 3.7), you will see that the message *I am ready to work* has been showed once, and the message *I promess. I will send the form data*, has been showed three times (figure 3.8).

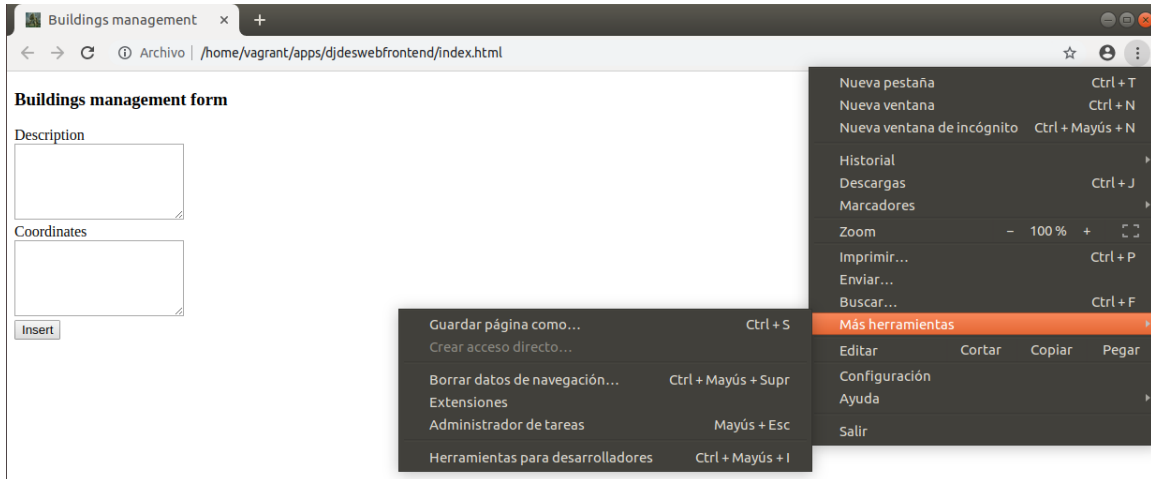


Figura 3.7: Show Google Chrome Developer Tools

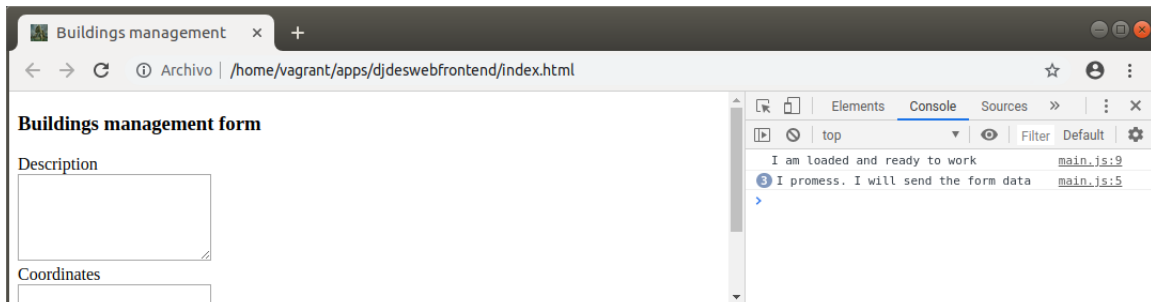


Figura 3.8: Message in the console tag of the web developer tools

This demonstrates that the function *mainInit* is executed only once, and when the page is loaded, and the function *buildingInsert* each time the user press the button *building-insert*.

3.9 What to do in case of error 2

You will find more help in the section 3.15, 106. In this section you will find only help for your current knowledge.

3.9.1 See the console messages

First step do not panic. You are not blind. The web browser, or Django, will tell you what is exactly happening. Open the developer tools, and go to the *Console* tab. You will see the error and the line where it happened (figure 3.9).

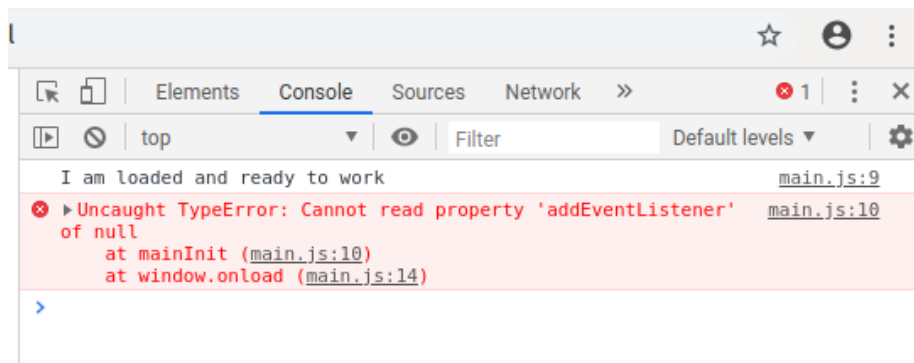


Figura 3.9: Console with the error message and the lines where the error was triggered

If you click over the first line number indication you will see the js code and the first line that triggered the error, figure 3.10.

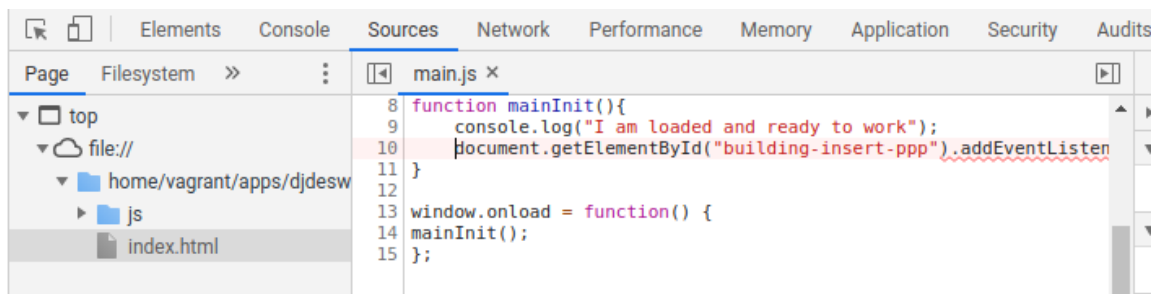


Figura 3.10: First line where the error was triggered

In this case the error *Cannot read property addEventListener of null* is a very common mistake. This means that the object before *addEventListener* is null. Like null objects do not have the method *addEventListener* the execution can not continue. In other words *document.getElementById(building-insert-ppp)* is null, what means there is not any html object with the id *building-insert-ppp*. The 80% of the times that you call the teacher is because this error. Fix the error typing the id of an existing element in your page.

3.9.2 Stop the JavaScript execution

You can also stop the execution whatever you want, execute the code line by line and see the local variable values. In the next listing we are going to make some calculations to show you how to stop the code and see the local variable values. Change the *buildingInsert* function code for the following:

```
function buildingInsert() {
    var a=5;
    var b=10;
    var c;
    c=a+b; //this will trigger an error
    console.log("The result is" + d)
    console.log("I promess. I will send the form data");
}
```

If you reload the page you will see that the message *I am loaded and ready to work* appears in the console. So the page is properly loaded. But on click on the insert button, you will see the following error on the console window figure 3.11.

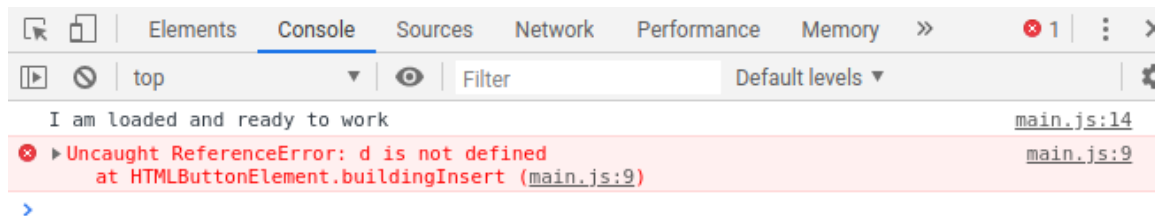


Figura 3.11: Reference error

The error is telling you that *d* is not defined. That means that there is not any *d*=something before the line 9 in the main.js file. Lets stop the execution a little bit before to see the local variables and run the code step by step. Click on the error line number (figure 3.12).

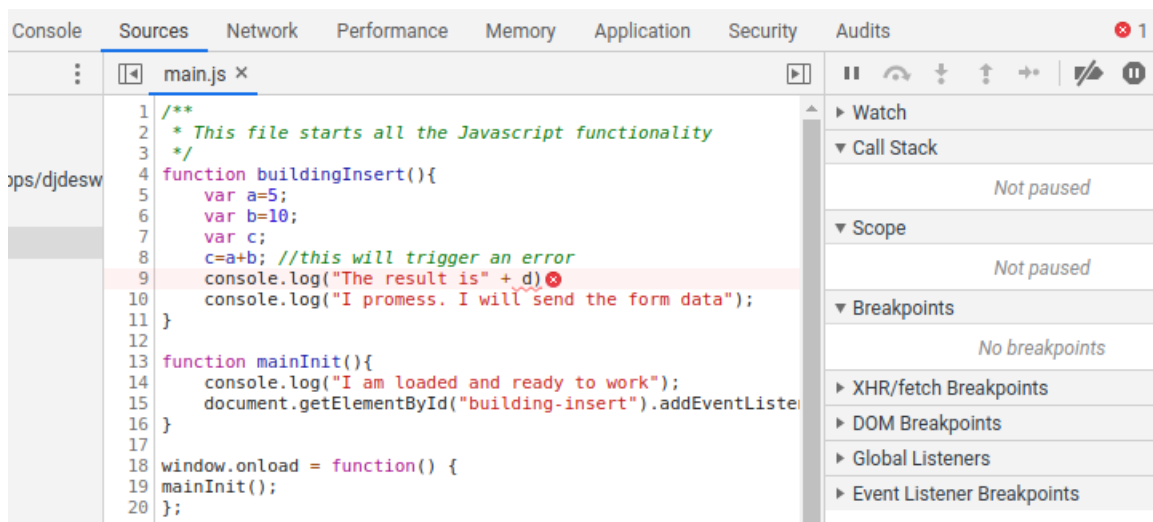


Figura 3.12: Reference error

Click on the left side of the code window some lines before the error line, but inside the the function who made the error, in the figure the fifth line.

