

Lecture 10 – Data Model Implementation

Open Data Management & the Cloud

(Data Science & Scientific Computing / UniTS – DMG)

Data model implementation



- Once we have designed a data model in UML, we need to convert the diagrams into machine readable formats
 - To perform additional validations to the data model, e.g. homogeneity, common naming rules
 - To be able to persist objects and relations which are compliant with the designed data model
- The implementation depends on the underlying technology:
 - For relational databases: database schema
 - For document oriented databases: the XML Schema Language (XSD) or the JSON Schema (JavaScript Object Notation)
- Document based systems can also be built on top of relational databases
- In this lecture we will focus on the relational database schema.

Object-relational impedance mismatch

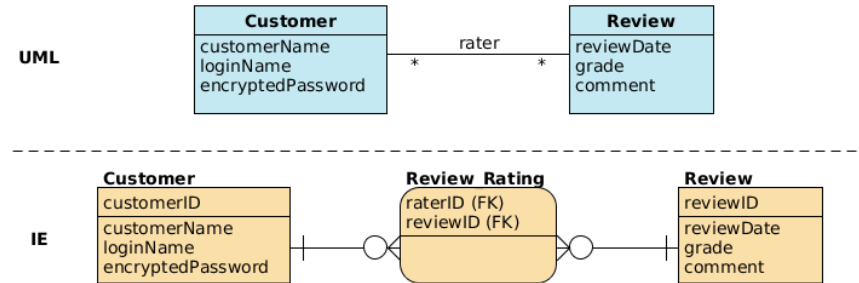


- A set of conceptual and technical difficulties that are often encountered when a relational database management system is being served by an application program written in an object oriented language
- We have already discussed some solutions when comparing the UML model with the IE model in the previous lecture
- Additional difficulties:
 - Hierarchical structure:
 - In UML, we can define complex hierarchical structures. A class can “aggregate” instances of other classes. The relational model only “accepts” atomic types for the entity attributes and relations
 - In the relational model, children point to their parent, while in the hierarchical model parents point to their children
 - Inheritance:
 - Not directly supported by the relational model. Several mappings can be implemented to keep the inheritance information
 - Class normalization vs data normalization

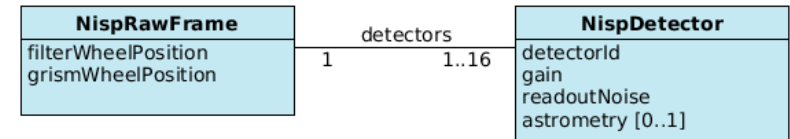
Examples



- Many-to-many associations, when mapped to a relational schema, require an additional table, i.e. an additional relation



