Crafting Web-GIS Applications from Scratch

From Data to Dynamic Maps and APIs using Python

AGU Fall Meeting Scientific Workshop : **https://bit.ly/4aQ4d4l**

Paul Célicourt, PhD.

Assistant Professor at the Centre Eau Terre Environnement, Institut national de la recherche scientifique

December 15, 2025

INRS Institut national de la recherche scientifique

AGU25 New Orleans, LA | 15–19 December 2025

Intro to the workshop

1 Context of the workshop

45-hour graduate-level Geoinformatics course taught at INRS (Quebec, CA) designed to build end-to-end Web-GIS app. Course materials trimmed for a 1-day presentation at a Summer school on agricultural data management and processing in Quebec.

2 Existing tools for geospatial web applications

Ex: Esri/QGIS web tools, Dashboard frameworks (Streamlit/Dash), Cloud geo platforms (GEE/Mapbox/CARTO)

3 Objective of the workshop

Build a working Web-GIS app (data, standard database, map, API) in 1 day

4 Requirements

Personal computers, **Github account**, **Abstract API account**

Rationale for the Workshop: As scientists, we are increasingly committed to FAIR data. Therefore, we need full control over the backend of our data systems and the database in particular. Many “easy/fast” Web-GIS tools come with opaque schemas, file-based database and closed/rigid workflows. Therewith, it is hard to truly meet FAIR principles.

Workshop Agenda

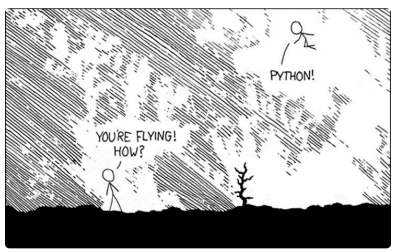
1. Part I: Introduction to Python and the Django library (1 hr)
2. Part II: Backend components of a Django-based Web-GIS application (2h)
3. Part III: Frontend components of a Django-based Web-GIS application (1 hr)
4. Part IV: Web Services and Django-based REST Web Services (1 hr)
5. Part V: Interactive frontend development techniques & backend integration (wrap-up; 1.5 hr)

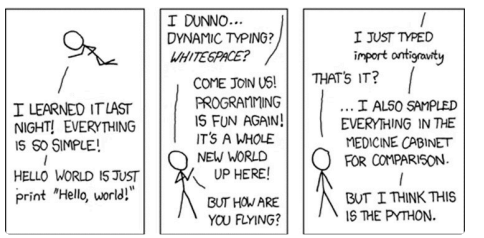
Part I: Quick Intro to Python and the Django Library

Quick introduction to Python programming

Python: A high-level, interpreted programming language developed by Guido Van Rossum, currently Distinguished Engineer at Microsoft. It features a simple syntax, making it easily readable and extremely user-friendly for beginners.

Fun Fact: Try the instruction `import antigravity` in a local Python installation and see the immediate redirection to **https://xkcd.com/353/**!





The GitHub Blog

Why Python keeps growing, explained

A deep dive into why more people are using Python than ever, its key use cases, and why it's still so popular 30-plus years after it was first released.

Quick introduction to Python programming

Variables are containers used to store one or more values, i.e., text, numbers, etc. The purpose of using variables is to allow for the later use of stored values by referring to the variable name.

Creating a variable: you must choose a name for the variable and then assign a value to that name. E.g.:

```
latitude = 46.8127781
longitude = -71.2243133
```

Rules for choosing a variable name:

1. The name must begin with a **letter** (uppercase or lowercase) or an **underscore**;
2. A variable name must only contain common alphanumeric characters;
3. You cannot use Python's **reserved keywords**!
4. The name must be **sufficiently descriptive** without being too long.

Try JupyterLite to execute Python codes in the browser: https://jupyter.org/try-jupyter/lab/ and define two variables of your choice.

Quick introduction to Python programming

Data are raw **values** (number, text, and symbol) uninterpreted, collected through observations or measurements. Ex: satellite imagery, aerial photographs, digital elevation models.

- **Data Types**: a category or class of data. We distinguish 5 main types:
  1. **floats** (decimal numbers; `float`): `latitude = 46.8127781`
  2. **integers** (whole numbers; `int`): `number_of_workshop_participants = 50`
  3. **booleans** (true/false; `bool`): `workshop_is_cancelled = False`
  4. **strings** (text enclosed in quotes; `str`): `first_name = 'Joe'`

- **Data Structures**: a way to organize and store data to perform operations efficiently. We distinguish the following types:
  1. **List**: a collection of **comma-separated elements** and **enclosed in square brackets** `[]`. These types are called mutable. `lab_location = [46.8127781, -71.2243133]`
  2. **Tuple**: a collection of **comma-separated elements** and **enclosed in parentheses**. They cannot be modified after their creation (immutability). These types are called immutable. `lab_location = (46.8127781, -71.2243133)`
  3. **Set**: a mutable but unordered collection of **comma-separated elements** and **enclosed in curly braces**, containing **no duplicate elements**. Example: `agu_meeting_locations = {'Washington, DC', 'New Orleans', 'Chicago', 'San Francisco'}`.
  4. **Dictionary**: a set of **key:value pairs** separated by commas and **enclosed in curly braces**. Each key must be unique within a dictionary. Example: `agu_meeting_locations = {2024 : 'Washington, DC', 2025 : 'New Orleans', 2022: 'Chicago', 2023 : 'San Francisco'}`

> Go to **https://jupyter.org/try-jupyter/lab** and define **two variables** that combine Python data types and data structures (e.g., a list of strings, a dictionary with numeric values).

Quick introduction to Python programming

Operators: a sign or symbol that allows an operation to be performed.

Arithmetic operators

Operator	Operation	Example
`+`	Addition	`5 + 2 = 7`
`-`	Subtraction	`4 - 2 = 2`
`*`	Multiplication	`2 * 3 = 6`
`/`	Division	`4 / 2 = 2`
`**`	Power	`4 ** 2 = 16`

Assignment operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

Quick introduction to Python programming

Functions: a set of instructions grouped under a name or command that allows a specific task to be performed by calling the function by its name.

- A function is **defined using the keyword** `def` and comprises five elements:
  1. The function's name
  2. The arguments passed to the function (optional)
  3. A text (docstring) explaining the function (optional)
  4. The function's code
  5. The data returned by the function (optional)

- **The syntax is as follows**:

```
def function_name(parameter 1, parameter 2,...):
    instruction block
```

- **Example of functions**:

```
def celsius_to_fahr(celsius):
    fahr = 9 / 5 * celsius + 32
    return fahr

temperature = 25 #degree C
fahr = celsius(temperature)
```

Python comes with a number of native (built-in) **functions** available in Python's standard library (print(), len(), range(), list(), dict(), etc.).

Try `print("Hello, AGU25 FM Participants!")` in JupyterLite.

Good practice: Similar to the variables, the function name should be **sufficiently descriptive** without being too long.

Quick introduction to Python programming

Control Structures: Specific instructions that allow controlling the execution of code. There are **conditional** and **loop** control structures.

Conditional control structures allow a block of code to be executed if a certain condition is met.

The **syntax** is as follows:

```
if condition1:
  bloc_instructions_1
elif condition2:
  bloc_instructions_2
else:
  bloc_instructions_3
```

Concrete example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Exercise: Climatic Zones

Use the example of how to use `if`/`elif`/`else` to determine the climatic zone given a `latitude=45` degree (or any number between 0 and 90) :

1. Hot zone: Latitude between 0 and 30 degrees:
2. Temperate zone: Latitude between 30 and 60 degrees:
3. Cold zone: Latitude between 60 and 90 degrees:

Note that two `if...else` structures can be nested within each other. See here for additional resources:

www.w3schools.com

Python conditional statements

Quick introduction to Python programming

Loop control structures: allow a block of code to be executed in a loop as long as a condition is met. There are two types of loops: **the bounded loop** `for` and **the unbounded loop** `while`.

The `for` loop allows iteration over the elements of a sequence. The program stops or continues after traversing the last element of the sequence.

The syntax is as follows:

```
for element in sequence:
  instructions (do something with element)
```

An example:

```
agu_meeting_locations = ['Washington, DC', 'New Orleans',
'Chicago', 'San Francisco']
for city in agu_meeting_locations :
   print(city)
```

Note that **conditional control structures** can contain **loop control structures** and vice-versa. See here for additional resources:

www.w3schools.com

Python for loop

Quick introduction to Python programming

The `while` loop allows a block of code to be executed as long as a condition is true.

The syntax is as follows:

```
while condition:
 instructions
```

The `break` statement allows stopping the execution of a loop when a certain condition is met.

The `continue` statement allows skipping the current iteration of the loop and moving directly to the next one.

An example:

```
agu_meeting_locations = ['Washington, DC', 'New Orleans', 'Chicago', 'San Francisco']

index_city = 0

while index_city < len(agu_meeting_locations):
    city_name = agu_meeting_locations[index_city]
    print(index_city, city_name)
    if city_name == 'New Orleans':
        break
    index_city += 1
```

www.w3schools.com

Python while loop

General rule: a `for` loop is used **when the number of repetitions is known in advance**, and a `while` loop otherwise.

Quick introduction to Python programming

Programming paradigms (OPTIONAL): approaches or methods for organizing computer programs independently of the language used. There are four main types of paradigms are: **imperative programming**, **functional programming**, **procedural programming**, and **object-oriented programming**.

Functional programming allows building programs using a set of functions that perform a precise task.

```
temperature = 30 #degre C

def celsius_to_fahr(celsius):
    fahr = 9 / 5 * celsius + 32
    return fahr

print (celsius_to_fahr(temperature))
```

Imperative programming defines a program as an ordered sequence of instructions.

```
agu_meeting_locations = ['Washington, DC', 'New Orleans', 'Chicago', 'San Francisco']

index_city = 0

while index_city < len(agu_meeting_locations):

print(index_city, agu_meeting_locations[index_city] )
    if index_city== 3:
        break
    index_city += 1
```

Procedural programming is also imperative programming, with the difference that code is grouped into procedures (routines or functions).

Conversion of the average of two temperature values from Celsius degrees to Fahrenheit degrees using the `convert_avg_celsius_to_fahr` function.

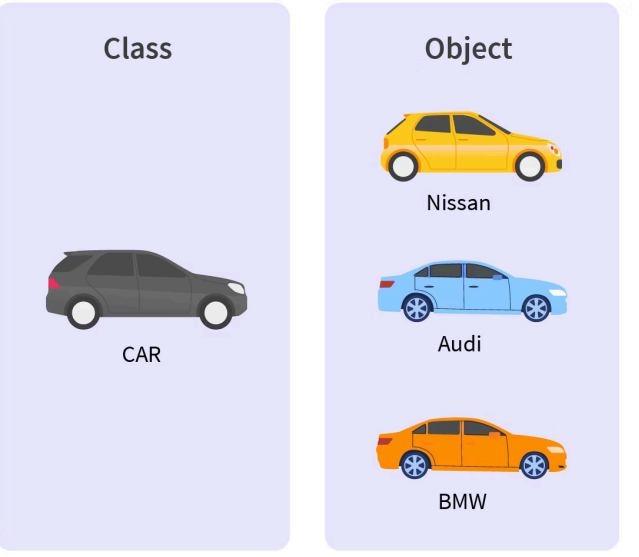
```
import statistics

temperature_room = 30 #degre C
temperature_outdoor = 20 #degre C

def celsius_to_fahr(celsius):
    fahr = 9 / 5 * celsius + 32
    return fahr

def convert_avg_celsius_to_fahr(temp_1, temp_2):
    fahr_1 = celsius_to_fahr(temp_1)
    fahr_2 = celsius_to_fahr(temp_2)
    moyenne_fahr = statistics.mean((fahr_1, fahr_2 ))
    return moyenne_fahr

print(convert_avg_celsius_to_fahr(temperature_room, temperature_outdoor))
```

Quick introduction to Python programming

Object-Oriented Programming (OOP): a programming paradigm that allows code to be organized into objects representing real-world entities or concepts. OOP is based on the concept of a **Class**.

A **Class** represents the **abstract description** (a "concept") **of a set or collection of objects** with **one or more similar characteristics**.