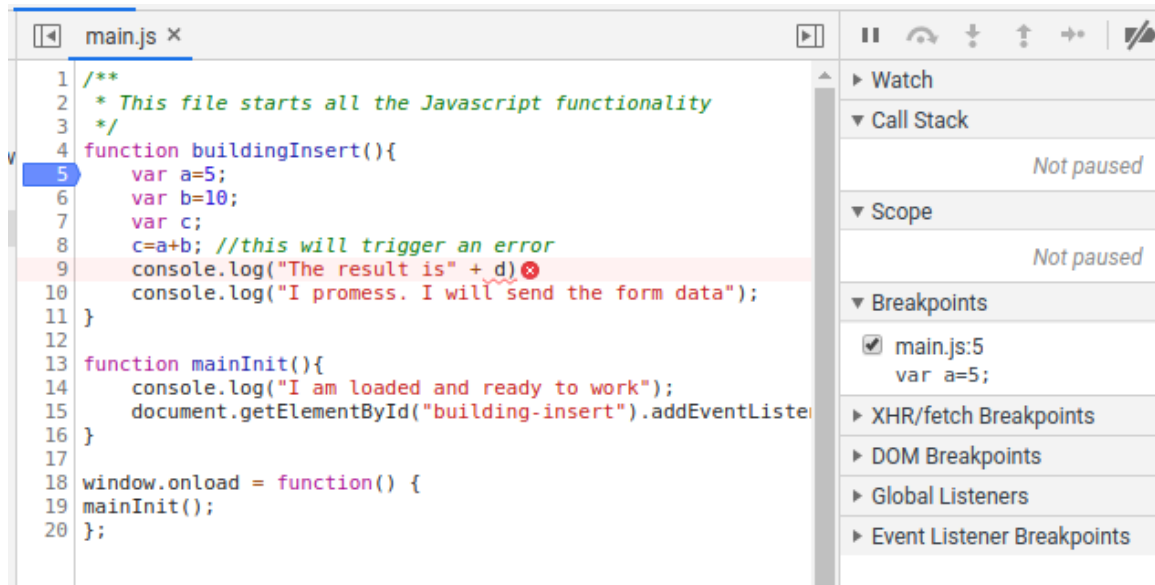


Figura 3.13: Stop the code execution in a line

Now you can click on the insert button to trigger the function again and to see how the execution is stopped. You can advance the execution by pressing one of the buttons in the control bar figure 3.14. To execute line by line press the curved arrow.



Figura 3.14: Buttons to advance the execution

After three clicks on the curved arrow you can see the local variable values in the *Local* section figure 3.15.

Once solved the problem, remove the stop you must click on the same line number where you clicked to stop the execution.

3.9.3 Check if all the files are being loaded

It is very common to make a mistake in the path or name of a js file in the *head* of the web page. To check if all the files are being properly loaded, on the web browser, seen the page, press the keys *control* and, keeping the *control* pressed, press *u*. You will see the html code of the page. click one by one in all the files linked to the page in the *head* section. If in any one of them you receive the message *Not found* means that the path of the file name have a mistake.

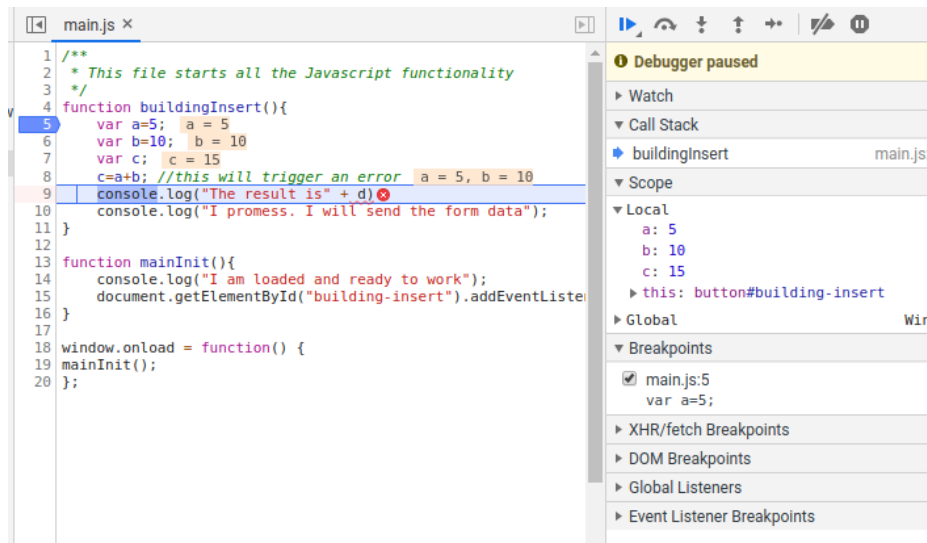


Figura 3.15: Local vars values

3.9.4 Check the order of the JavaScript files

An other common mistake working with js code is that you have to import the js file in order. That is that you have to import first the definition of the things that you use latter. If you try to use a function that is imported in a file which is behind, so is imported later, you will get the error *not defined*. For example if you have a file called *js/mySettings.js*, where you define the global variable *MAP*, you must import it, in the head of the html file, before any function use it. If you use *OpenLayers* class (*ol*) in a js file, you have to import the *OpenLayers* library before, etc. In the next listing the order of import is not casual. First the external libraries are imported, later the global variables, latter the files with function definition, and to finish, the js files that currently use the libraries and functions imported before. Usually who start all the work is a function inside the *js/main.js* file, so this file is imported the last.

```
<!-- EXTERNAL JS -->
<script type="text/javascript" src="externalLibs/jquery3.4.1/jquery.
  min.js"></script>
<script type="text/javascript" src="externalLibs/ol/v6.1.1-dist/ol.
  js"></script>
<script type="text/javascript" src="externalLibs/ol/v6.1.1-dist/ol-
  layerswitcher.js"></script>
<script type="text/javascript" src="externalLibs/bootstrap4.3.1/
  bootstrap.min.js"></script>

<!-- MY JS -->
<script type="text/javascript" src="js/mySettings.js"></script>
<script type="text/javascript" src="js/map/mapDraw.js"></script>
<script type="text/javascript" src="js/map/mapMain.js"></script>
<script type="text/javascript" src="js/main.js"></script>
```

3.10 Create an interactive navigation menu with Bootstrap and JavaScript

In this section you are going to create the following menu figure 3.16:

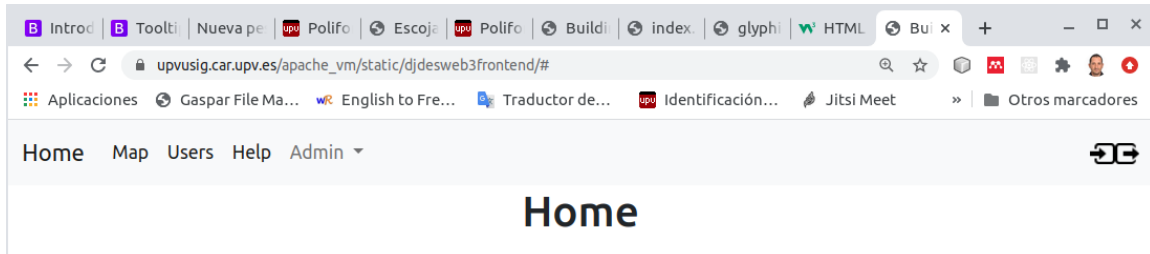


Figura 3.16: Navigation menu with Bootstrap

The menu has some JavaScript code that allow show-hide *divs* in the web page. Check it in the url: <https://gisserver.car.upv.es/desweb/>. The code is too long to be copied here so I am going to explain the main ideas. You can download the code in Poliformat. Download the file *bootstrap_menu.zip*.

The navigation bar base code has been copied from the first example of <https://getbootstrap.com/docs/5.0/components/>. The idea of showing and hiding *divs* in the web page is the following:

1. Create the navigation bar and assign an *id* to each link you want.
2. Create a *div* which will contain the group of *divs* that you want show-hide. Assign *ids* to every one.

```
<div id="div-main" class="container-fluid">
  <div id="div-home">
    <div class="row text-center">
      <h1>Home</h1>
    </div>
  </div>

  <div id="div-users">
    <div class="row text-center">
      <h1>Users</h1>
    </div>
  </div>

  <div id="div-help">
    <div class="row text-center">
      <h1>Help</h1>
    </div>
  </div>
  ...
</div>
```

3. Link the clicks on the menu options by using the *ids* to the functions that are going to show only the corresponding *div*:

```
function showDivHome() {
  libs_general_hideAllDivsInDivExceptOne("div-main", "div-home")
}
```

3.10 Create an interactive navigation menu with Bootstrap and JavaScript

```
}
function showDivMap(){
    libs_general_hideAllDivsInDivExceptOne("div-main", "div-map
    ")
}
function showDivUsers(){
    libs_general_hideAllDivsInDivExceptOne("div-main", "div-
    users")
}
function showDivHelp(){
    libs_general_hideAllDivsInDivExceptOne("div-main", "div-help
    ")
}
function showDivLogin(){
    libs_general_hideAllDivsInDivExceptOne("div-main", "div-
    login")
}
function showDivLogout(){
    libs_general_hideAllDivsInDivExceptOne("div-main", "div-
    logout")
}

function linkMenuEvents(){
    document.getElementById("menu-home").addEventListener("click
    ", showDivHome);
    document.getElementById("menu-map").addEventListener("click
    ", showDivMap);
    document.getElementById("menu-users").addEventListener("
    click", showDivUsers);
    document.getElementById("menu-help").addEventListener("click
    ", showDivHelp);
    document.getElementById("menu-login").addEventListener("
    click", showDivLogin);
    document.getElementById("menu-logout").addEventListener("
    click", showDivLogout);
}

function linkButtonEvents(){
    /*
    document.getElementById("button-login").addEventListener("
    click", login);
    document.getElementById("button-logout").addEventListener("
    click", logout);
    document.getElementById("building-insert").addEventListener
    ("click", insertBuilding);
    */
}

function mainInit(){
    //alert("I am loaded and ready to work");
    linkMenuEvents();
    linkButtonEvents();
    showDivHome();//on init shows only the div-home div
}
```


3.10 Create an interactive navigation menu with Bootstrap and JavaScript

```
window.onload = function () {  
    mainInit();  
};
```