- In the relational schema we cannot define an upper limit on the multiplicity



- Abstract classes have multiple mapping options, each one with some limitations

Specialization and generalization

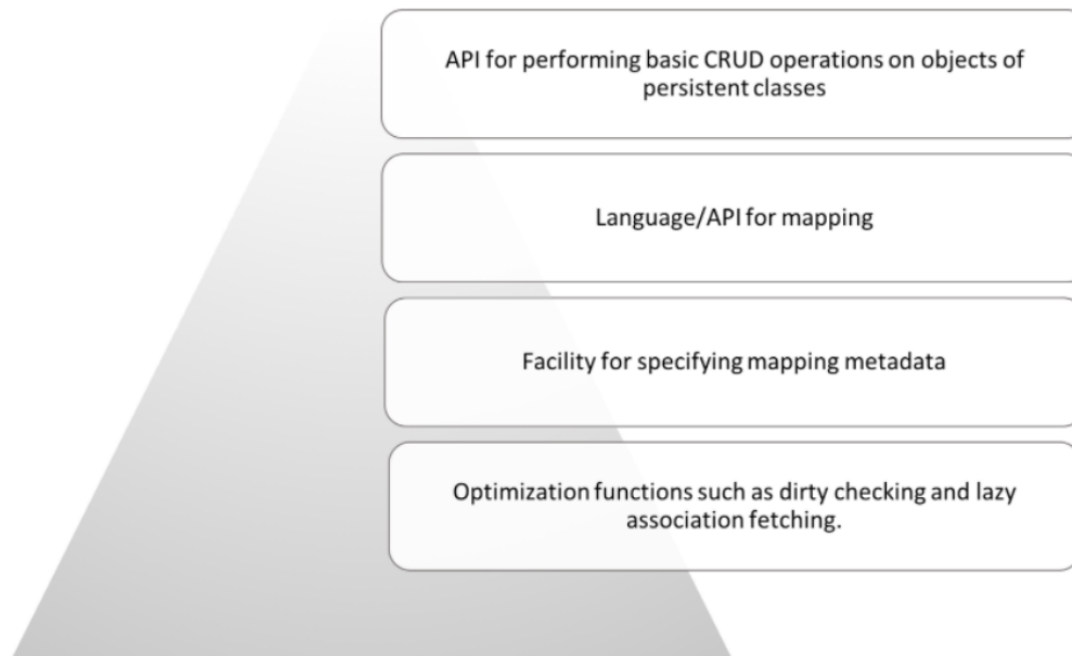


- We consider here only the single inheritance
- To convert each specialization with m subclasses $\{S_1, S_2, \dots, S_m\}$ and superclass C , where the attributes of C are $\{k, a_1, a_2, \dots, a_n\}$ and k is the primary key, into a relation schema, the options are:
 - **Multiple relations - superclass and subclasses.** Create a relation L for C with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$ and $\text{PK}(L) = k$. Create a relation L_i for each subclass S_i , with attributes $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $\text{PK}(L_i) = k$.
 - **Multiple relations – subclass only.** Create a relation L_i for each subclass S_i , with the attributes $\text{Attrs}(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \dots, a_n\}$ and $\text{PK}(L_i) = k$.
 - **Single relation with one type attribute.** Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t\}$ and $\text{PK}(L) = k$. The attribute t is called type (or discriminating) attribute whose value indicates the subclass to which each tuple belongs
 - **Single relation multiple type attributes.** As above, but instead of a single type attribute t , there is a set $\{t_1, t_2, \dots, t_m\}$ of m boolean type attributes indicating whether or not a tuple belongs to subclass S_i .

Object-Relational Mapping (ORM)

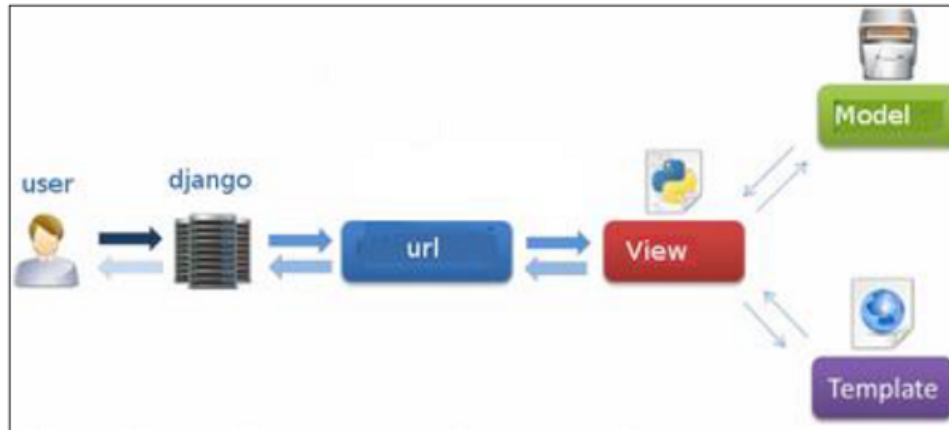


- **Object-relational mapping (ORM)** uses different tools, technologies and techniques to map data objects in a target programming language to relations and tables of a RDBMS
- An ORM solution consists of the following four pieces:



- An ORM abstracts your application away from the underlying SQL database and SQL dialect
- If the tool supports a number of different databases (and most do), this confers a certain level of portability on your application
- Several programming languages have at least one ORM solution
 - Java: it provides both a standard specification, named Java Persistence API (JPA), and several implementations of the specification (Hibernate, EclipseLink)
 - C++: possible ORM solutions are
 - ODB: <https://www.codesynthesis.com/products/odb>
 - QxOrm: https://www.qxorm.com/qxorm_en/home.html
 - Python:
 - **SQLAlchemy**: <https://www.sqlalchemy.org/>
 - The **Django** framework: <https://docs.djangoproject.com/en/2.1/topics/db/>
 - Pony: <https://ponyorm.com/>
 - Ruby: ActiveRecord, DataMapper, Sequel

- Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design <https://www.djangoproject.com>
- Django follows the model-template-view (MTV) architectural pattern
 - An object-relational mapper, defining a data model as python classes (**Models**)
 - A system for processing HTTP requests (**Views**) with a web templating system (**Template**)
 - A regular-expression-based URL dispatcher (**Url**)

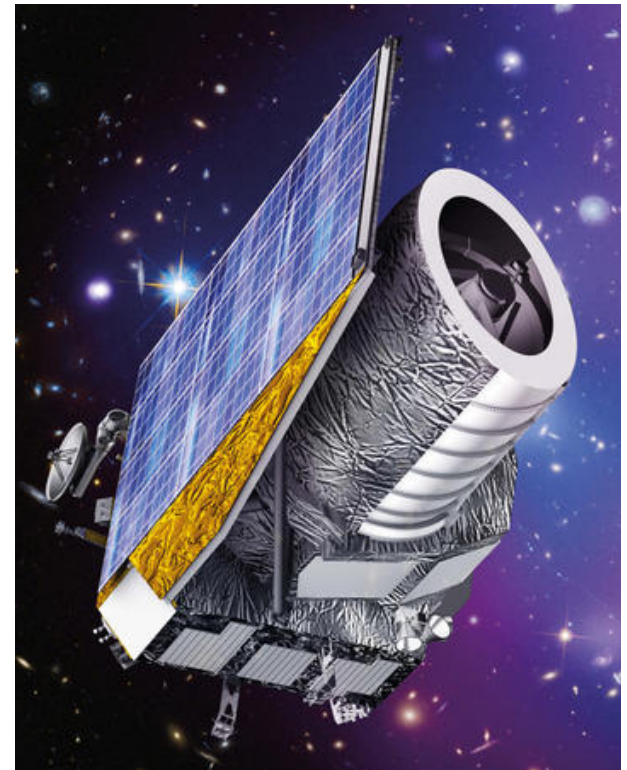


- Django comes with a lightweight standalone web server for development and testing
- A serialization system that can produce and read XML and/or JSON representation of Django models
- Lot of reusable packages provided by the community:
<https://djangopackages.org/>