A Class is a model, template, or prototype that defines (a) the **variables** (properties or attributes) and (b) the **methods** (functions) common to the **Objects** described by the class. An **object** is an instance or realization of the **Class**.



Django Models or data schemas are defined as Classes, a reason why OOP matters.

Quick introduction to Python programming

Building a **class:** It is minimally defined in Python with the keyword **class** and the name of the class.

All classes have a special function called `__init__()` executed when an instance of the class is created. It is called the **constructor** and allows to build and customize objects while returning an object containing the attributes passed as parameters.

```
class AGUConferenceCity:
    def __init__(self, name, state, population, location):
        self.city_name = name
        self.state = state
        self.population = population
        self.geographic_location = location

    def get_city_name(self):
        return self.city_name

agu_2025 = AGUConferenceCity('New Orleans', 'Louisiana',  360000, (29.95465, -90.07507))
print(agu_2025.city_name)
print(agu_2025.get_city_name())
```

ⓘ The keyword `self` refers to the current instance of the class. The syntax for defining a method or function remains the same as that for procedural programming, except that the first parameter must always be `self`, and it is possible to use arguments passed to the constructor as attributes.

Quick introduction to Python programming

Inheritance in OOP: **Inheritance in OOP**: a type of relationship between two or more classes that facilitates the transmission or sharing of data from one class to another.

```
class AGUConferenceVenue (AGUConferenceCity):
    def __init__(self, name, city_name, state, adress):
        self.name = name
        self.city_name = name
        self.state = state
        self.adress = adress

agu_2025 = AGUConferenceVenue ('New Orleans Ernest N. Morial Convention Center', 'New Orleans', 'Louisiana',  '900 Convention Center Blvd, New Orleans, LA 70130')
print(agu_2025.city_name)
print(agu_2025.get_city_name())
```

Two ways to import libraries in Python: `import library_name` OR `from library_name import function_name` OR `from library_name import class_name`

Quick introduction to Python programming

Inheritance in OOP: To retain the inheritance of the parent's `__init__()` function, we call the parent class's `__init__()` function using the parent's name in the child class:

```
class AGUConferenceVenue (AGUConferenceCity):
    def __init__(self, name, state, population, location): #semantic issue here
        AGUConferenceCity.__init__(self, name, state, population, location)

agu_2025_venue = AGUConferenceVenue ('New Orleans Ernest N. Morial Convention Center', 'Louisiana',  360000,
(29.9438933, -90.0639814))
print(agu_2025_venue.name)
```

Two ways to import libraries in Python: `import library_name` OR `from library_name import function_name` OR `from library_name import class_name`

Quick introduction to web programming in Python

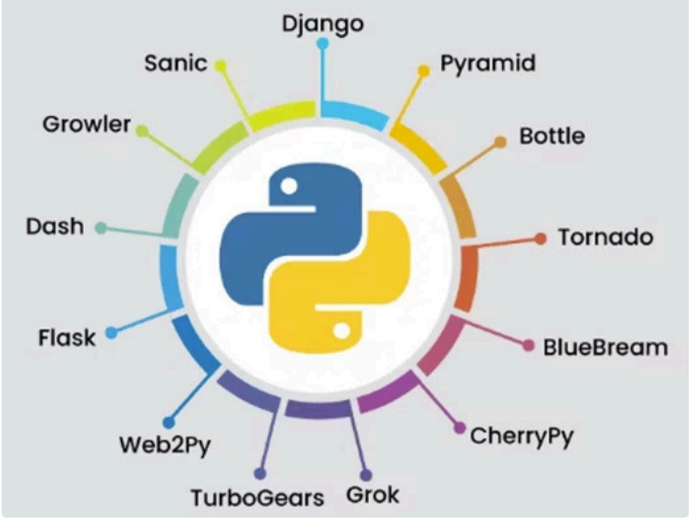
A panoply of frameworks for developing web applications in Python is available. Each has its advantages and disadvantages.

A Web framework is a collection of packages or modules that allow developers to write web applications or web services without having to manage low-level details.

Django, **Pyramid**, **Web2py**, and **TurboGears** are complete frameworks and take into account back-end functionalities, the front-end interface, and the database.

CherryPy, **Bottle**, **Flask**, and **Dash** are micro-frameworks or lightweight frameworks with limited functionalities.

Tornado, **Sanic**, and **Growler** are asynchronous frameworks powered by the **Asyncio** package. They allow developers to manage a massive set of simultaneous processes.



Sources: smcegypt.org & https://www.turing.com/

Takeaway: Django offers a single, open-source framework to handle **data models, database, APIs, and web pages (the full stack)**—exactly what we need to build a complete Web-GIS application from scratch.

Quick introduction to web programming in Python

The **Django web development framework** is designed to accelerate the fast development of complex web applications with minimal effort. There are several ways to install Django in a **virtual environment**:

```
python -m pip install Django
```

Key features: fast development, secure and scalable web applications.

The framework's strength lies not only in its vast collection of packages and the **Object-Relational Mapper** functionality for database connectivity, but also in its software architecture pattern which promotes component reusability.

> It was created in 2003 by and for a news agency, Lawrence Journal-World, in the state of Kansas in the United States. The framework was made available to the public two years later. Several public websites, including Pinterest, Instagram, and Mozilla are now based on `Django`.

Quick introduction to web programming in Python

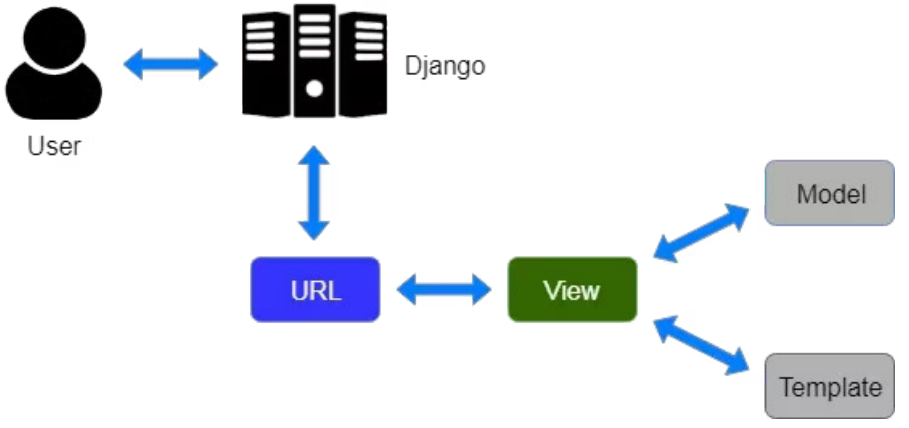
`Django` uses the *software design pattern* named `Model-View-Template` which enables applications to be separated into three interconnected logical elements: Model, View, and Controller.

The **Model** is responsible for data representation and database interactions;

The **View** is responsible for web requests processing and the return of the appropriate response, often through the **Template** element after querying the **Model**.

The **Template** is made up of HTML files containing front-end logics that define the data presentation to the user.

The Django framework architecture



Django request execution diagram (Source: **hSamuel Getachew**)

> Django provides a Template engine **(Django Template Language)** capable of displaying variables, and executing conditional structures (*if/else/elif*) or loops (*for*) within the HTML codes.

Quick introduction to web programming in Python

How do we create the MVT components? `django-admin startproject <projectname>` and `django-admin startapp <appname>`.

When running the command `django-admin startproject <projectname>`, at the root directory level of the Django project (<projectname>), several essential files and directories are generated as follows :

```
#Source https://getcyber.me/posts/anatomy-of-a-django-project-a-comprehensive-guide-to-files-and-structure/
<projectname>/      <-- Outer container directory (name doesn't matter to Django)
 ├──── manage.py       <-- Command-line utility that acts as the primary interface for interacting with your Django project
 └──── <projectname>/  <-- Inner project directory (Python package for your project)
    ├──── __init__.py
    ├──── settings.py   <-- Project settings/configuration that we will edit during the workshop
    ├──── urls.py       <-- Project-level URL declarations that we will edit during the workshop
    ├──── asgi.py       <-- ASGI entry point for async features
    └──── wsgi.py       <-- WSGI entry point for traditional deployment
```

Quick introduction to web programming in Python

When running the command `django-admin startapp <appname>`, at the root directory level of the Django project (<appname>), several essential files and directories are generated as follows :

```
#Source https://getcyber.me/posts/anatomy-of-a-django-project-a-comprehensive-guide-to-files-and-structure/
<appname>/
├── __init__.py
├── admin.py       <-- Admin site configuration for this app
├── apps.py        <-- App configuration
├── migrations/    <-- Database migration files that we will add/edit during the workshop
│   └── __init__.py
├── models.py      <-- Database models (data structure) that we will edit during the workshop
├── tests.py       <-- Automated tests for this app
└── views.py       <-- Request handling logic (views) that we will edit during the workshop
```

static files (css, javascript, html, images) live in the appname folder and templates are part of the static folder. The file structure is **<appname>/static/templates/**.

Django Project

How to manage static files (e.g. images, JavaScript, CSS) | Django d...

The web framework for perfectionists with deadlines.

Codespaces Demo: Setting up a Django WebGIS Development Environment

Demo: Setting up a Django WebGIS Development Environment

To complete this practical exercise, you will need to **follow the instructions in the README file** of the dedicated github repository (left colum of the table below). If run successfully, the end result should look like the image in the right column of the table.

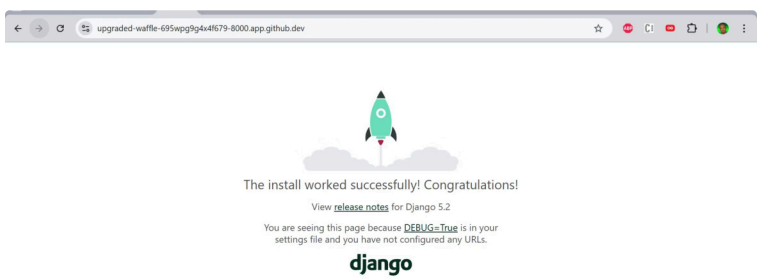
licourt/
fm2025

0 Issues 0 Stars 0 Forks

GitHub

agufm2025/README.md at main · pcelic...

Contribute to pcelicourt/agufm2025 development by creating an account on GitHub.



An alternative is to use a Django-specific development template : **https://github.com/github/codespaces-django**

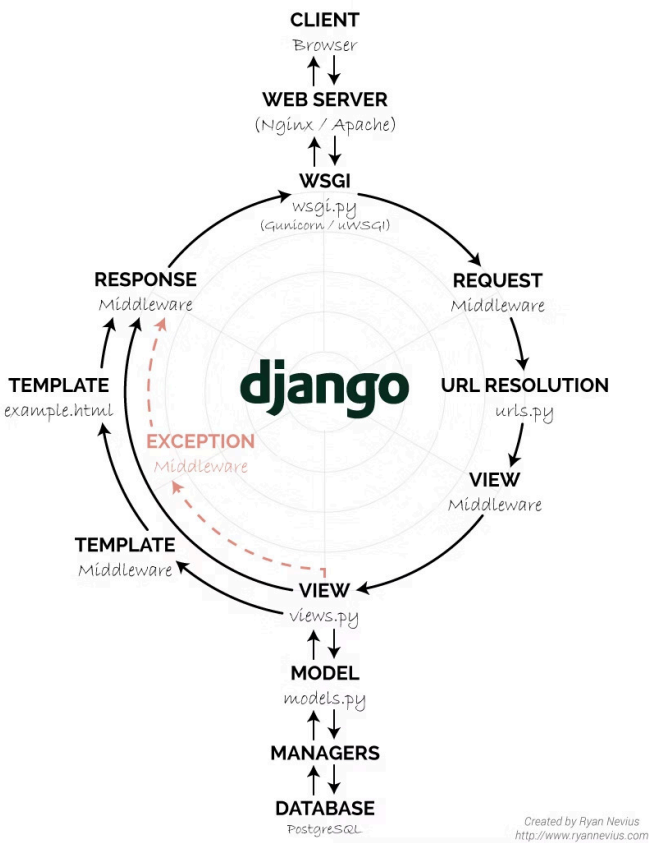
Part II: Back-end of a Django-based Web Application

Back-end of a Django-based Web application

All the files generated with `django-admin startproject` and `python manage.py startapp` are at the fundamental level part of the Django back-end or the Django server.