4. You need now the function *libs_general_hideAllDivsInDivExceptOne* does the work of hiding all the divs contained in a div except one. Here is the function:

```
/**  
 * Hides all the firts children divs in a div. Does not modify de  
   internal divs of the children.  
 * @method function libs_general_hideAllDivsInDiv  
 * @param {str} divName - string with the id of the div that  
   contains all the divs to hide  
 * @return none  
 */  
function libs_general_hideAllDivsInDiv(divName) {  
    var selector="#" + divName + ">div";//select all divs wich  
        the parent is divname (only the first level)  
    var divs= document.querySelectorAll(selector);  
    var div;  
    var n=divs.length;  
    var i;  
    for (i = 0; i < n; i++) {  
        div=divs[i];  
        div.style.display = 'none';//hide the div  
    }  
}  
  
/**  
 * Hides all the divs in a div, except one  
 * @method libs_general_hideAllDivsInDivExceptOne  
 * @param {str} divParentName - string with the id of the div  
   that contains all the divs to hide and the div to show  
 * @param {str} divName - string with the id of the div in the  
   divParentName to show  
 * @return none  
 */  
function libs_general_hideAllDivsInDivExceptOne(divParentName,  
divName) {  
    libs_general_hideAllDivsInDiv(divParentName);  
    var selector="#" + divName;  
    var div= document.querySelector(selector);//selects only one  
    div.style.display = 'block';  
}
```

5. You can put these function anywhere. I recommend you to put it in *js/libs/general.js*, and load that file in the head of the page index.html:

```
...  
<link href="js/externalLibs/bootstrap5/bootstrap.min.css"  
    rel="stylesheet">  
<script type="text/javascript" src="js/libs/general.js"></  
script>  
<script type="text/javascript" src="js/main.js"></script>
```

...

3.11 Get the form data

To send the form data you have first to get the form data. The server is waiting for something like this for the url *app_login*:

```
'{"formData":{"username":"the_username", "password":"the_password"}}'
```

or

```
'{"username":"the_username", "password":"the_password}"'
```

And for the url *'building_insert*:

```
'{"formData": {"descripcion": "edificio 5", "geomWkt": "POLYGON
((728155.94273754325695336 4373095.60990698169916868,
728217.62150992138776928 4373098.00261797849088907,
728200.6066761618712917 4373038.98241337575018406,
728154.34759687830228359 4373040.84341081790626049,
728155.94273754325695336 4373095.60990698169916868))"}'
```

or

```
'{"descripcion": "edificio 5", "geomWkt": "POLYGON
((728155.94273754325695336 4373095.60990698169916868,
728217.62150992138776928 4373098.00261797849088907,
728200.6066761618712917 4373038.98241337575018406,
728154.34759687830228359 4373040.84341081790626049,
728155.94273754325695336 4373095.60990698169916868))"}'
```

In the above listings you can see that you can send the form data in a json directly, or a json that contains a dictionary that contains the key *formData*, which value is the form data. In both cases, the function `appdesweb.pyCode.libs.general.getPostFormData` will return a dictionary with the form data.

3.11.1 Get a form control value

You can get the value of a form control in this way:

```
var form = document.getElementById("form-buildings");
var description= form.elements["descripcion"].value;
```

The above listing gets first the form object. You can have many forms so `document.getElementById` gets the form who *id* property its *form-buildings*. There are other ways to select from the page, called DOM (Document Object Model):

```
var x = document.getElementsByClassName("example"); //gets all
elements from class="example"
var x = document.getElementsByName("fname"); //gets the element
which name="fname"
var x = document.getElementsByTagName("LI"); //gets all elements LI
```

With the form object in a variable, the above listing gets the *descripcion* control value from the *elements* collection controls.

3.12 Use Ajax to send the form data to the server and wait for its response

3.11.2 Set a form control value

You can set the value of a form control in this way:

```
var form = document.getElementById("form-buildings");
form.elements["descripcion"].value = "new value";
```

The above listing gets first the form object. You can have many forms so *document.getElementById* gets the form who *id* property its *form-buildings*. Latter *form.elements["descripcion"]* gets the form control called *descripcion*. Setting the *value* property you can set the text that the control shows.

3.11.3 Get all form control values at once

You can get a json strig with all the form data with the following code:

```
var f = document.getElementById("form-buildings");// extract the
    form
var formData = new FormData(f) //generates an object from
    FormData class
var data =Object.fromEntries(formData); //generate a kind of
    dictionary
var js= JSON.stringify(data); //returns the dictionary
    transformed into a json string
```

The variable *js* contains exactly what the *building_insert* view is expecting. Something like:

```
'{"formData":{"descripcion": "edificio 5", "geomWkt": "POLYGON
((728155.94273754325695336 4373095.60990698169916868,
728217.62150992138776928 4373098.00261797849088907,
728200.6066761618712917 4373038.98241337575018406,
728154.34759687830228359 4373040.84341081790626049,
728155.94273754325695336 4373095.60990698169916868))}'
```

You can put this code in a function:

```
function getFormData(formId) {
    var f = document.getElementById(formId);// extract the form
    var formData = new FormData(f) //generates an object from
        FormData class
    var data =Object.fromEntries(formData); //generate a kind of
        dictionary
    var js= JSON.stringify(data); //returns the dictionary
        transformed into a json string
    return js;
}
```

3.12 Use Ajax to send the form data to the server and wait for its response

3.12.1 Login the user

The goal here is to send the building form data to the Django view *building_insert* because is what this view is expecting, but this view expects the user to be authenticated. So first step is to sent the user credentials to the view *app_login* in order to login the user.

The JavaScript library JQuery (<https://jquery.com/>) has a method, called *ajax*, to send data to am url by POST or GET. This method also receives a function to receive the server answer. This function

3.12 Use Ajax to send the form data to the server and wait for its response

is called *callback*. The delivery of the data and the server answer may take from some seconds to minutes. The *callback* function remains waiting for the answer, and the user is able to continue using the page. In the next listing you have how to recover the data from the form *form-login*, send it to the server, wait for the server response and show the server response to the user in a paragraph. Put this function in the file `js/main.js`

Using Ajax to send data to the server to login the user

```
function login(){
    var f = document.getElementById("form-login");
    var formData = new FormData(f)
    var data =Object.fromEntries(formData);
    var js= JSON.stringify(data);

    $.ajax({
        url: URL_DJANGO_API + 'app_login/', //url where the data is
        sent
        type: "POST", //send method POST or GET
        dataType: 'json', // type of data to send
        data: js, //the data to send
        contentType: 'application/json;charset=UTF-8', // content
        type
        success: function (data){ //callback function. It will wait
            for the server answer
            console.log(data); //logs the server answer in the
                console

            //Shows the server message and the gid of the new
                building on the paragraph p-message
            document.getElementById("p-login-message").innerHTML =
                data.message;
        },
        error: function (err){
            console.log(err); //in case of error logs the error in
                the console
        }
    });
}
```

The symbol `$` means the JQuery library, so you first have to load that library, before to be used. This means, that the load of the JQuery library must be done before the load of the `js/main.js` file. Simply uncomment the following lines from the *head* section of the *index.html* file:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="description" content="Buildings management" />
    <meta name="author" content="Gaspar Mora-Navarro" />
    <meta name="keywords" content="Web develop, geoportal, postgis"
        />
    <title>Buildings management</title>

    <!-- External JS -->
    <script type="text/javascript" src="externalLibs/jquery/jquery
        -2.1.3.min.js"></script>
```

```

<!-- MY STYLES -->
<!-- <link rel="stylesheet" href="css/styles.css"> -->

<!-- MY JS -->
<script type="text/javascript" src="js/main.js"></script>

</head>

```

Reload the page by pressing the keys *ctrl + f5*, fill the login form and press the *Login* button. You will get a CORS error.

In the above listing, the variable `URL_DJANGO_API` is a global variable. We will talk about JS global variables later, in the section 3.14. For now you only have to know that its value for this case is `http://localhost:8000`, which is the url for the Django development server.

3.13 Solve the CORS error of Google Chrome

The CORS error is a security measure of Google Chrome. Chrome try to avoid you page get content from other origins different than the page that you are visiting. In this case, you are visiting `http://localhost/djdesweb_static/djdeswebfrontend/index.html`, and you sent data to `http://localhost:8000`. They are different origins so Chrome blocks it figure 3.17.

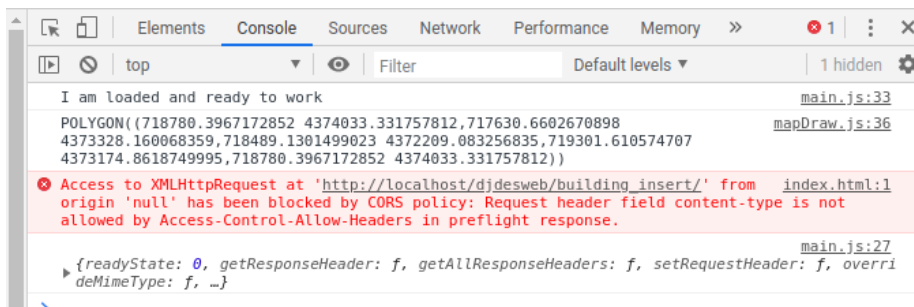


Figura 3.17: Chrome CORS error

To solve this problem while you are developing, you need to create a folder in your *home* directory, for example *chrome*. Close Chrome and open a new terminal. Launch Chrome with this command:

```
google-chrome --disable-web-security --user-data-dir="[ /home/vagrant /chrome] "
```

In Windows is similar. Please check Chrome location.

```
"C:\Program Files\Google\Chrome\Application\chrome.exe" --disable-
web-security --disable-gpu --user-data-dir=%LOCALAPPDATA%\Google\
chromeTemp
```

You will see an error message but you can continue using Chrome. Visit the page `http://localhost/djdesweb_static/djdeswebfrontend/index.html` again, and try to send the login form again. Any CORS message error is given, as it is disabled. If all your code is correct, in the developer tools, in the *Network* tab, you will see the *Status 200*. This means all was ok (figure 3.18).

Of course this is only for development. In a real application you will serve the Django application with Apache. Writing in the Apache configuration file (`/etc/apache2/sites-available/000-default.conf`), at the beginning, the following line, your page will be able to get information from different origins with out to disable the Chrome security.

3.14 Use a Javascript settings file to configure the Javascript application. *mySettings.js*

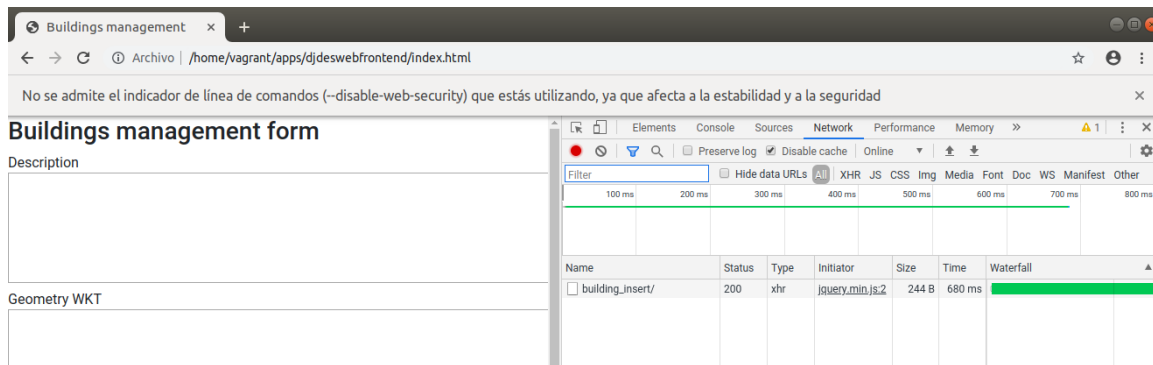


Figura 3.18: Chrome CORS security disabled for development

Header always set Access-Control-Allow-Origin "*" "

The complete Apache configuration file is:

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
```

Header always set Access-Control-Allow-Origin "*" "

```
####DJDESWEB####
WSGIDaemonProcess desweb python-home=/home/vagrant/apps/env python-
    path=/home/vagrant/apps/desweb/djdesweb:/home/vagrant/apps/env/
    lib/python2.7/site-packages
WSGIScriptAlias /djdesweb/ /home/vagrant/apps/djdesweb/djdesweb/wsgi
    .py/ process-group=desweb
<directory /home/vagrant/apps/djdesweb/djdesweb>
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>

</VirtualHost>
```

You can send the Ajax request to Apache instead of to the Django development server if by changing the value of the global variable `URL_DJANGO_API` to `http://localhost/djdesweb/`

3.14 Use a Javascript settings file to configure the Javascript application. *mySettings.js*

In a real application you will perform a lot of http requests with Ajax. This means that if you change the url where to send the requests, if you do not take care of this, you will have to modify this url in every Ajax request. To avoid this I recommend you to use a global variable. In this way you can change the url o the server at one point. To develop a real application you will use probably this urls to access to the Django application:

- `http://localhost:8000:` to access to the Django development server

3.14 Use a Javascript settings file to configure the Javascript application. *mySettings.js*

- `http://localhost/djdesweb/` to access to the Django API with Apache in local
- `https://realserverDomain/djdesweb/` to access to the Django API with Apache in a real server

The same will happen with the Geoserver location:

- `http://localhost:8080/geoserver:` to access to Geoserver in local
- `http://localhost/geoserver` to access to Geoserver in local thought an Apache proxy
- `https://realserverDomain/geoserver/` to access to Geoserver in a real server thought an Apache proxy

So at least this urls will change depending on the stage of your development. The most intelligent way of change the urls at once is to have three modes of use of the application:

- Mode 1: to use Django developing server and Geoserver in localhost
- Mode 2: to use Apache server and Geoserver in localhost
- Mode 3: to deploy your geoportal in a real server.

I use the file `js/mySettings.js` file to set the mode. In the following listing you have the possible js content:

```
var mode=1

switch(mode) {
case 1:
    //Local develop mode
    var URL_DJANGO_API='http://localhost:8000/';
    var URL_GEOSERVER='http://localhost:8080/geoserver/';
    break;
case 2:
    //local develop mode with apache
    var URL_DJANGO_API='http://localhost/djdesweb/';
    var URL_GEOSERVER='http://localhost:8080/geoserver/';
    break;
case 3:
    //production mode
    var URL_DJANGO_API='https://upvusig.car.upv.es/desweb_g1/';
    var URL_GEOSERVER='https://upvusig.car.upv.es/desweb/geoserver
        /';
    break;
}

//### SETTINGS FOR OPENLAYERS THAT WE WILL USE IN THE FUTURE ###
var MAP; //openlayers map
var MAP_DRAW_POLYGON; // Draw interaction. Global so we can remove
    it later
var SOURCE_DRAW = new ol.source.Vector({wrapX: false}); //needed for
    draw
var VECTOR_DRAW = new ol.layer.Vector({source: SOURCE_DRAW}); //The
    layer were we will draw
```

Lo load the file `js/mySetting.js` put the following file in the *head* section of the *index.html* file:

```
<script type="text/javascript" src="js/mySettings.js"></script>
```

3.14 Use a Javascript settings file to configure the Javascript application. *mySettings.js*

3.14.1 Login

Reload the page. Now all the pieces are in place you will be able to login: you have done the following:

1. You have an API rest view in the `http://localhost:8000/app_login` url.
2. The view `http://localhost:8000/app_login` is waiting for a POST request and a json with `user` a `password` keys.
3. Your Apache2 installation has access to the `index.html` file, which, on load, loads also the files `jquery.js`, `mySettings.js` and `main.js`, in this order.
4. On load the file `main.js`, on finishing the load of all the page elements, the function `mainInit` is automatically executed, which connects the click button events to the corresponding functions.
5. On click in the `Login` button an Ajax request is sent by POST to the view `http://localhost:8000/app_login`.
6. The server receives the request, sends it to the `app_login` view who checks the user and password. If it matches, stores the session ID in the database and sends a json dictionary to the client in the answer. The answer includes also a cookie whit the session ID. Thanks to that, the user, in every request will send the session ID in a cookie. This is done automatically. Django, also automatically, in every request gets this session ID and searches it in the database, in order to know if he was already authenticated.
7. The `success` parameter of the Ajax function is executed automatically on receiving the server answer. This function is called `callback function`. This function receives the `data` parameter, which contains the server answer. The `data` variable is an object variable. This object has the same property names than the dictionary that sent the server. Because that `data.message` in JavaScript is the `answer['message']` in Python.
8. The content of a paragraph in the page is changed from the callback function.

You can see the result of the `login` operation in the figure 3.19

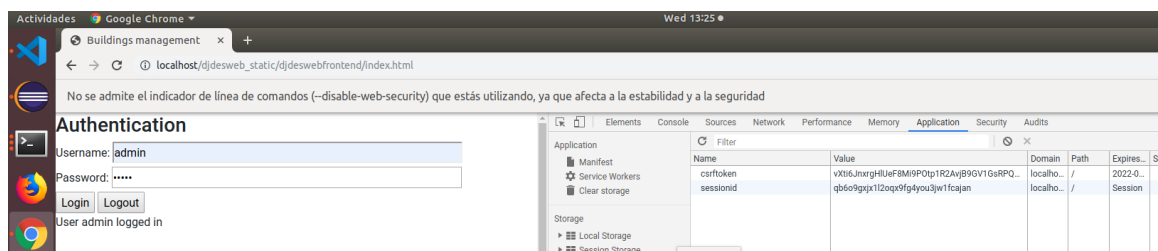


Figura 3.19: Login answer, and session ID cookie

3.14.2 Logout

In the following listing you have the code to logout. Pay attention that you send a *GET* request.

Using Ajax to send data to the server to insert a building

```
function logout(){
    $.ajax({
        url: URL_DJANGO_API + 'app_logout/', //url where the data
        is sent
        type: "GET", //send method POST or GET
        dataType: 'json', // type of data to send
        data: '', //the data to send
        contentType: 'application/json;charset=UTF-8', // content
        type
        success: function (data){ //callback function. It will wait
            for the server answer
            console.log(data); //logs the server answer in the
            console

            //Shows the server message and the gid of the new
            building on the paragraph p-message
            document.getElementById("p-login-message").innerHTML =
            data.message;
        },
        error: function (err){
            console.log(err); //in case of error logs the error in
            the console
        }
    });
}
```

3.14.3 Insert a building

In the following listing you have the code to be able to insert a building, once you are logged in:

Using Ajax to send data to the server to insert a building

```
function buildingInsert(){
    var f = document.getElementById("form-buildings");
    var formData = new FormData(f)
    var data =Object.fromEntries(formData);
    var js= JSON.stringify(data);

    $.ajax({
        url: URL_DJANGO_API + 'building_insert/', //url where the
        data is sent
        type: "POST", //send method POST or GET
        dataType: 'json', // type of data to send
        data: js, //the data to send
        contentType: 'application/json;charset=UTF-8', // content
        type
        success: function (data){ //callback function. It will wait
            for the server answer
        }
    });
}
```

```

console.log(data); //logs the server answer in the
  console

  //Shows the server message and the gid of the new
  building on the paragraph p-message
  document.getElementById("p-message").innerHTML = data.
    message + ". gid " + data.data[0].gid;
  f = document.getElementById("form-buildings").reset();
  //clean the form
},
error: function (err){
  console.log(err); //in case of error logs the error in
    the console
  }
})
}

```

3.15 What to do in case of error 3

Now you are sending data to the server, you need to know how to check where are you sending the data, what are you sending and what is the server responding. All the answer to these questions are in the Web developer tools of Chrome.

If the page does not work as you expect, you must extract the Web developer tools of Chrome an to see the messages in the *Console* tab, as was explained in the section 3.9, 91. Apart of that you must check:

3.15.1 Where I am sending the data

To know that you must open the tab *Network*, figure 3.20. In the figure you can see three requests. The second request is in red. That means that there was an error.