The TEMPLATE files, although they are HTML files, are rather blueprints for the content displayed in the browser to the CLIENT.

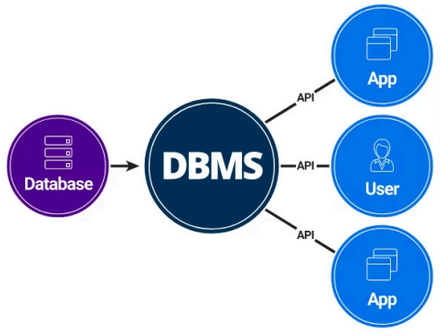
The back-end of a Django-based Web Application comprises, according to the opposite Request-Response Cycle, all elements starting from the WEB SERVER to the DATABASE.



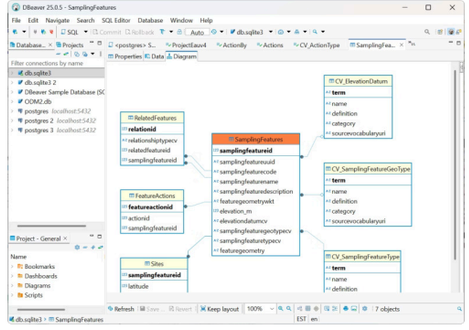
Back-end of a Django-based Web application

Database: an organized collection of structured, semi-structured, or raw data linked together coherently to be easily queried, managed, and updated.

Database Management System (DBMS): a set of specialized software that allows for the storage, retrieval, and manipulation of data.



Relationship between DB and DBMS (Source: SWATHI RAMYA)



Examples of DBMS Interface (Source: https://nsinfo.yo.fr)

DBMS (Source: https://fr.getapp.ca/software/122219/postgresql)

DBMS acts as an **interface between users/applications AND databases**, enabling the execution of operations of interest (e.g., CRUD: Create, Retrieve, Update, Delete) on a database. We will SQLite DB in this workshop.

Back-end of a Django-based Web application

Some roles of a DBMS:

Data integrity preservation

- Avoid data inconsistency during updates;
- Avoid data redundancy;

Data access management

- Ensure easy and fast storage and access to data;
- Provide flexible methods for accessing each data element;

Data security

- Prevent unauthorized access;
- Provide different access levels;
- Coordinate conflicting modification operations.

Back-end of a Django-based Web application

Data modeling: a process that consists of creating an **abstract and structured representation of data** used by an organization. It involves structuring and organizing data in a logical and consistent manner to effectively meet the needs of an application.

Data modeling aims to improve query performance, facilitate database maintenance, and ensure data integrity (accuracy, completeness, and reliability). It begins with the development of a **conceptual model**, which evolves into a **logical model**, then into a **physical model**.

1

Conceptual Data Models: Definition of the structures and concepts of a project or domain.

2

Logical Data Models: Definition of logical entities (data types or classes), the data attributes that define these entities, and the relationships between them.

3

Physical Data Models: Design of the internal schema of a database (tables, table columns, and relationships between them).

Data modeling uses standardized schemas and formal techniques such as the visual modeling language Unified Modeling Language (UML) or a data manipulation language (SQL).

Back-end of a Django-based Web application

Some types of data models:

- The **relational model** (SQL) which organizes data in table form;
- The **non-relational model** (noSQL) which organizes data in a format optimized according to the requirements of the type of data stored (JSON);
- The hierarchical model where data is classified according to a descending tree structure (e.g., XML);
- The document-oriented data model arranges data in tabular form in text files (e.g., csv);
- The object-oriented data model allows managing data structures with complex relationships.

The relational model will be studied in this course.

Back-end of a Django-based Web application

Fundamental concepts in data modeling:

Entity: **person**, **place**, and **object** that can or must be described in a data model. An entity is represented by a table in relational databases or a Class in Django-based data models.

Regular entity: its existence does not depend on the existence of another entity.

Weak entity: its existence depends on the existence of another entity.

Examples of entities: **Land, Building, Occupant.**

The `Building` entity is a `weak entity`; therefore, it only exists if the corresponding `Plot` (`regular entity`) is present in the database.

Back-end of a Django-based Web application

Fundamental concepts in data modeling:

Attribute: characteristics or properties of entities. In a data model, an attribute can be **mandatory** or **optional** and can have a domain that defines it values.

Parcel
ID
Owner
Address

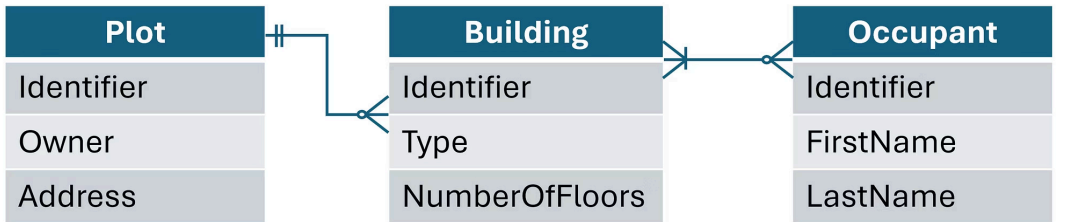
Building
ID
Type
Number of Floors

Occupant
ID
Name
Contact

Back-end of a Django-based Web application

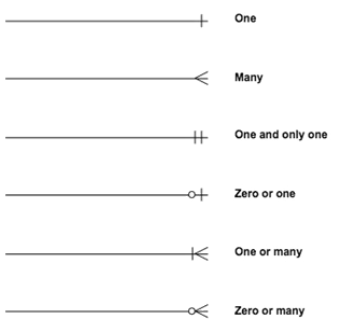
Fundamental concepts in data modeling:

Relationships: they represent the links or associations between entities. The **entity-relationship diagram** shows how entities are related to each other, i.e., their **cardinality** and **ordinality** using a conventional notation such as the Crow's foot notation.



Cardinality represents the maximum number of participations allowed for an entity in a relationship with another entity. It is described using a notation system called crow's foot notation.

Ordinality represents the minimum number of participations allowed for an entity in a relationship with another entity, i.e., whether the relationship is **optional** or **mandatory**.

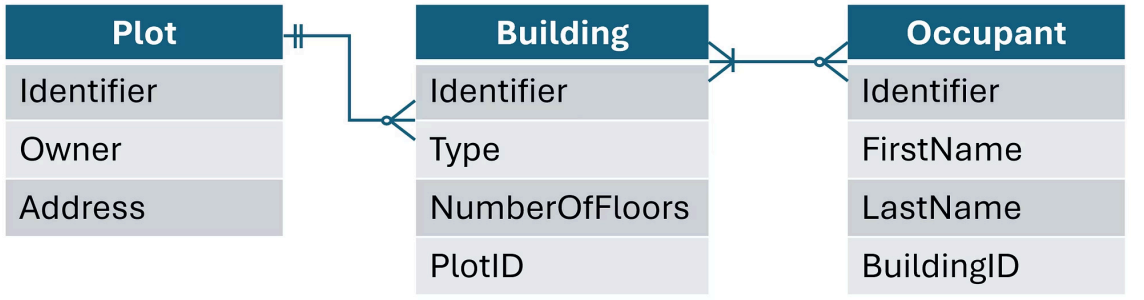


Back-end of a Django-based Web application

Logical data modeling is a more detailed representation of data, derived from **conceptual modeling**. It follows these rules:

- **Rule 1**: Each entity from the conceptual model becomes a table in the logical model.
- **Rule 2**: Each identifier of an entity from the conceptual model becomes a `primary key` of the corresponding table in the logical model. Other properties become the attributes of the table.
- **Rule 3**: The values of the cardinalities of each association are defined by (a) adding a `foreign key` to an existing table or (b) creating a new table whose `primary key` is obtained by concatenating foreign keys corresponding to the linked entities.

The `primary key` uniquely identifies a record in a table (the *Identifier* attribute); the `candidate key` is a minimum set of attributes that can act as the primary key; the `foreign key` creates a link with the primary key of another table (the *PlotId* and *BuildingId* attributes).



Back-end of a Django-based Web application

Integrity constraints define rules to ensure data consistency. Several types exist:

- The **domain constraint** defines the set of possible values for an attribute.
- The **key constraint (primary and foreign)** defines a minimum set of columns such that the table cannot contain two records with the same values for these columns.
- The **mandatory constraint** or **non-null constraint** specifies that one or more attributes must always have a value.
- The **referential integrity constraint** requires the linking of two columns or two sets of columns from two different tables. This constraint is maintained by the use of a foreign key.
- The **uniqueness constraint** ensures that another column could replace the primary key column.
- The **check constraint** indicates that the values in a column must satisfy a condition.

Keywords (e.g., `Not NULL, NULL, CHECK, CONSTRAINT, UNIQUE`) are used to define these constraints in DBMS. In Django Models, these constraints are expressed as parameters (`null=False, blank=False, unique=True`) to the class (CharField, FloatField) that defines the attribute

Back-end of a Django-based Web application

Normalization (OPTIONAL) is an indicator of the quality of a table with regard to the problem of data redundancy or repetition. The more normalized a table is, the fewer risks of semantic inconsistencies there are in relational schemas. There are three normal forms characterizing relational tables:

First Normal Form (1NF): a relation is in first normal form if **every attribute is atomic**, i.e., non-decomposable (e.g., the postal codes in the figure opposite are not atomic).

Second Normal Form (2NF): a relation is in second normal form if and only if (a) it is in first normal form and if each non-key attribute depends fully and not partially on the primary key.

Out[21]:

	City	city_ascii	province_id	province_name	lat	lng	population	density	timezone	ranking	postal	id
0	Toronto	Toronto	ON	Ontario	43.7417	-79.3733	5647656.0	4427.8	America/ Toronto	1	M5T M5V M5P M5S M5R M5E M5G M5A M5C M5B M5M M5...	1124279679
1	Montréal	Montreal	QC	Quebec	45.5089	-73.5617	3675219.0	4833.5	America/ Toronto	1	H1X H1Y H1Z H1P H1R H1S H1T H1V H1W H1H H1J H1...	1124586170
2	Vancouver	Vancouver	BC	British Columbia	49.2500	-123.1000	2426160.0	5749.9	America/ Vancouver	1	V6Z V6S V6R V6P V6N V6M V6L V6K V6J V6H V6G V6...	1124825478
3	Calgary	Calgary	AB	Alberta	51.0500	-114.0667	1306784.0	1592.4	America/ Edmonton	1	T1Y T2H T2K T2J T2M T2L T2N T2A T2C T2B T2E T2...	1124690423
4	Edmonton	Edmonton	AB	Alberta	53.5344	-113.4903	1151635.0	1320.4	America/ Edmonton	1	T5X T5Y T5Z T5P T5R T5S T5T T5V T5W T5H T5J T5...	1124290735

> The disaggregation of the `postal` column will lead to redundancy of the primary key `id`. The non-key attribute (postal) will not depend entirely on the primary key, hence the non-compliance with the second normal form. Another table must be created (e.g., postal with a foreign key id_city) to bring the city table to the Second Normal Form (2NF).

Third Normal Form: the relation is in first and second normal forms and there is no functional dependency between non-key columns.

Back-end of a Django-based Web application

The Model component: Django implements the **Object-Relational Mapping** (ORM) functionality, which is a programming technique used to bridge a **database** and **OOP** languages.

The ORM provides a powerful way to interact with a database using Python code without writing raw SQL code.

To do this, database tables are mapped to **classes (OOP)** defined in Python, and database records are mapped to instances of these classes. These classes are called models and define the structure and behavior of the data.

The classes in the `models.py` module are created by inheriting Django's '**django.db.models.Model**' class or '**django.contrib.gis.db.models.Model**' class in the case of geoweb applications based on **GeoDjango**. With the classes in place, Django, using commands like `makemigrations`, and `migrate` can automatically generate the database schema (including tables, attributes, and relationships) and populate the DB.

> **GeoDjango** is a contrib module (inactive by default) included in Django that transforms it into a geoweb framework.

Back-end of a Django-based Web application

GeoDjango provides (a) the necessary data types to the Django model for defining *Geometry* types of the **OGC Simple Features Access standard** and raster data, (b) extensions to Django's ORM for querying and manipulating spatial databases, and (c) high-level Python interfaces for spatial analysis of geometric and raster types, as well as data manipulation in different formats.