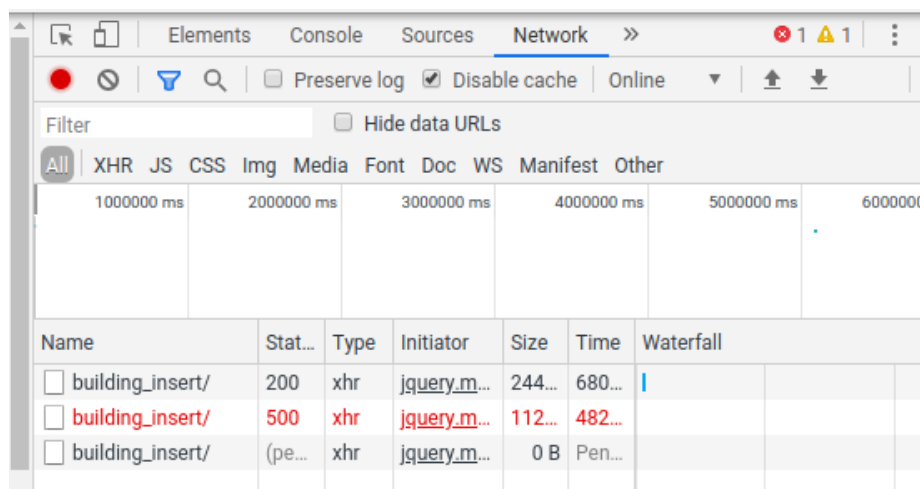


Figura 3.20: How to see the requests that my web does

To know where I sent the request click over the request and go to the *Headers* tab. In the above part you will see the *Request url*. In the figure 3.21, http://localhost:8000/building_insert/. There is where you are sending the data. You also can see the method, in this case *POST*.

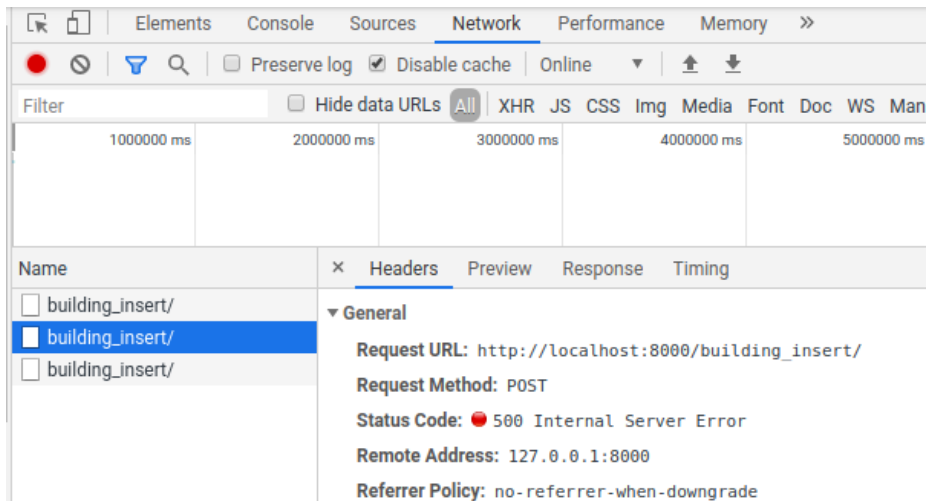


Figura 3.21: How to see the where I am sending the data

3.15.2 What I am sending to the server

Many times you will get an error because you are sending to the server something that the server is not expecting. You first have to know at which view of your API are you sending the data. After that what is your view is expecting, and later to check what are you sending. This last step is done, in Chrome developer tools, click into the *Network* tab, click into the request you want to check, and later into the *Headers* tab. At the bottom of that tab you have the *Request Payload* field, figure 3.22. There you can see what are you sending. In the figure a json with two fields: *descripcion* and *geomWkt*.

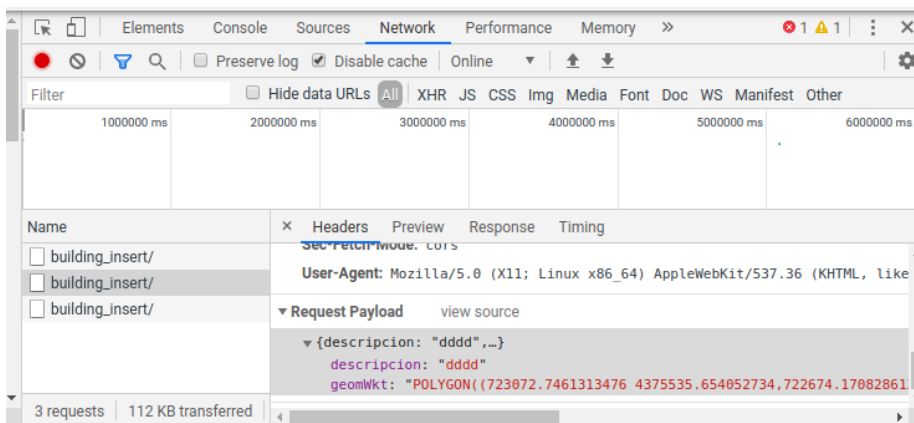


Figura 3.22: How to see what I am sending to the server

3.15.3 What is the server responding

As your js code uses the server responses, in case of error, you must check if the server is answering what are you expecting.

In Chrome developer tools, click into the *Network* tab, click into the request you want to check, and later into the *Preview* or the *Response* tab.

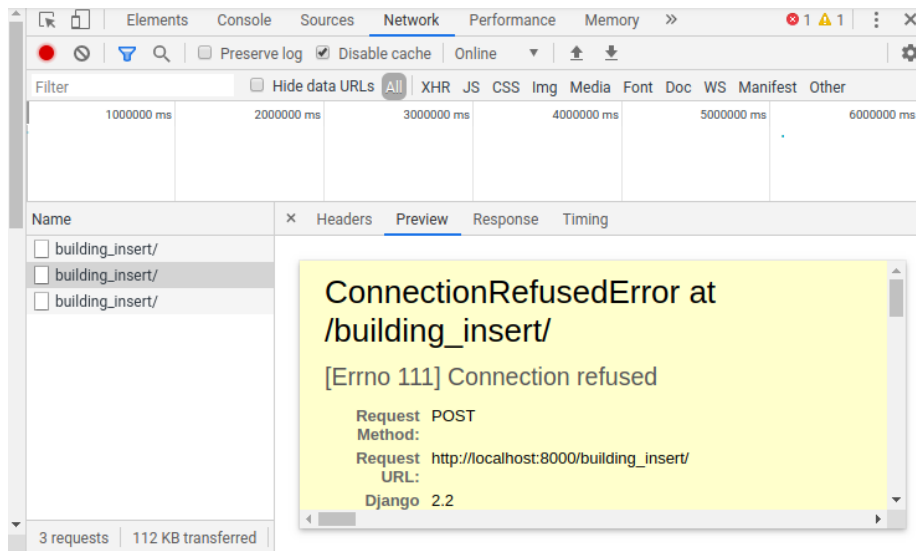


Figure 3.23: How to see the server response

In this case, the problem is that, for whatever reason the server is not running. You have to run `python manage.py runserver` again. Remember, to run your Django application you have first to activate the Python virtual environment.

3.15.4 My JavaScript code does not refresh

Some times you will note that the web browser is not loading the last version of the css, or js, files. This is because the web browser saves in its memory, the files already loaded, in order not to ask them again and again. This is called cache. You need the last version of all files as you are developing and you want to see the changes in the page. You have to disable the cache. To do this, check the *Disable Cache* option in the web developer tools, figure 3.24.

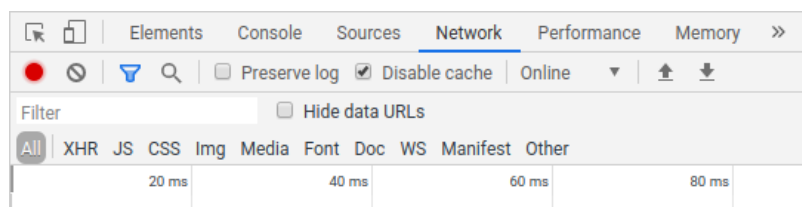


Figure 3.24: Disable Chrome cache to use the last version of the js code

3.16 Whether to use Django developing server or Apache server

You must use the Django developing server with the project settings `DEBUG` variable set to `True` while you are developing. This is the mode 1 in your `mySettings.js` file. This mode will give you in the console, and in the web browser, all the message error details.

Once you web is finished you have to test it with Apache in local mode. The `DEBUG` variable must be still `True` in your Django settings module, and mode 2 in your `js/mySettings.js` file.

When you deploy your application in a real server you must set `DEBUG` to `False`, in your Django settings module and, the mode 3 in your `js/mySettings.js` file.

3.17 Change the page content to show to the user the server answer

This task is done by the callback function of the listing 3.14.3. That function is executed in case of the request was answered as success (code 200). The callback function receives one, and only one, argument: the server answer transformed in an Javascript object. The server sends a dictionary, and this dictionary is automatically converted into a Javascript object. Each dictionary key is now a property of the object.

3.18 Create an Apache alias for the web page

To be able to serve a static page in your home folder with Apache, you have to do something similar to what you did to serve the Django wsgi file. First you have to change the Apache configuration in `/etc/apache2/sites-available/000-default.conf`. Add the following in the virtual environment:

```
###DJDESWEBSITEFRONTEND###
Alias /djdeswebfrontend/ /home/vagrant/www/djdesweb_static/
    djdeswebfrontend/
```

Restart the Apache service, to reload the new configuration:

```
sudo service apache restart
```

Now you have to change the mode of use in `js/mySettings.js` to 2. You can open Chrome as usually, without avoid csrf protection. Visit the page at `http://localhost/djdeswebfrontend/`. You can visit the page and send requests to the API because:

- The index page that you are visiting and the Django API are at the same location `http://localhost`
- Apache sets the header `Access-Control-Allow-Origin *`, due to the following configuration in the file `/etc/apache2/sites-available/000-default.conf`.

```
Header always set Access-Control-Allow-Origin "*"
```

3.19 Hide the JavaScript code

There is not way to hidden the javascript code. You never have to write in it sensible information: user names, passwords, etc.

The most used method is to obfuscate the code with special programs: this introduces special characters, removes tabs, enters and comments. You can use for example: `http://www.javascriptobfuscator.com/Javascript-Obfuscator.aspx`.

An other thing is to use the following

Instead of

```
<html lang="es">
```

You can use

```
<html lang="es" oncontextmenu="return false;" onkeydown="return
false">
```

This prevent that text in the page could be copied and the keys ctrl + u show the page code

3.20 Exercise 2.

3.20.1 Part 1. Test. Value 2 points

In this test you can be asked about Apache configurations, to type js functions, or HTML code:

- How to connect a WSGI program with Apache.
- About Django: urls, views, users managements and authentication.
- Send Ajax requests, getting the values to send from a form
- To link click events on buttons with functions
- To change the value of form controls
- To read the value of form controls
- To change paragraphs content
- Create a html form

For example you can be asked for create a web page, that load several js files. The page must have a form and, in the js code you will have to calculate something, getting the information from the form and putting the result in a paragraph or an other control.