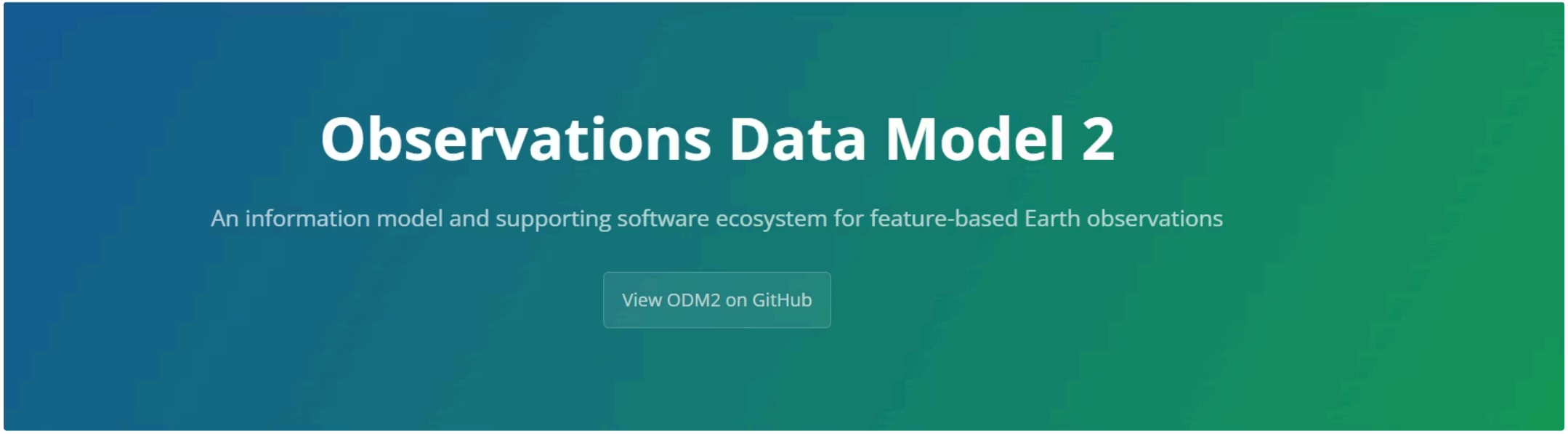
```python
from django.contrib.gis.db import models


class WorldBorder(models.Model):
    # Regular Django fields corresponding to the attributes in the
    # world borders shapefile.
    name = models.CharField(max_length=50)
    area = models.IntegerField()
    pop2005 = models.IntegerField("Population 2005")
    fips = models.CharField("FIPS Code", max_length=2, null=True)
    iso2 = models.CharField("2 Digit ISO", max_length=2)
    iso3 = models.CharField("3 Digit ISO", max_length=3)
    un = models.IntegerField("United Nations Code")
    region = models.IntegerField("Region Code")
    subregion = models.IntegerField("Sub-Region Code")
    lon = models.FloatField()
    lat = models.FloatField()

    # GeoDjango-specific: a geometry field (MultiPolygonField)
    mpoly = models.MultiPolygonField()

    # Returns the string representation of the model.
    def __str__(self):
        return self.name
```

An example of Geospatial model definition in Django.

- GeoDjango Model API
  - Spatial Field Types
    - GeometryField
    - PointField
    - LineStringField
    - PolygonField
    - MultiPointField
    - MultiLineStringField
    - MultiPolygonField
    - GeometryCollectionField
    - RasterField

The Geometry fields (data types) defined in the GeoDjango Model API.

Back-end of a Django-based Web application

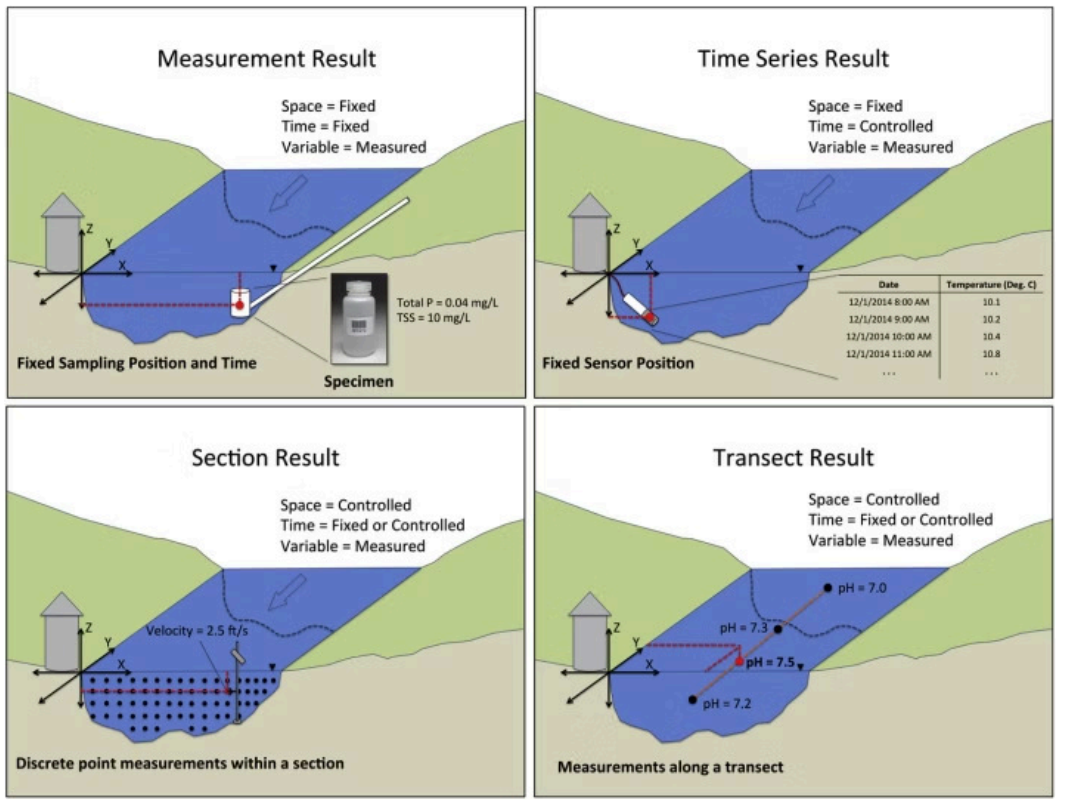


The Observations Data Model

ODM2 is an information model and supporting software ecosystem for entity-based environmental observations, designed to facilitate interoperability between scientific disciplines and domain cyberinfrastructures.

An ODM2 database is capable of integrating various types of data from data acquisition processes, as shown in the adjacent figure. This includes, for example, hydrological time series, soil and sediment geochemistry, biodiversity surveys, oceanographic sensor profiles, and more.

ODM2 was developed with the help of the hydroinformatics community and is a profile of the OGC (Open Geospatial Consortium) Observations and Measurements (O&M) standard.

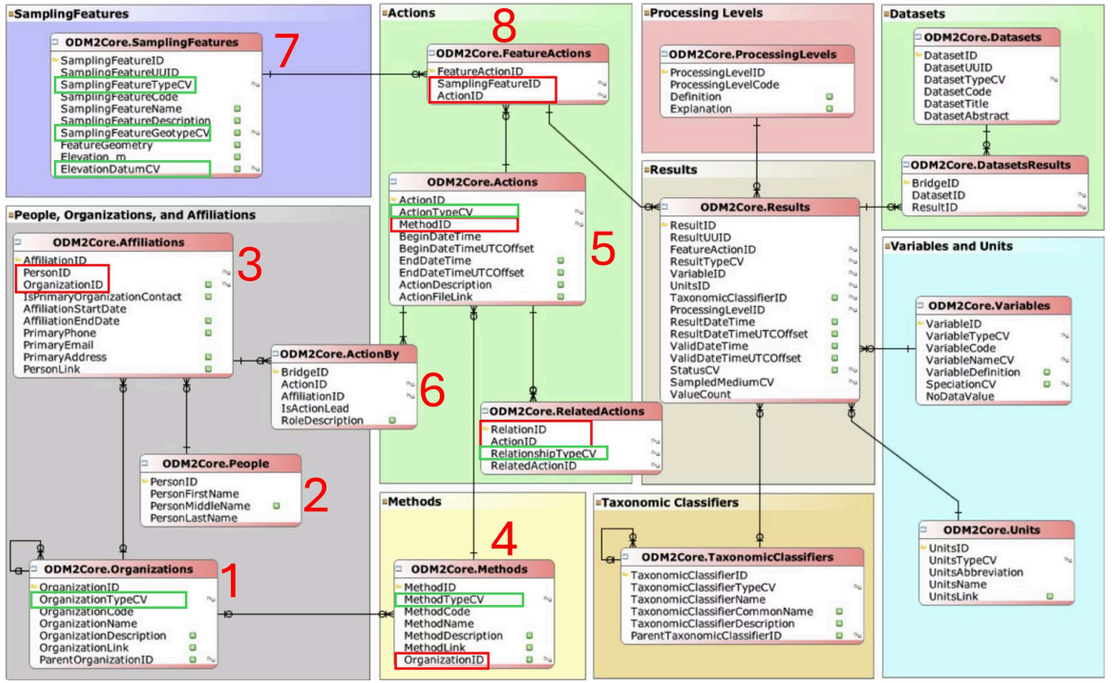


The Observations Data Model

To unambiguously describe observations, ODM2 integrates a rich set of descriptive metadata about the observed variables and the context in which an observation was made to support the discovery and interpretation of observational data. The core of ODM2, ODM2Core, includes the following tables:

Within the ODM2 core, an "Observation" consists of two elements: an **Action** (Table **Actions)** performed on or within a **SamplingFeature** (Table **SamplingFeatures)** which produces an observation result (Table **Results**), which is the culmination of that action.

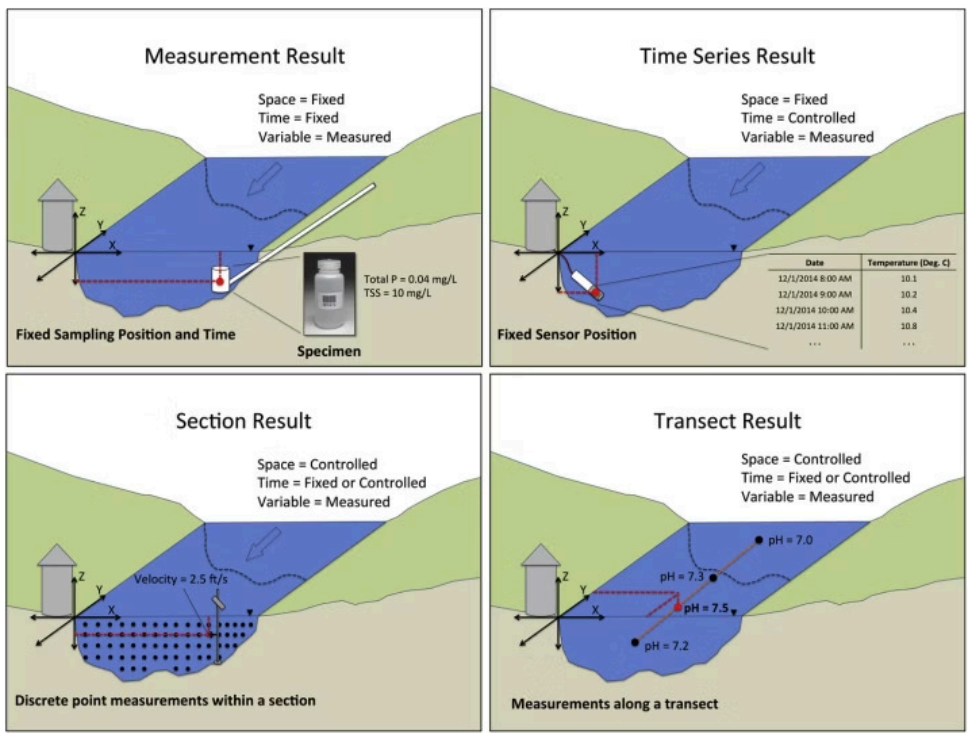
ODM2 explicitly models **Actions** and the individuals or teams (**People**) who perform them, as well as their affiliations. Separating **Actions** from their **Results** allows, for example, a single action to produce numerous results, and 2) to differentiate between actions.

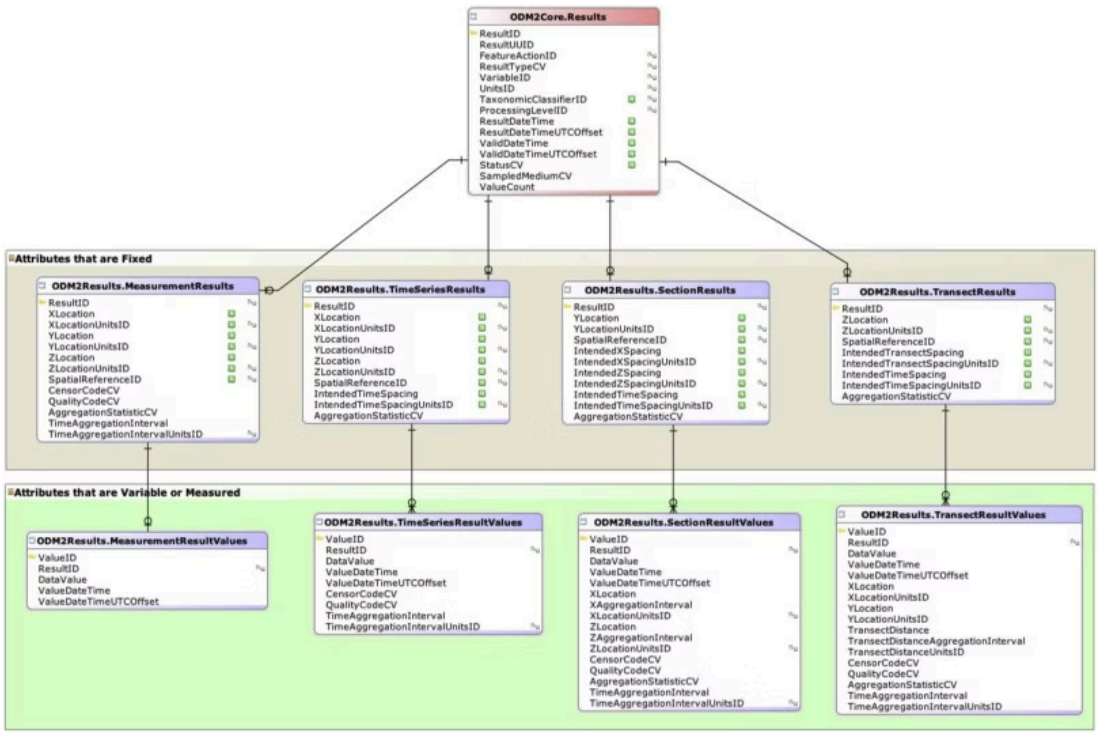


Recommended reading: Horsburgh, J. S., Aufdenkampe, A. K., Mayorga, E., Lehnert, K. A., Hsu, L., Song, L., ... & Whitenack, T. (2016). Observations Data Model 2: A community information model for spatially discrete Earth observations. *Environmental Modelling & Software*, *79*, 55-74.

The Observations Data Model

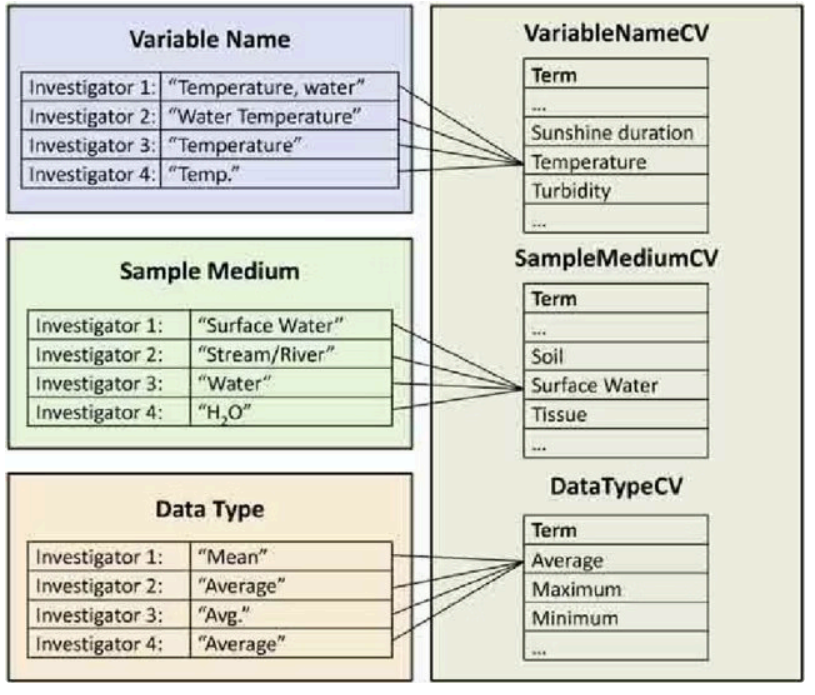
In ODM2, each Result type has a spatial component, a temporal component, and an observed variable component.





The Observations Data Model

ODM2 is also supported by a set of **controlled vocabularies** that promote semantic consistency in the language used to describe observations. In information science, a **controlled vocabulary** is a list of carefully chosen words and phrases following a consensus among parties and used to label documents and data, in order to retrieve them more easily during a search.



Examples of controlled vocabularies:

vocabulary.odm2.org

ODM2: Controlled Vocabularies

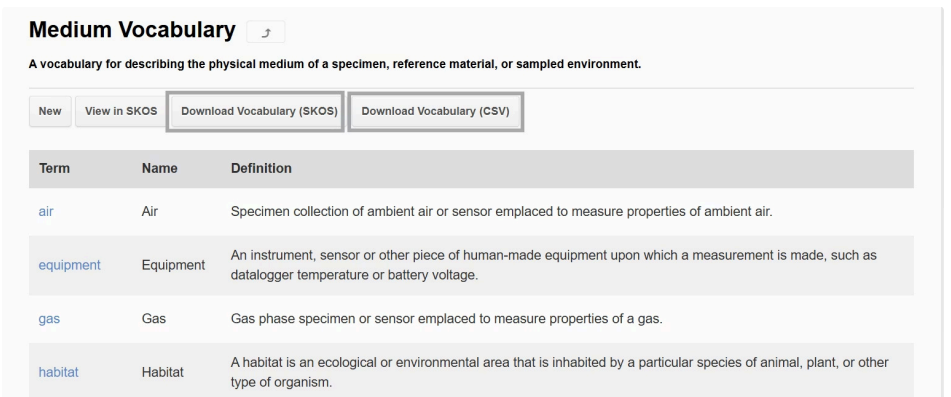
Version 2 of the Observations Data Model (ODM2) has several controlled vocabularies. This web page was developed to promote consistency between...

AGROVOC

AGROVOC Multilingual Thesaurus – AGROVOC

The Observations Data Model

The Controlled Vocabularies of ODM2 are defined as tables, can be downloaded in csv format for insertion into the database. Below is an example of ODM2 controlled vocabulary definition and its Django model definition.



Medium Vocabulary

A vocabulary for describing the physical medium of a specimen, reference material, or sampled environment.

New | View in SKOS | Download Vocabulary (SKOS) | Download Vocabulary (CSV)

Term	Name	Definition
air	Air	Specimen collection of ambient air or sensor emplaced to measure properties of ambient air.
equipment	Equipment	An instrument, sensor or other piece of human-made equipment upon which a measurement is made, such as datalogger temperature or battery voltage.
gas	Gas	Gas phase specimen or sensor emplaced to measure properties of a gas.
habitat	Habitat	A habitat is an ecological or environmental area that is inhabited by a particular species of animal, plant, or other type of organism.

```python
from django.contrib.gis.db import models

class CV_Medium(models.Model):
    term = models.CharField(max_length=255)
    name = models.CharField(primary_key=True, max_length=255)
    definition = models.CharField(max_length=5000, blank=True, null=True)
    category = models.CharField(max_length=255, blank=True, null=True)
    sourcevocabularyuri = models.CharField(max_length=255, blank=True, null=True)

    class Meta:
        managed = True
        db_table = 'CV_Medium'
```

The Observations Data Model

Examples of defining ODM2 tables using the Django/GeoDjango library.

```
class SamplingFeatures(models.Model):
    samplingfeatureid = models.AutoField(primary_key=True)
    samplingfeatureuuid = models.UUIDField()
    samplingfeaturetypecv = models.ForeignKey(CV_SamplingFeatureType, models.DO_NOTHING,
                                              db_column='samplingfeaturetypecv')
    samplingfeaturecode = models.CharField(unique=True, max_length=255)
    samplingfeaturename = models.CharField(max_length=255, blank=True, null=True)
    samplingfeaturedescription = models.CharField(max_length=5000, blank=True, null=True)
    samplingfeaturegeotypecv = models.ForeignKey(CV_SamplingFeatureGeoType, models.DO_NOTHING,
                                                 db_column='samplingfeaturegeotypecv', blank=True, null=True)
    featuregeometry = models.GeometryField(srid=3857, blank=True, null=True)
    featuregeometrywkt = models.CharField(max_length=10000000, blank=True, null=True)
    elevation_m = models.FloatField(blank=True, null=True)
    elevationdatumcv = models.ForeignKey(CV_ElevationDatum, models.DO_NOTHING,
                                         db_column='elevationdatumcv', blank=True, null=True)

    class Meta:
        db_table = "SamplingFeatures"
        managed = True


class RelatedFeatures(models.Model):
    relationid = models.AutoField(primary_key=True)
    samplingfeatureid = models.ForeignKey(SamplingFeatures, models.DO_NOTHING, db_column='samplingfeatureid')
    relationshiptypecv = models.ForeignKey(CV_RelationshipType, models.DO_NOTHING, db_column='relationshiptypecv')
    relatedfeatureid = models.ForeignKey(SamplingFeatures, models.DO_NOTHING, db_column='relatedfeatureid',
                                         related_name = 'relatedsamplingfeatures')
    #spatialoffsetid = models.ForeignKey(SpatialOffsets, models.DO_NOTHING, db_column='spatialoffsetid', blank=True, null=True)

    class Meta:
        managed = True
        db_table = 'RelatedFeatures'
```

Practical session: Loading spatial data and timeseries into the ODM2

Loading spatial data and timeseries into the ODM2

For this practical session, we will use data from the Cook Agronomy Farm, a long-term direct-seed cropping systems research program run by the Washington State University and the USDA. The data is produced by a field-scale sensor network deployed for monitoring and modeling the spatial and temporal variation of soil moisture in the farm. It is available free of charge on figshare via the link: **https://doi.org/10.15482/USDA.ADC/1349683**.

Department of Crop and Soil Sciences

Cook Agronomy Farm

Washington State University's newest farm, named for R. James Cook, was launched as a long-term direct-seed cropping systems research program by a team of WSU and USDA-ARS scientists in 2000.

To complete this practical exercise, you will need to **follow the instructions in the README file** of the dedicated github repository (link below).

pcelicourt/
agufm2025

1 Contributor 0 Issues 0 Stars 0 Forks

GitHub

agufm2025/README.md at geodjangoapp · pcelicourt/agufm2025

Contribute to pcelicourt/agufm2025 development by creating an account on GitHub.

Loading spatial data and timeseries into the ODM2

The next step of the practical session consists in loading spatial and timeseries data into an ODM2-based Database. You will need to **follow the instructions in the README file** below.

pcelicourt/
agufm2025

1 Contributor 0 Issues 0 Stars 0 Forks

GitHub

agufm2025/README.md at odm2loader · pcelicourt/agufm2025

Contribute to pcelicourt/agufm2025 development by creating an account on GitHub.

Loading spatial data and timeseries into the ODM2

If run successfully, the end result should look like the image below. The markers must be displayed and when hovering onany of them, the tooltip must display the description of the marker.

Back-end of a Django-based Web application

The **Django Template** component is a text file primarily written in HTML that may contain CSS and Javascript codes (within `<style>` and `<script>` tags) or links to external CSS and JavaScript files.

To create dynamic web pages, the HTML codes integrate specific elements or placeholders named tags, variables, and filters that are interpreted using the Django Template Language syntax (next slide) on the server side before the page is rendered in the browser.