Also you can be asked about:

- What is useful *window.onload = function()* for
- About Apache aliases or ports
- About Linux permissions to grant Apache permission to read or execute something
- What is happening, giving an error case
- What is *window.selectElementByid*
- What is a callback function

3.20.2 Part 2. Project. Value 1 point. Create a web page to update the tables of your database.

The goal of this exercise is to create a web page to insert rows in the tables of your database, with Ajax and using your Django API. You have to manage at least three tables, and at least two of them must have a geometry field, and not of the same type of geometry. The mandatory operation is to insert in at least three tables. Delete, update and select operations are optional in the web. If you are doing the Spatial Information Distribution subject, you must use at least three tables of the database created there, about one of the INSPIRE themes, to connect both subjects. Perform the following steps:

- Create a web page in a new Static Web Project of Eclipse.
- Add to the web at least three forms to insert in at least three tables. **With a minimum of 4 fields each form.**
- Use Ajax to authenticate users.
- Use Ajax to insert the data put into the forms in the corresponding tables. Do not allow unauthenticated users to modify the database, only select.
- You must give a message, changing a paragraph content, to alert the user that all was ok or not. If all was ok you must to show the gid of the new row inserted.
- If the insert was ok, clear the form
- You have to visit your page with Apache, and this page has to use the Django API in **production mode**, that is, you must not use the Django developing server. This force you to configure Apache to publish both your Django API and your web. You must also set the adequate permissions to Apache over the files.

Delivery:

- Upload the complete Eclipse projects, compressed as *exercise2.zip*, to the subject shared space in Poliformat.
- Upload a file called *groupMembers.txt*, where you specify name and surname of all the members of the group.
- All the members have to upload the same two files: *groupMembers.txt* and *exercise2.zip*.
- You will show the project to the teacher. You have to have prepared a demonstration.
- You will have to answer correctly the teacher questions about the code to obtain the whole note of the exercise.

CAPÍTULO 4

Create a map with OpenLayers 6.1.1

4.1 Goals

In this chapter our goal is to build a map with two base layers (PNOA, and Cadastre) and the layer *Buildings*. All these layers will be added to the map from WMS service. We will have a button to draw a new building in the map, and on draw-end, get the WKT definition of the building and put it automatically into the form. After the building be drawn, the building is only in OpenLayers, but if the user press the button *Insert*, the form will be sent to the server, the building will be inserted into the database, so will appear in the WMS buidings layer service.

4.2 Download and install the libraries

To use OpenLayers you need the OpenLayers js and css file. You can add them to the page from its original CDN like this:

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/gh/openlayers/
  openlayers.github.io@master/en/v6.1.1/css/ol.css" type="text/css
">
<script src="https://cdn.jsdelivr.net/gh/openlayers/openlayers.
  github.io@master/en/v6.1.1/build/ol.js"></script>
```

But the recommendable way is to download the files, save them to the server, and load them from your own server. Download and install the following libraries:

```
<!-- EXTERNAL STYLES -->
<link rel="stylesheet" href="externalLibs/ol/v6.1.1-dist/ol.css
">
<link rel="stylesheet" href="externalLibs/ol/v6.1.1-dist/ol-
  layerswitcher.css">
<link rel="stylesheet" href="externalLibs/bootstrap4.3.1/
  bootstrap.min.css">

<!-- EXTERNAL JS -->
<script type="text/javascript" src="externalLibs/jquery3.4.1/
  jquery.min.js"></script>
<script type="text/javascript" src="externalLibs/ol/v6.1.1-dist/
  ol.js"></script>
<script type="text/javascript" src="externalLibs/ol/v6.1.1-dist/
  ol-layerswitcher.js"></script>
<script type="text/javascript" src="externalLibs/bootstrap4.3.1/
  bootstrap.min.js"></script>
```

jQuery (<https://jquery.com/>) is a common requirement for other libraries, and Bootstrap (<https://getbootstrap.com/docs/4.0/getting-started/introduction/>) is a library to avoid to use CSS. I recommend you to use Bootstrap, instead of CSS when possible. Bootstrap allow you to create forms, menus, sidenavs, etc, easily and with a standard style (<https://www.w3schools.com/bootstrap4/>). We are not going to cover this in this course.

The LayerSwitcher is a button to allow the user to switch on-off the layers.

4.3 Create the WMS service of the layer buildings

To create a WMS service from a PostGis layer with Geoserver there are a lot of tutorials, for example https://geoserver.geo-solutions.it/edu/en/adding_data/postgis_lay.html. It is not going to be explained here.

In the next section you have to know that the WMS service for the layer buildings is available in the address <http://localhost:8080/geoserver/wms?>, in the workspace *desweb2019*, and with the name *buildings*.

As you are going to publish your geopotat in a real server, you will have also to publish a WMS service of your layers. You will have to repeat your configurations in Geoserver in a real server located in <https://geoshape.upvusig.car.upv.es/geoserver/>. In this server you will be forced to publish your layers in the workspace called *desweb2019*, so it is a good idea to use this workspace name now in your virtual machine. In this way you will not have to change your js code to access to the WMS layer, except for the server name.

4.4 Create a map with the Spanish Cadastre and The Spanish ortophoto (PNOA)

The map is going to be placed in a *div* tag. You need the following in your code:

```
<div id="map" class="map"></div>
```

Next you need to set the size of the map. Create the file *css/styles.css* and put inside the following code:

```
@charset "UTF-8";

.map {
  width: 100%;
  height: 600px;
  background: #f8f4f0;
}
```

The above listing selects all the objects of the class *map* and sets the with, the height and the background color.

Now load the css file from the page

```
<!-- MY STYLES -->
<link rel="stylesheet" href="css/styles.css">
```

We want to create a map after the page had been loaded, so we are going to call a function to create the map in the function *js/main:mainInit* (that is a new notation to indicate where are the functions. That indicates that the *mainInit* function is in the *js/main.js* file).

```
//file js/main.js
function mainInit(){
  console.log("I am loaded and ready to work");
  document.getElementById("building-insert").addEventListener("
    click", buildingInsert);
  mapMain(); //initializes the map in the div id="map"
}
```

Now create the function *mapMain* in a new file; *js/map/mapMain.js*. This will execute *mapMain()* every time the web page be loaded or reloaded. Copy the following code. The code is explained as comments in the code. The result can be seen in the figure 4.1.

4.4 Create a map with the Spanish Cadastre and The Spanish ortophoto (PNOA)

```
//file js/map/mapMain.js
/**
 * Here is where the map is started., with the function mapInit()
 * The function inimap is called from the mainInit function in js/
   main.js
 */

function mapMain(){
  //Map projection
  epsg25830 = new ol.proj.Projection({
    code: 'EPSG:25830',
    // The extent is used to determine zoom level 0.
    // Recommended values for a
    // projection's validity extent can be found at http://
    // epsg.io/.
    extent: [716682.702,4365814.329,732380.437,4376383.664],
    units: 'm'
  });
  ol.proj.addProjection(eps25830);

  //style for the vector layer used to draw
  //OL3 allow points, lines and polygons at the same layer,
  //so the style specifies an style for each type of geometry
  // fill: fill color for polygons
  // stroke: type of line for lines
  // image: for points. Allow different symbols and images
  var vector_draw_style = new ol.style.Style({
    fill: new ol.style.Fill({
      color: '#D7DF01'
    }),
    stroke: new ol.style.Stroke({
      color: '#DF013A',
      width: 3,
      lineJoin: 'round'
    }),
    image: new ol.style.Circle({
      radius: 4,
      fill: new ol.style.Fill({
        color: '#DF013A'
      })
    })
  });

  //VECTOR_DRAW is a global variable, defined in js/mySettings.js
  //The layer were we will draw
  VECTOR_DRAW.setStyle(vector_draw_style);
  VECTOR_DRAW.setOpacity(0.5);

  //Layer PNOA. Orthophoto aerea. To add other layers you only
  // have to copy this
  //layer definition and change the url of the WMS service, and
  // the layer names to load
  var lyr_pnoa = new ol.layer.Tile({
    source: new ol.source.TileWMS({
```

4.4 Create a map with the Spanish Cadastre and The Spanish ortophoto (PNOA)

```
        url: 'http://www.ign.es/wms-inspire/pnoa-ma',
        params: {"LAYERS": "OI.OrthoimageCoverage", 'VERSION':
            "1.3.0", "TILED": "true"},
    })),
    title: "PNOA-MA"
});

WMS_BUILDINGS_LAYER = new ol.layer.Tile({
    source: new ol.source.TileWMS({
        url: URL_GEOSERVER + 'wms?',
        params: {"LAYERS": "desweb:buildings", 'VERSION': "1.3.0",
            "TILED": "true"},
    })),
    title: "Buildings"
});

//Adds the mouse coordinate position to the map
var mousePositionControl = new ol.control.MousePosition({
    coordinateFormat: ol.coordinate.createStringXY(4),
    projection: 'EPSG:25830',
    // comment the following two lines to have the mouse position
    // be placed within the map.
    //className: 'custom-mouse-position',
    //target: document.getElementById('mouse-position'),
    undefinedHTML: '&nbsp;';
});

//Map definition. The variable MAP is a global var. All the
//global variables
//are named in upperletters and placed in js/mySettings.py
MAP = new ol.Map({
    controls: ol.control.defaults({
        attributionOptions: /** @type {olx.control.
            AttributionOptions} */ ({
            collapsible: false
        })
    }).extend([mousePositionControl]),
    target: 'map', //the map will be placed in the div id=map
    renderer: 'canvas',
    layers: [lyr_pnoa, WMS_BUILDINGS_LAYER, VECTOR_DRAW], //the
        layers are added here
    view: new ol.View({
        projection:epsg25830, //the projection of the map is set
            here
        maxZoom: 28, minZoom: 1,
        center: [724950.649,4371212.645], //the initial center of
            the map
        zoom: 2 //the initial zoom
    })
});

//Layer swicher definition
var layerSwitcher=new ol.control.LayerSwitcher({
```

4.4 Create a map with the Spanish Cadastre and The Spanish ortophoto (PNOA)

```
        tipLabel: 'Leyenda'  
    });  
    //adds the layer swicher to the map  
    MAP.addControl(layerSwitcher);  
  
}
```

Now you have to load the file `js/map/mapMain.js` before the file `js/main.js`

```
<script type="text/javascript" src="js/map/mapMain.js"></script>
```

Please note that the map projection is EPSG:25830, and the extent is [716682.702, 4365814.329, 732380.437, 4376383.664]. This extent is fitted to Valencia. If you have data in other place, you will need to change the projection and the extent. OpenLayers does not show anything outside the extent.