GeeksforGeeks

Django Templates

Templates are the third and most important part of Django's MVT Structure. A template in Django is basically written in HTML, CSS, and Javascript in a .html file. Django framework efficiently handles and generates dynamic HTML web pages that are visible to the end-user.

Back-end of a Django-based Web application

The Django Template Language includes the following elements. The syntaxes are included in HTML files to control the data displayed in the browser.

Elements	Syntax	Description
Variables	{{ variable }}	Displays the value of a variable.
Comments	{# comments #}	Adds a comment that will not appear in the final rendering.
Blocks	{% block name %}...{% endblock %}	Defines an editable section in a parent template.
Conditions	{% if condition %}...{% endif %}	Executes a block of code if the condition is true.
Loops	{% for item in sequence %}...{% endfor %}	Iterates over a sequence and executes the block for each element.
Filters	{{ variable|filter }}	Modifies the display of a variable.
Template Inheritance	{% extends 'parent.html' %}	Indicates that the template inherits from a parent template.
Template Inclusion	{% include 'file.html' %}	Includes another template in the current template.

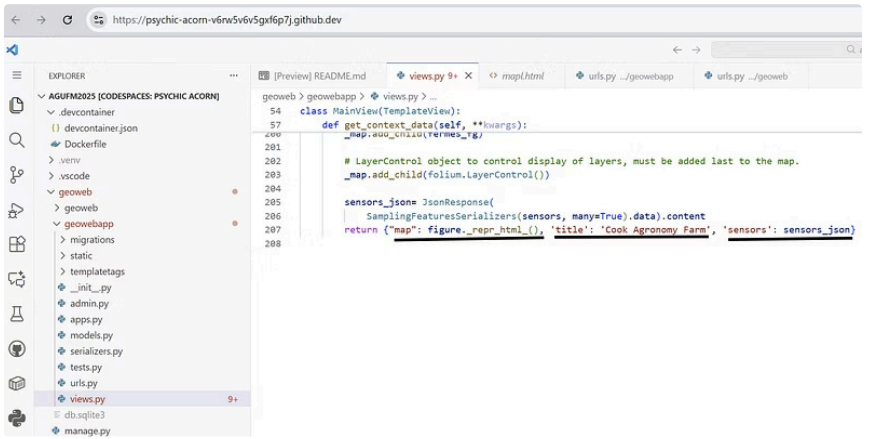
Django Project

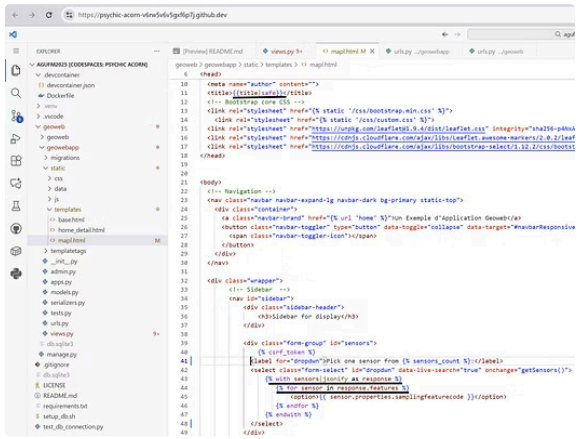
Built–in template tags and filters | Django documentation

The web framework for perfectionists with deadlines.

Back-end of a Django-based Web application

From the practical examples executed earlier, we demonstrate the importance of template tags (built-in and custom-made) as well as the relationship between Views and Templates.



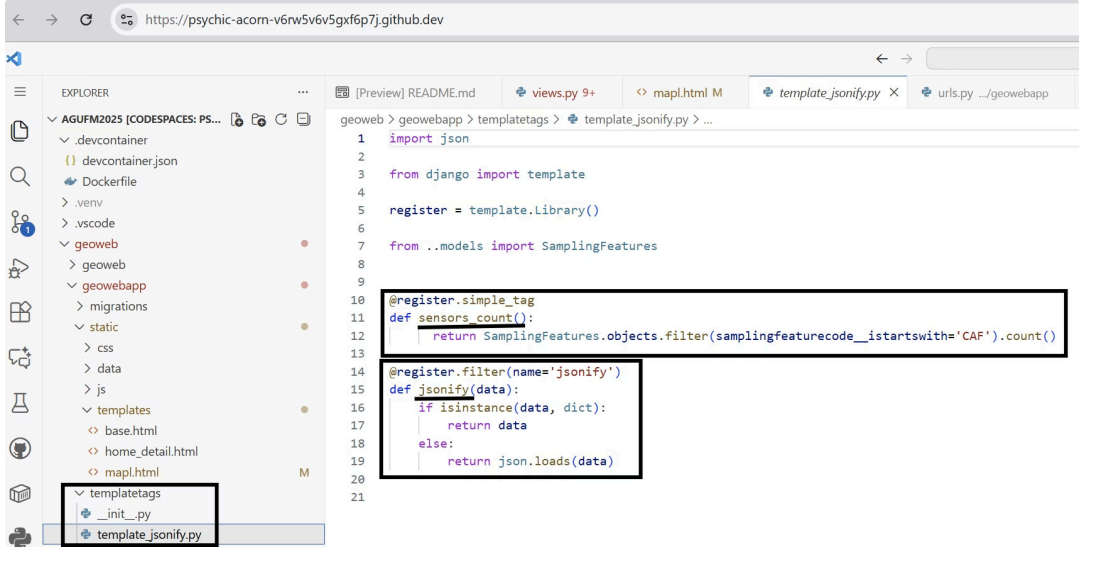


Rationale for template tags: When the page is being rendered, the **Views** has the ability to pass some variables along for displaying in the front-end (`{{title|safe}}` and `{% with sensors|jsonify as response %}` underlined in the left image above), in a dropdown button for example. The variable tag in the template is used to display those variables (the same tags underlined in the right image above), and other tags (**for** tag) can be used to loop over a variable, if it is a list for example.

Back-end of a Django-based Web application

OPTIONAL: Beyond built-in tags and filters, Django offers the ability to create custom tags and filters to render data in the browser.

The example shows a custom tag named `sensors_count` which calculates the total number of sensors stored in the database via the Model (table `SamplingFeatures` of the ODM2) from the previous example and registered using the `@register.simple_tag` decorator, where `register` is an instance of the `Library` class of the Django's `template` module.

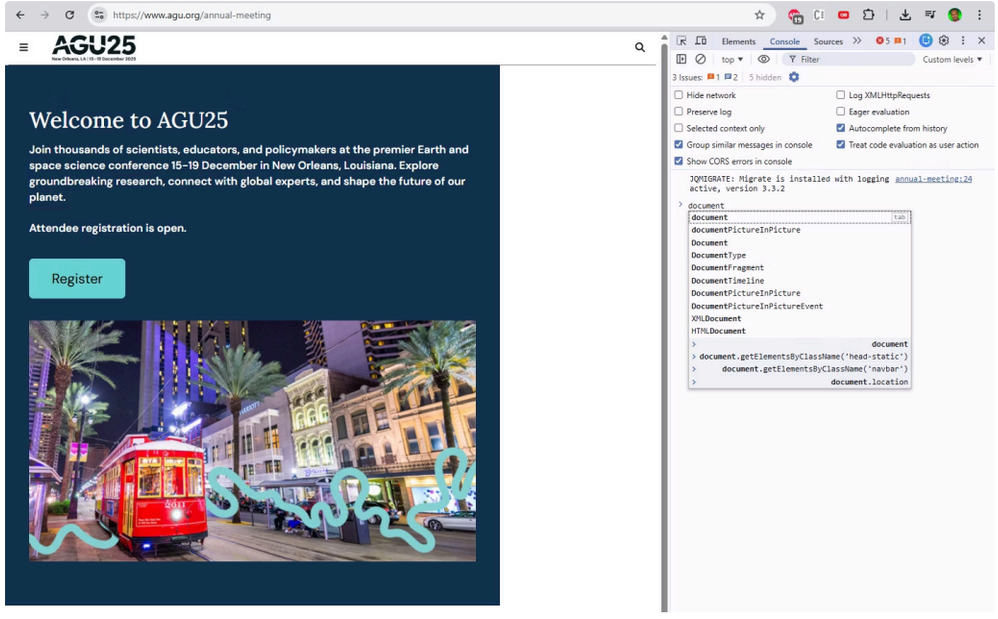


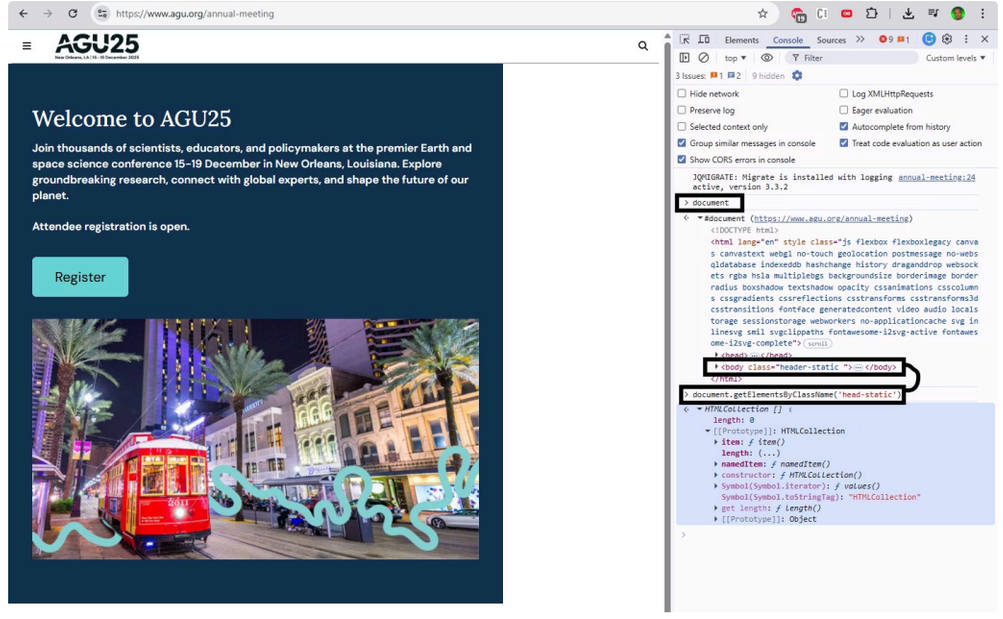
The module containing the custom tags and filters (`template_jsonify.py`) must be inside a package or folder named `templatetags`, which itself is located within the Django app (geowebapp). The command `{% load template_jsonify %}` and the `{% sensors_count %}` tag are be added to the template (mapl.html) in the correct place. Check the template file and find out!

Part III: Front-end Components of a Django-based application

The Document Object Model

The Document Object Model (DOM) is an **object-oriented API built in the browser**, among many others, that represents the structure and content of a document on the web. It is accessible via the keyword 'document' as shown below.





developer.mozilla.org

Introduction to the DOM – Web APIs | MDN

The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web. This guide will...

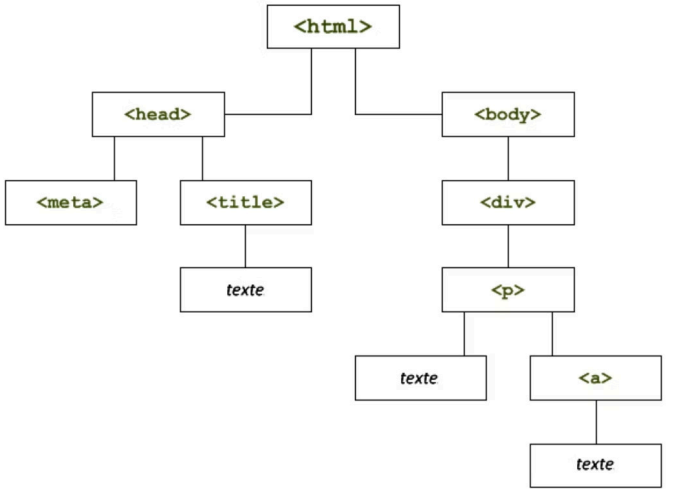
The Document Object Model

The content of a web page (text, images, and others) is annotated using different "tags" enclosed in angle brackets, which allows for display in a web browser.