```
//CHANGE THE MAP PROJECTION  
epsg25830 = new ol.proj.Projection({  
    code: 'EPSG:25830',  
    // The extent is used to determine zoom level 0.  
    // Recommended values for a  
    // projection's validity extent can be found at http://  
    epsg.io/.  
    //CHANGE THE EXTENT  
    extent: [716682.702, 4365814.329, 732380.437, 4376383.664],  
    units: 'm'  
});
```

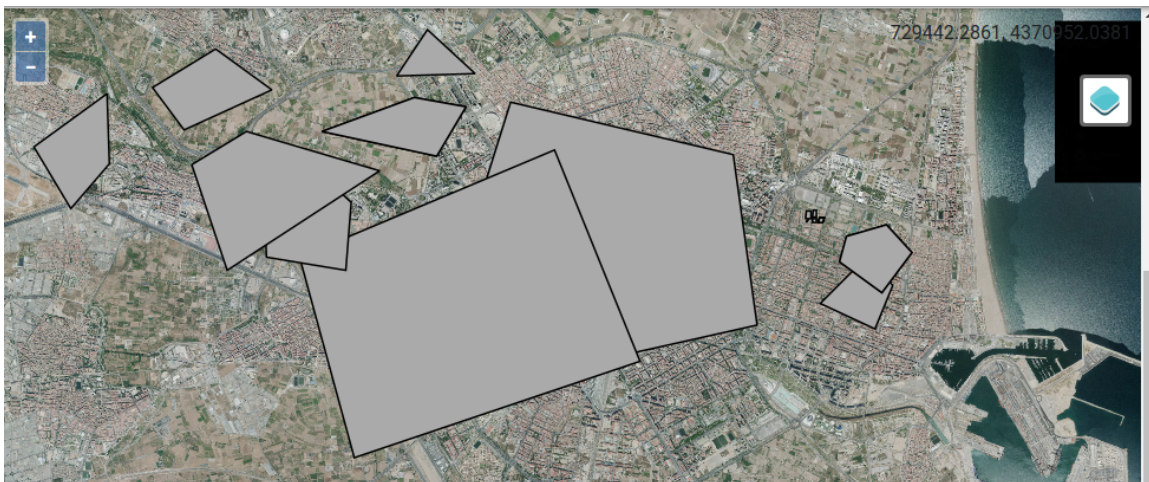


Figura 4.1: OpenLayers map

4.5 How to use the OpenLayers examples

Important note. In the OpenLayers examples, they import OpenLayers classes. For example:

```
import Map from 'ol/Map';
import View from 'ol/View';
import TileLayer from 'ol/layer/Tile';
import OSM from 'ol/source/OSM';
import TileWMS from 'ol/source/TileWMS';
```

And later they create objects directly from the classes. For example:

```
var map = new Map(...)
var view = new View(...)
var tileLayer = new TileLayer(...)
var osm = new OSM(...)
var tileWms = new TileWMS(...)
```

To import classes like in the previous listing you have to do some other tutorials, and use *nodejs* (<https://nodejs.org/es/>). For simplicity, in this course we do not import anything, in order avoid to use nodejs. All the classes are available in the file (ol.js) loaded in the *head* of the web page. To access to the same classes that in the examples, you have to use the following system, changing in the above listing the slashes by points:

```
var map = new ol.Map(...)
var view = new ol.View(...)
var tileLayer = new ol.layer.Tile(...)
var osm = new ol.source.OSM(...)
var tileWms = new ol.source.TileWMS(...)
```

4.6 Draw polygons in the map

4.6.1 Add the draw interaction to the map

To draw things into the map, you have to add an interaction to the map. You are going to draw polygons in the layer. Remember that we defined the following variables in the file `js/mySettings.js`:

```
var MAP; //openlayers map

var MAP_DRAW_POLYGON; // Draw interaction. Global so we can remove
it later
var SOURCE_DRAW = new ol.source.Vector({wrapX: false}); //needed for
draw
var VECTOR_DRAW = new ol.layer.Vector({source: SOURCE_DRAW}); //The
layer were we will draw
var WMS_BUILDINGS_LAYER; //Our WMS layer
```

As they are global, because they are not defined inside any function, they are accessible from any function. If you need to access to a variable from many functions, a common way of achieve it is to create a global variable. You must use the minimum global variables possible.

In the previous listing you can see the variable `VECTOR_DRAW` which is a layer, and the variable `SOURCE_DRAW`, which is the object who contains the geometries of the layer. This object is of the class *source*. To add geometries to a layer, you have to add them to its *source* object. In the following listing you have how to add a draw interaction to the map, linked to the `SOURCE_DRAW` variable.

```
//file js/map/mapDraw.js

function addDrawPolygonInteraction() {
  /*Possible values for tipo_geom:
  *      "Point", "LineString", "Polygon"
  *      MAP_DRAW_POLYGON, SOURCE_DRAW and MAP are global variables,
  *      defined in
  *      mySettings.js, so that are accesible from any function
  * */
  MAP_DRAW_POLYGON = new ol.interaction.Draw({
    source: SOURCE_DRAW, //source of the layer where the
    poligons will be drawn
    type: /** @type {ol.geom.GeometryType} */ ('Polygon') //
    geometry type
  });

  //adds the interaction to the map. This must be done only once
  MAP.addInteraction(MAP_DRAW_POLYGON);
}

```

Put the previous listing in the file *js/map/mapDraw.js*.

4.6.2 Enable or disable the draw interaction

Calling the function of the previous listing you will add the draw interaction to the map to draw polygons. The interaction must be added only once. You can latter enable or disable de draw interaction with the following functions. Add them to the file *js/map/mapDraw.js*.

```
//file js/map/mapDraw.js

//Enables the polygons draw
function enableDrawPolygons() {
  MAP_DRAW_POLYGON.setActive(true);
}

//Disables the polygons draw
function disableDrawPolygons() {
  MAP_DRAW_POLYGON.setActive(false);
}

```

4.6.3 Clear the content of the vector layer

Wen you draw elements, they remain in the OpenLayers memory, in the layer VECTOR_DRAW. The elements of the VECTOR_DRAW layer are in the VECTOR_SOURCE component of the VECTOR_DRAW. To clear the vector layer you have to call to the method *clear()* of the VECTOR_SOURCE object. Add them to the file *js/map/mapDraw.js*.

```
//file js/map/mapDraw.js

//Clear the vector layer
function clearVectorLayer() {
  SOURCE_DRAW.clear();
}

```

4.6.4 Reload a WMS Layer

Once a geometry has been added to the postgres layer, in the database, you have to reload the wms layer, in order to show the new object.

```
//file js/map/mapDraw.js
//Reload Buildings WMS Layer
function reloadBuildingsWmsLayer(){
    WMS_BUILDINGS_LAYER.getSource().updateParams({"time": Date.now()})
}
}
```

You can access to the WMS_BUILDINGS_LAYER variable because it is defined in the mySettings.js file as global variable.

You should call *clearVectorLayer* and *reloadBuildingsWmsLayer* after having inserted a geometry, in the Ajax callback function, in the success property.

Remove vector layer content and reload the wms layer

```
function buildingInsert(){
    var f = document.getElementById("form-buildings");
    var formData = new FormData(f)
    var data =Object.fromEntries(formData);
    var js= JSON.stringify(data);

    $.ajax({
        url: URL_DJANGO_API + 'building_insert/', //url where the
            data is sent
        type: "POST", //send method POST or GET
        dataType: 'json', // type of data to send
        data: js, //the data to send
        contentType: 'application/json; charset=UTF-8', // content
            type
        success: function (data){ //callback function. It will wait
            for the server answer
            console.log(data); //logs the server answer in the
                console

            //Shows the server message and the gid of the new
                building on the paragraph p-message
            document.getElementById("p-message").innerHTML = data.
                message + ". gid " + data.data[0].gid;
            f = document.getElementById("form-buildings").reset();
                //clean the form
            clearVectorLayer();
            reloadBuildingsWmsLayer();

        },
        error: function (err){
            console.log(err); //in case of error logs the error in
                the console
        }
    })
}
```


4.6.5 Call the *addDrawPolygonInteraction* function

To add the draw interaction to the map you have to call the function *addDrawPolygonInteraction*, but it is in a js file which is not still loaded. Load the file *js/map/mapDraw.js* in the page, before the *js/map/mapMain.js* file:

```
<script type="text/javascript" src="js/map/mapDraw.js"></script>
```

Now, in the file *js/map/mapMain.js*, after having created the *MAP* object in the function *mapMain*, you can call the function *addDrawPolygonInteraction*, and disable the interaction, so the user can not draw still anything.

You must call the function *addDrawPolygonInteraction* after having created the *MAP* object because that function uses it.

```
function mapMain(){
    ...
    ...
    addDrawPolygonInteraction();
    disableDrawPolygons();
}
```

4.6.6 Add buttons to enable and disable the draw interaction

In the file *index.html* add two new buttons, next the button *building-insert*:

```
<button id="building-insert">Insert</button>
<button id="building-start-drawing">Start drawing</button>
<button id="building-stop-drawing">Stop drawing</button>
```

Connect the click event over the new buttons with the respective functions, to enable-disable the map draw interaction.