The file header and the body of the page are delimited by the tags `<html>` and `</html>` and include special "elements" such as `<head>`, `<title>`, `<body>`, `<header>`, `<footer>`, `<div>`, `<select>`, `<nav>`, `<ul>`, `<ol>`, `<li>` and many others.

The minimal structure of a web page is represented as nested boxes or as a logical tree as follows:

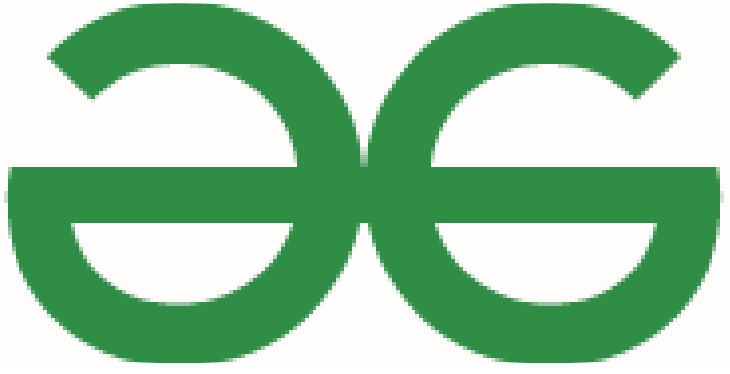
```
<html>
  <head>
    <title>Page title</title>
  </head>
  <body>
    <h1>This is a heading</h1>
    <p>This is a paragraph.</p>
    <p>This is another paragraph.</p>
  </body>
</html>
```



The Document Object Model

HTML tags can contain attributes composed of a name and a value that allow additional information to be stored. There are universal, event, data, and individual attributes.

Universal Attributes	Description	Example
id	Uniquely identifies an HTML element.	id="map"
class	Associates a class or category with one or more HTML elements.	class="navbar"
style	Defines the style of an HTML element, i.e., color, font, or font size.	style="color: blue; font-size: 2em"
title	Adds additional information about the content of an HTML element.	title="Geoinformatics Course"
lang	Defines the base language of a document.	<html lang="fr">



GeeksforGeeks

HTML Tags – A to Z List

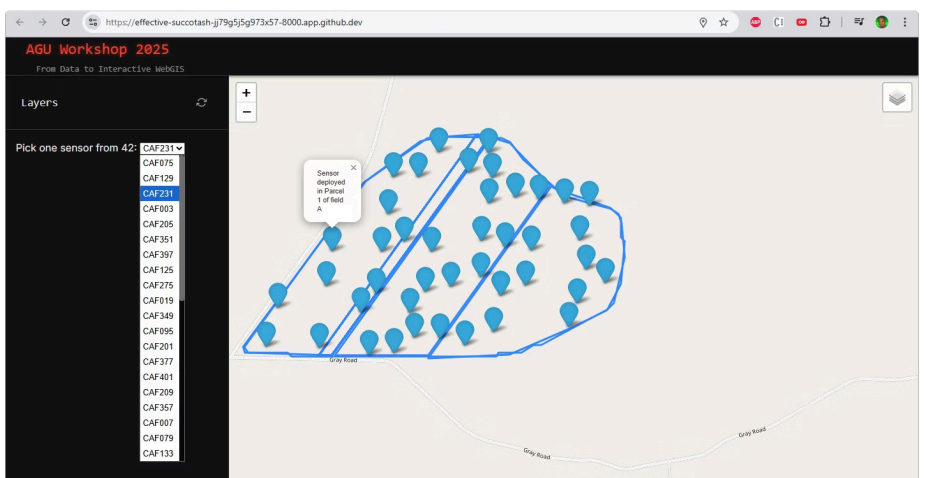
Your All-in-One Learning Portal: GeeksforGeeks is a comprehensive educational platform that empowers learners across domains-spanning computer science and...

The Document Object Model

A **web event** is an action or occurrence that triggers a signal in a system, which in turn, provides a mechanism (an event handler, event listeners, JavaScript function) to support the event.

The listener listens for the event that occurs, and the handler is the code that is executed in response to the event being triggered. `button.addEventListener("change", callbackfunction());`

An example of capturing a `change` event using a callback function (called callback in English) from a `select` element, `getSensors()`.



The Document Object Model

The JavaScript function example of the `change` event handler from the previous example with `Django` tags.

The HTML element in question, `select`, was assigned an `id` attribute defined by `id=dropdwn` which allows its value to be retrieved using the DOM as follows: `var sensorCode = document.getElementById("dropdwn").value;`

The CSRF (Cross-Site Request Forgery) token (`{% csrf_token %}`) generated by the server is a unique security measure designed to protect web applications against unauthorized or malicious requests.

The HTTP request method named **POST** is used to send data to the server for processing.

The Document Object Model

Native event attributes, i.e., those that are part of the standard DOM, are directly supported by the browser and can be listened to using the DOM's `addEventListener()` method.

Event Attributes	Description	Example
onclick	Triggers a JavaScript event via a mouse click.	<button onclick="function ( ) ">Click here</button>
onscroll	Triggers when the element is scrolled.	
onkeydown	Triggers when a key is pressed.	<input type="text" onkeydown="function ( ) ">
onsearch	Triggers a JavaScript function as soon as a search is entered in the search mask.	<input type="search" onsearch="function ( ) ">
onselect	Triggers a JavaScript function as soon as text is selected within the input element.	<input type="text" onselect="function ( ) " value="Example text">

MDN Web Docs

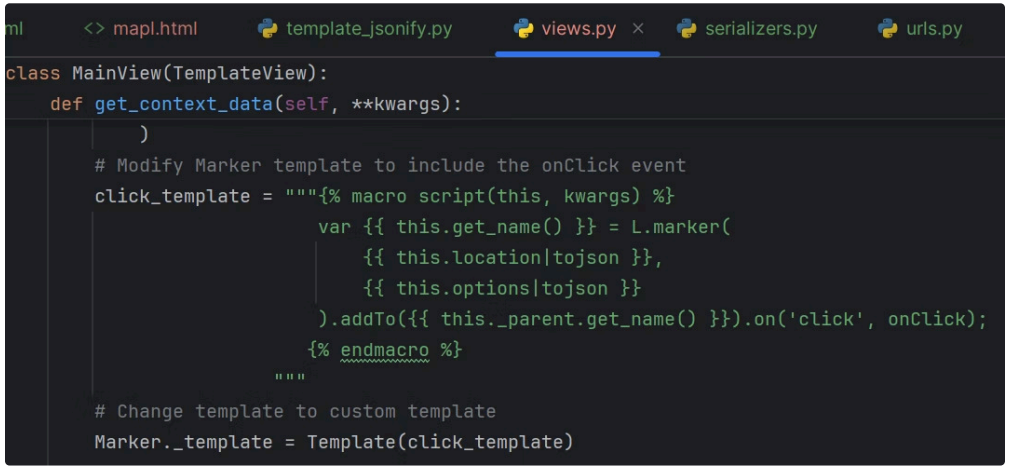
Introduction to events – Learn web development | MDN

Events are things that happen in the system you are programming, which the system tells you about so your code can react to them. For example, if the user clicks a...

The Document Object Model

`ADVANCED`: JavaScript event handler functions can be defined, not only on the client side (in a Django application's Templates), but also on the server side (in a Django application's Views when the folium library is used for creating maps).

This example illustrate the structure of a 'click' event handler for a 'folium' marker layer defined on the server side.



Step 1: Modification of the '_template' object of Folium's Marker class to add the event listener for the event of interest.

```
_template = Template(
    """
    {% macro script(this, kwargs) %}
        var {{ this.get_name() }} = L.marker(
            {{ this.location|tojson }},
            {{ this.options|tojavascript }}
        ).addTo({{ this._parent.get_name() }});
    {% endmacro %}
    """
)
```

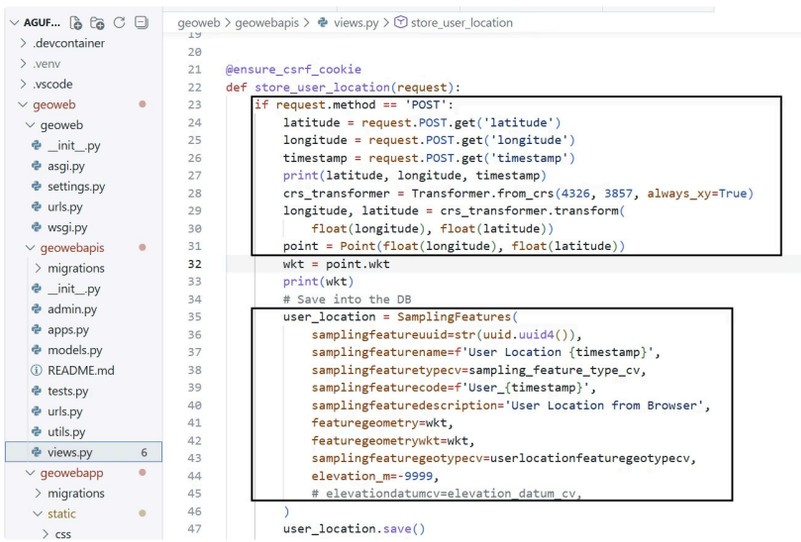
The original definition of the '_template' object in the folium library looks like the code above.

The Geolocation API

The **Geolocation API** allows the user to provide their location to web applications. User's permission is requested by the application and the user has to approve the request to share locations. The Geolocation API is accessed via a call to `navigator.geolocation`.

geoweb > geowebapp > static > js > eventhandler.js > ...

```
}

function getLocation() {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(
            function(position) {
                // Send precise client-side coordinates to the server
                const csrftoken = getCookie('csrftoken');
                console.log('position:', position);
                console.log('Sending location:', position.coords.latitude, position.coords.longitude);

                fetch("{% url 'user_location' %}", {
                    method: 'POST',
                    credentials: 'include',
                    headers: {
                        'Content-Type': 'application/x-www-form-urlencoded',
                        'X-CSRFToken': csrftoken
                    },
                    body: `latitude=${position.coords.latitude}&longitude=${position.coords.longitude}&timestamp=${position.timestamp}`
                })
                .then(response => response.json())
                .then(data => console.log('Location stored:', data))
                .catch(error => console.error('Error storing location:', error));
            },
```



MDN Web Docs

Geolocation API – Web APIs | MDN

The Geolocation API allows the user to provide their location to web applications if they so desire. For privacy reasons, the user is asked for permission to report locatio...

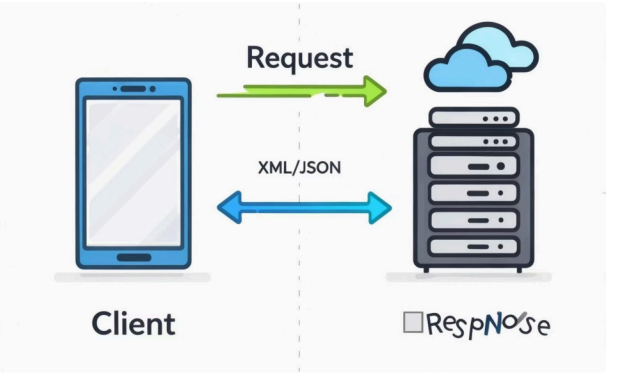
Part IV: Web services

Quick intro to Web Services

Web Services: web applications that allow data to be exchanged with other applications without necessarily being developed in the same language. This exchange is generally done using two protocols: SOAP or REST.

The simplified mechanism of a Web Service is as follows:

- The client or user with a connected device makes a request from a script or client software to the server hosting the Web Services;
- The request is transmitted to the server which processes it and delivers the response in a **predefined downloadable format** (XML, JSON or HTTP). The format can also be specified in the url using query strings.



Quick intro to Web Services

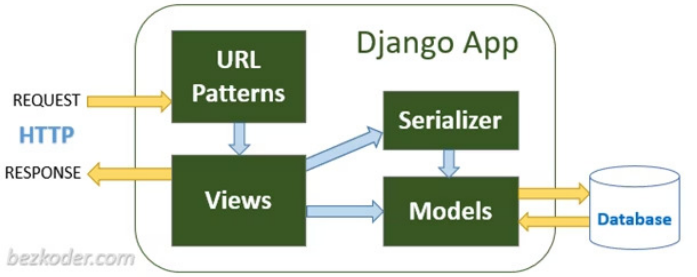
The **Representational State Transfer (REST)**: an approach to developing Web Services based on the design rules of **stateless services** where the server and client operate independently, and the server does not need any knowledge of prior messages to understand a received message. This is the key difference between the REST and SOAP approaches.