```
//file js/main.js

function mainInit(){
    console.log("I am loaded and ready to work");
    document.getElementById("building-insert").addEventListener("
        click", buildingInsert);
    document.getElementById("building-start-drawing").
        addEventListener("click", enableDrawPolygons);
    document.getElementById("building-stop-drawing").
        addEventListener("click", disableDrawPolygons);

    mapMain(); //initializes the map in the div id="map"
}
```

Refresh the page and draw some polygons figure 4.2

After having drawn some polygons, if you reload the page, you will realise that the new polygons have disappeared. This is because the polygons are only in the OpenLayers memory, and when you reload the page, the map is created again, and the *source* of the layer which contains the drawn polygons is created again too. This means that the layer is empty. To keep the drawn polygons you need to send the geometry to the database in the server which is permanent.

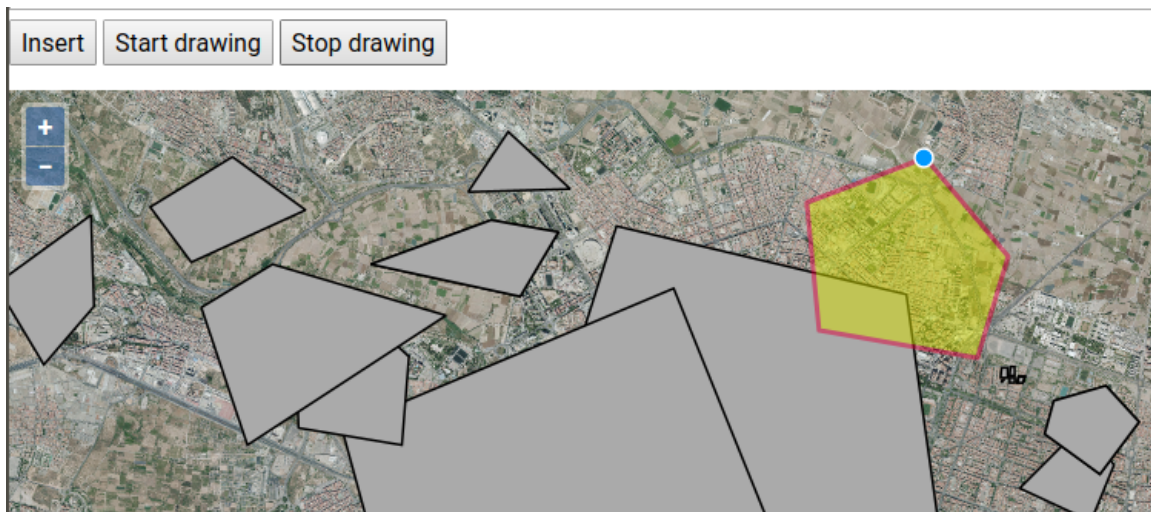


Figura 4.2: Draw polygons with OpenLayers

4.7 Send the drawn polygons in the web page to the database

4.7.1 Link the draw end event of the geometry with a callback function

Modify the function `addDrawPolygonInteraction` in the file `js/map/mapDraw.js`. The `MAP_DRAW_POLYGON.on(draw manageDrawEnd)`; line means that, when a polygon is drawn, the callback function `manageDrawEnd` will be executed. The system pass to the function a parameter `e`, which is an object with a lot of properties, one of which is the geometry of the geometry just drawn.

This must be done only once.

```
\\file js/map/mapDraw.js

function addDrawPolygonInteraction() {
  /*Possible values for tipo_geom:
  *      "Point", "LineString", "Polygon"
  * MAP_DRAW_POLYGON, SOURCE_DRAW and MAP are global variables,
  * defined in
  * mySettings.js, so that are accesible from any function
  * */
  MAP_DRAW_POLYGON = new ol.interaction.Draw({
    source: SOURCE_DRAW, //source of the layer where the
      polygons will be drawn
    type: ('Polygon') //geometry type
  });

  //When a polygon is drawn the callback function manageDrawEnd
  will be executed.
  //The system pass to the function a parameter e, which is an
  objects with
  //a lot of properties, one of which is the geometry of the
  geometry just drawn
  //This must be done only once
  MAP_DRAW_POLYGON.on('drawend', manageDrawEnd);

  //adds the interaction to the map. This must be done only once
```

4.8 Add draw interaction of different geometry types to the map

```
MAP.addInteraction(MAP_DRAW_POLYGON);
}
```

4.7.2 The callback function gets the geometry coordinates and puts them into the form

Now you have to get the coordinates of the geometry just drawn, from the function *manageDrawEnd*. This function will be called each time the user finishes drawing a geometry. The function *manageDrawEnd* will receive an object *e* which contains the geometry coordinates. The function gets the geometry coordinates in WKT format and puts them in the form. To insert the geometry into the database, the user only has to click the on the button *Insert*.

```
\\file js/map/mapDraw.js

/**
 * Function which is executed each time that a polygon is finished
 * of draw
 * Inside the e object is the geometry drawn
 * */
function manageDrawEnd (e) {
  var feature = e.feature;//this is the feature that fired the
  event
  var wktFormat = new ol.format.WKT();//an object to get the WKT
  format of the geometry
  var wktRepresentation = wktFormat.writeGeometry(feature.
  getGeometry());//geometry in wkt
  console.log(wktRepresentation);//logs a message
  var form = document.getElementById("form-buildings");//get the
  buildings form html object
  form.elements["geomWkt"].value=wktRepresentation;//set the
  geometry in wkt format to the geomWkt input
}
```

You could ask for the rest of attributes of the geometries before and send the form data automatically on finishing the draw of the geometry.

4.8 Add draw interaction of different geometry types to the map

In the previous section you have created a function to add, enable and disable the *enableDrawPoints* interaction.

The function *addDrawPolygonInteraction()* should be called only once after the map has been created.

In the same way you can add other interactions to draw other kind of geometries *addDrawPointInteraction*, catching the *drawend* event with other function (*pointDrawEnd*, for example).

The key here is that the interaction can be added to the map once, but once added, they can be activated and deactivated. So when you want to draw a point, you should first deactivate the *MAP_DRAW_POLYGON* interaction, with the function *disableDrawPolygons()*, and the other draw interactions you could have. Later enable the adequate interaction eg. *MAP_DRAW_POINT* with the function *enableDrawPoints*.

You should copy those functions, create the adequate global variables, change the geometry type of the draw interactions, and have a function to deactivate all the interactions (*deactivateAllDrawInteractions()*). Before activating one interaction call to the *deactivateAllDrawInteractions* function.

All the time is the same. The difficult is the organization.

4.9 Exercise 3.

4.9.1 Part 1. Test. Value 2 points

In this text you can be asked about anything explained in this course. Also you can be asked to develop some small function.

4.9.2 Geoportal project requirements and evaluation.

1 points for the following functionalities:

- **It has to work in production mode. The Django API must be served by Apache, not by the development server.**
- A map, done with OpenLayers with the following elements:
 - At least one WMS layer. The layers have to be created by the student with PostGIS and be published by the GeoServer installed in the developing virtual machine.
 - Base layers like ortophotos, OSM, etc.
 - Cursor coordinates indicator and a layer control.
 - A layer switcher control

1 points for the following functionalities

- To be able to modify the database you must be logged in.
- You have to be able to draw new elements and to add them to the database, sending the corresponding form. You have to have a button to start the drawing interaction, to draw a geometry, and on finishing the draw of the geometry, to catch the geometry coordinates in WKT and put them into the form. Complete the form data and send the form to the server in order to insert the complete record: geometry and the rest of the field values.

Web developing implies self learning. To get the remaining 2 points of the subject and avoid to take the exam you can:

- Currently all the forms and the map are seen together. You can make a simple menu to see only one thing at the same time. For example, if you have a map with two layers, two forms to insert geometries into the layers (f1 and f2), and one extra form to insert in a table without geometries (f3). You should (0.5 pts):
 - Show a menu with the following options: Map, f3, Help, About and Login.
 - The Map option shows the map and two buttons, to draw in the layers. On clicking in the button to draw in the layer 1, you should activate the OpenLayers draw interaction, and show the form associated with that layer (f1). The same with the second button. To create a menu and show-hide parts of the web see the section 3.10, 95.
 - The f3 option and Login option, only shows the forms f3 and login respectively.
 - The help option shows an invented dummy content. I will valorize the inclusion of images, centred and with tooltip.
 - Once the user is logged in try to show in the menu bar the name of the user.

- Implement the operations select, update and delete of the elements of at least one layer in the geoportal (0.5 pts). The select operation must put the field values obtained from the database in the html form. Optionally you can get the cursor coordinates, send them to the backend, get the element that intersects, or is inside or nearer of that coordinates, and send its data to the front end, who will put the field values into the html form.
- Symbolize at least one WMS layer with GeoServer and SLD. You have to label the elements and change its color. You will find examples in <https://docs.geoserver.org/stable/en/user/styling/sld/cookbook/>. (0.25 pts)
- Your Django API must reject geometries that are too near, too far, inside, outside or intersects with the geometries of the editing layer, or the ones of the other layers. You will find an example in the section 2.15, page 77. (0.25 points)

Bibliografía

- [1] Jose Carlos Martínez Llario, *PostGIS 2. Análisis Espacial Avanzado*, 2012.
 - [2] Adrian Holovaty, Jakob Kaplan-Moss, *La guía definitiva de Django*, 2009.
 - [3] Joseph W. Lowerly, Mark Fletcher, *HTML 5 para desarrolladores*, 2011.
 - [4] Juan Diego Gauchat, *El gran libro de HTML5, CSS y JavaScript*, 2012
- Páginas web.**
- [5] Página de referencia HTML5: <http://www.w3schools.com/html>
 - [6] Página de referencia de CSS3: <http://www.w3schools.com/css/default.asp>
 - [7] Tutorial: <http://www.tutosytips.com/aprende-html5-desde-0>
 - [8] Referencia HTML 5 para desarrolladores. <http://www.html-5.com>
 - [9] Compatibilidad de HTML 5 con dispositivos móviles: <http://mobilehtml5.org>
 - [10] Tipos de controles input: http://www.w3schools.com/html/html_form_input_types.asp
 - [11] Control canvas <http://www.html5canvastutorials.com/advanced/html5-canvas-mouse-coordinates/>
 - [12] Open Geospatial Consortium, OGC. www.opengeospatial.org
 - [13] PostgreSQL. <http://www.postgresql.org>
 - [14] PostGis. <http://postgis.refractory.net>. Extensión que da soporte de datos espaciales a la base de datos
 - [15] Descripción del lenguaje SQL <http://www.postgresql.org/docs/9.1/static/tutorialsql.html>
 - [16] Descripción del lenguaje PL/SQL <http://www.postgresql.org/docs/9.1/static/plpgsql.html>
 - [17] Quantum Gis, Qgis. <http://www.qgis.org>
 - [18] Python. <http://www.python.org>
 - [19] PyQt4. <http://www.riverbankcomputing.co.uk/software/pyqt/intro>
 - [20] PIL. <http://www.pythonware.com/products/pil>. Biblioteca Python el trabajo con imágenes
 - [21] psycopg2. <http://pypi.python.org/pypi/psycopg2>. Biblioteca Python para la conexión con PostgreSQL
 - [22] Eclipse. <http://www.eclipse.org>. Entorno de desarrollo (IDE)
 - [23] PyDev <http://pydev.org>. Plugin para Eclipse para el desarrollo en el lenguaje Python