To allow access to resources, URLs identifying the resources as well as permitted HTTP operations (**GET**, PUT, DELETE, **POST**) are defined on the server side.

In Python, the **Django REST Framework (DRF)** is an extension of Django that facilitates the development of web services using the principles and components of a Django application. The architecture of a DRF application includes three main components: `Views`, `Models`, and **Serializers**.

Serializers have a special role and allow to **convert complex data structures, composed of data from one or more Django models, into easily consumable formats like JSON or XML**.

They handle the serialization and deserialization process effortlessly, thus simplifying data manipulation.



Quick intro to Web Services

A simple example of a web service using the **JsonResponse** class from Django.http module:

geoweb > geoweb > urls.py > ...

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

from geowebapp import views


urlpatterns = [
    path("admin/", admin.site.urls),
    path('', include('geowebapp.urls')),
    path('', include('geowebapis.urls')),
    path('', views.MainView.as_view(), name='home'),
]

```

geoweb > geowebapis > urls.py > ...

```python
from django.urls import path

from .views import sensor_data


urlpatterns = [
    path('sensor/', sensor_data, name='sensor'),
]
```

API endpoint : a specific URL used to access a resource provided by a Web application. The API receives requests and returns responses from this URL.

[Preview] README.md | README.md | urls.py | views.py .../geowebapis 2

geoweb > geowebapis > views.py > sensor_data

```python
from django.shortcuts import render
from django.http import JsonResponse
from django.views.decorators.csrf import ensure_csrf_cookie
import uuid

import requests
from geowebapp.models import SamplingFeatures, TimeSeriesResults, \
    TimeSeriesResultValues, Results, FeatureActions, CV_SamplingFeatureGe
from shapely.geometry import Point
from pyproj import CRS, Transformer

from geowebapp.serializers import SamplingFeaturesSerializers


@ensure_csrf_cookie
def sensor_data(request):
    if request.method == 'POST':
        latitude = request.POST.get('latitude')
        longitude = request.POST.get('longitude')
        sensor_code = request.POST.get('sensor_name')
    elif request.method == 'GET':
        latitude = request.GET.get('latitude')
        longitude = request.GET.get('longitude')
        sensor_code = request.GET.get('sensor_name')

    print(latitude, longitude, sensor_code)
    sampling_feature = SamplingFeatures.objects.filter(
        samplingfeaturecode=f"{sensor_code}").first()

    featureaction = FeatureActions.objects.filter(
        samplingfeatureid=sampling_feature).first()

    results = Results.objects.filter(
        featureactionid=featureaction).first()

    timeseries_results = TimeSeriesResults.objects.filter(
        resultid=results).first()

    timeseries_values = TimeSeriesResultValues.objects.filter(
        resultid=timeseries_results).all().order_by('-valuedatetime')[:5]

    # Convert query results to JSON-serializable format
    values_list = []
    for value in timeseries_values:
        values_list.append({
            'value': float(value.datavalue),
            'datetime': value.valuedatetime.isoformat() if value.valuedat
        })
        print(f"Value: {value.datavalue}, DateTime: {value.valuedatetime}

    variable = results.variableid.variablenamecv.term
    print(f"Variable: {variable}")
    unit = results.unitsid.unitsname
    print(f"Unit: {unit}")

    # Return structured JSON data
    return JsonResponse({
        'status': 'success',
        'sensor_code': sensor_code,
        'variable': variable,
        'unit': unit,
        'data': values_list,
        'count': len(values_list)
    })

```

Quick intro to Web Services

Creating a DRF application: To separate the web service from other Django or GeoDjango applications, you need to create a new application in your Django project by executing the command: `python manage.py startapp geowebapis` from your Django project directory.

You need to add `djangorestframework` and `djangorestframework-gis` and the new app to your Django project as you did for GeoDjango, i.e., by adding them to your INSTALLED_APPS parameter in the project's `settings.py` file as shown opposite.

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    'django.contrib.gis',
    'geowebapp',
    'webservice',
    'rest_framework',
    'rest_framework_gis',
    'leaflet',
]
```

Quick intro to Web Services

For **non-spatial data serialization**, the `django-rest-framework` provides the `ModelSerializer` class as a shortcut to automatically create a Serializer class with fields corresponding to those of a Django Model.

For **geospatial data serialization**, the `django-rest-framework-gis` library provides the `GeoFeatureModelSerializer` class, to automatically create a Serializer class with fields corresponding to those of a Django Model with geometric attributes such a SamplingFeatures of the ODM2. It requires the definition of a *geo_field* attribute that will be formatted like the "geometry" type field of the Model to be encoded.

geoweb > geowebapp > serializers.py > SamplingFeaturesSerializers > Meta

```python
from rest_framework_gis.serializers import GeoFeatureModelSerializer


from .models import SamplingFeatures

class SamplingFeaturesSerializers(GeoFeatureModelSerializer):
    """ A class to serialize locations as GeoJSON compatible data """

    class Meta:
        app_label  = 'geowebapp'
        model = SamplingFeatures
        geo_field = "featuregeometry"
        fields = '__all__'
```

geoweb > geowebapis > views.py > ...

```python
from geowebapp.models import SamplingFeatures, TimeSeriesResults, \
    TimeSeriesResultValues, Results, FeatureActions, CV_SamplingFeatureGeoType,

from geowebapp.serializers import SamplingFeaturesSerializers


@ensure_csrf_cookie
def sampling_features(request):
    all_features = SamplingFeatures.objects.all()
    serializer = SamplingFeaturesSerializers(all_features, many=True)
    return JsonResponse(serializer.data, safe=False)
```

[Preview] README.md | README.md M | urls.py .../geowebapis M | urls.py .../geoweb

geoweb > geowebapis > urls.py > ...

```python
from django.urls import path

from .views import sensor_data, sampling_features


urlpatterns = [
    path('sensor/', sensor_data, name='sensor'),
    path('samplingfeatures/', sampling_features, name='samplingfeatures'),
]
```

https://effective-succotash-jj79g5j5g973x57-8000.app.github.dev/samplingfeatures/

Pretty-print


```
    ]
  },
  "properties": {
    "samplingfeatureuuid": "8ff22d60-8843-49ea-a643-661216ff6a33",
    "samplingfeaturecode": "FieldB",
    "samplingfeaturename": "Field B",
    "samplingfeaturedescription": "A field of the Cook Agronomy Farm",
    "featuregeometrywkt": "MULTIPOLYGON (((-13033186.682045 5906676.076225, -13033175.391663 5906665.866896, -13033175.298533 5906665.765118, -13033162.910345 5906649.351468, -13033162.801939
5906649.147186, -13033162.762574 5906648.919155, -13033162.796192 5906648.690200, -13033162.899428 5906648.483240, -13033618.165419 5906012.527873, -13033618.372073 5906012.333728, -13033618.636868
5906012.233189, -13033633.912808 5906009.659020, -13033634.058845 5906009.649353, -13033652.095217 5906010.276373, -13033682.716157 5906011.203677, -13033682.722917 5906011.194218, -13033682.889688
5906011.026643, -13033683.101086 5906010.921296, -13033683.334955 5906010.889220, -13033706.516356 5906011.510417, -13033736.461062 5906011.478357, -13033736.685715 5906011.513689, -13033736.888592
5906011.616729, -13033737.013593 5906011.741234, -13033779.892970 5906008.848364, -13033779.927859 5906008.846851, -13033797.454949 5906008.508263, -13033797.619458 5906008.343937, -13033797.831123
5906008.239226, -13033798.065041 5906008.207824, -13033836.540693 5906009.347719, -13033851.943774 5906009.170767, -13033851.950912 5906009.171819, -13033851.957935 5906009.170742, -13033911.928196
5906009.650086, -13033912.104596 5906009.405338, -13033912.277737 5906009.234401, -13033912.497249 5906009.130013, -13033912.738763 5906009.103762, -13033967.393185 5906012.366521, -13033967.609819
5906012.413293, -13033967.802442 5906012.523187, -13033967.953262 5906012.686050, -13033968.048346 5906012.886841, -13033968.078913 5906013.107011, -13033968.042139 5906013.326224, -13033967.941420
5906013.524231, -13033382.761514 5906828.164143, -13033382.620865 5906828.311327, -13033382.445182 5906828.413851, -13033382.248116 5906828.463747, -13033382.044976 5906828.457141, -13033341.476807
5906821.377789, -13033341.315225 5906821.329762, -13033331.122074 5906816.959845, -13033331.050210 5906816.915318, -13033330.985407 5906816.882878, -13033328.510796 5906815.113544, -13033322.043536
5906812.598181, -13033321.923957 5906812.538513, -13033314.277622 5906807.800938, -13033314.129342 5906807.679427, -13033308.665187 5906801.844521, -13033308.652954 5906801.831129, -13033296.631171
5906788.337284, -13033296.500921 5906788.129128, -13033296.446750 5906787.889444, -13033296.466350 5906787.718665, -13033264.777913 5906750.648843, -13033264.577619 5906750.630307, -13033264.378290
5906750.548476, -13033264.211486 5906750.411800, -13033250.005658 5906734.731457, -13033234.522616 5906717.821895, -13033234.464879 5906717.827700, -13033234.225081 5906717.768746, -13033234.018641
5906717.632895, -13033217.270808 5906701.959626, -13033200.523906 5906687.016251, -13033200.387136 5906686.852448, -13033200.303392 5906686.655968, -13033200.282743 5906686.469926, -13033186.728468
5906676.114836, -13033186.682045 5906676.076225)))",
    "elevation_m": -9999,
    "samplingfeaturetypecv": "fieldArea",
    "samplingfeaturegeotypecv": null,
    "elevationdatumcv": "MSL"
  }
},
```

Part V: Interactive frontend development techniques & backend integration

Practical session

pcelicourt/
agufm2025

1 Contributor
0 Issues
0 Stars
0 Forks

GitHub

agufm2025/README.md at api-frontend · pcelicourt/agufm2025

Contribute to pcelicourt/agufm2025 development by creating an account on GitHub.

Wrap-up

Completed an end-to-end theoretical and practical workshop on Web-GIS applications using the Python language and the Django ecosystem libraries.

- General Introduction to Python and the Django library ecosystem
- Basic understanding of the Data modeling processes and the Observations Data Model
- Discussion on front-end and back-end programming
- Integration of browser APIS and development of custom-made APIs
- Interactive graphical user interface with two-way communication to the back-end

Next steps:

- Deployment on your application using Github codespace or an online platform such as **render.com**
- Follow-up survey by AGU
- Contact: **paul.celicourt@inrs.ca**

Appendix

Object-Oriented Programming in Python

Inheritance: To retain the inheritance of the parent's `__init__()` function, we call the parent class's `__init__()` function using the parent's name in the child class:

```
class ProvinceState(Ville):
    def __init__(self, nom, province_etat, pays, population_M, localisation):
        Ville.__init__(self, nom, province_etat, pays, population_M, localisation)
```

```
prov_state = ProvinceState('Quebec','QC', 'Canada', 8.5, (53.000000, -70.000000))
```

```
prov_state.getPopulation()
```

```
8.5
```

```
prov_state.getLocation()
```

```
(53.0, -70.0)
```

```
prov_state._position_geographique
```

```
(53.0, -70.0)
```

```
prov_state.province_etat
```

```
'QC'
```

Back-end of a Django-based Web application

Logical data modeling is a more detailed representation of data, derived from **conceptual modeling**. It follows these rules:

- **Rule 1**: Each entity from the conceptual model becomes a table in the logical model.
- **Rule 2**: Each identifier of an entity from the conceptual model becomes a `primary key` of the corresponding table in the logical model. Other properties become the attributes of the table.
- **Rule 3**: The values of the cardinalities of each association are defined by (a) adding a `foreign key` to an existing table or (b) creating a new table whose `primary key` is obtained by concatenating foreign keys corresponding to the linked entities.

The `primary key` uniquely identifies a record in a table (the *Identifier* attribute); the `candidate key` is a minimum set of attributes that can act as the primary key; the `foreign key` creates a link with the primary key of another table (the *IdParcelle* and *IdBatiment* attributes).