Example from the Euclid mission



- M2 mission in the framework of **ESA Cosmic Vision Program**
- Euclid mission objective is to map the geometry and understand the nature of the **dark Universe** (dark energy and dark matter)
- **Federation** of 8 European + 1 US Science Data Centers and a Science Operation Center (ESA)
- Large amount of data produced by the mission
 - Due to reprocessing
 - Large amount of **external data** needed (ground based observations)
 - Grand total: **90 PB**
- Two instruments on board:
 - VIS: Visible Imager
 - NISP: Near Infrared Spectro-Photometer



A NISP instrument simulated image

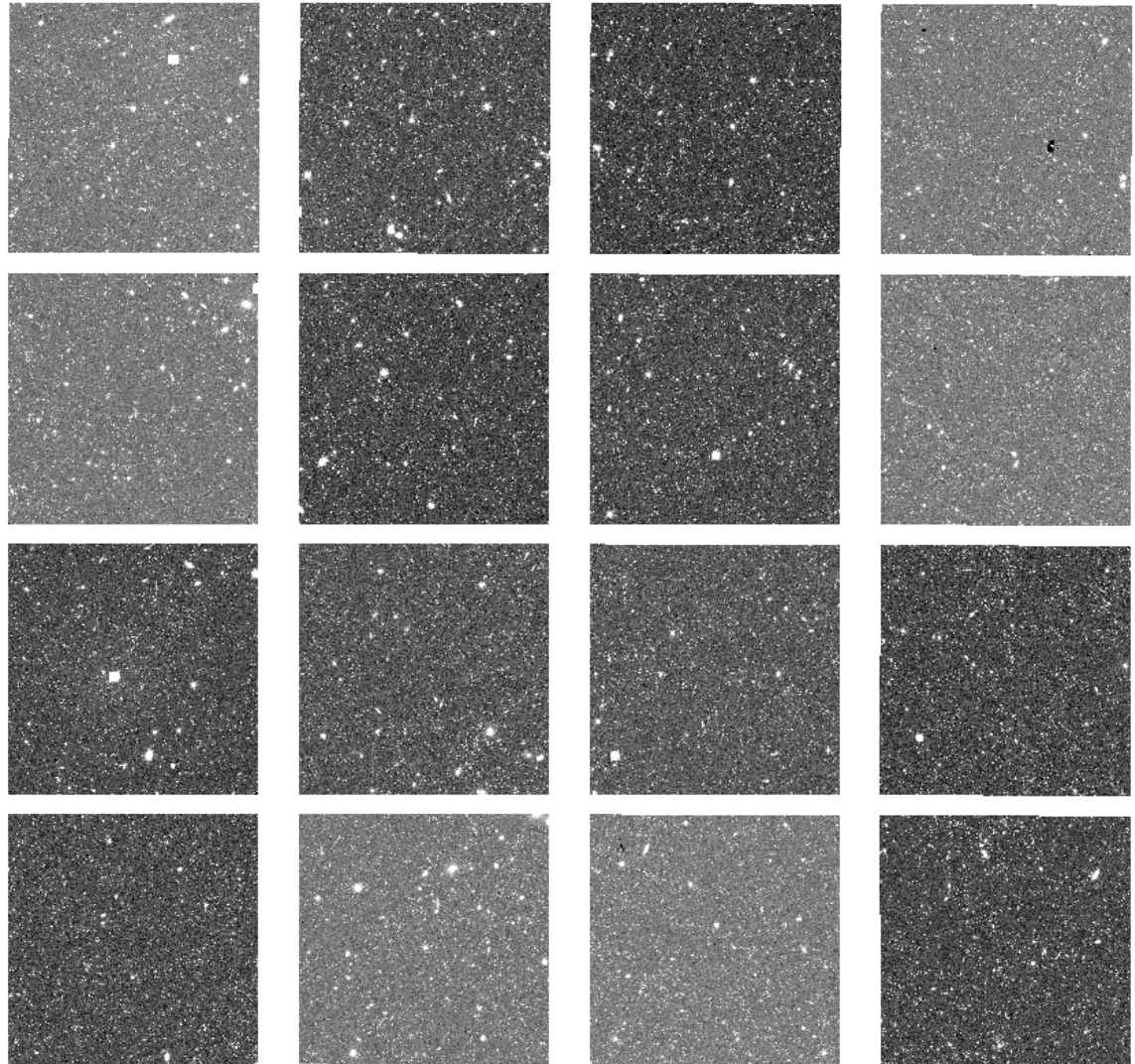


The NISP focal plane is composed of a matrix of 4×4 2040×2040 18 micron pixel detectors

The photometric channel is equipped with 3 broad band filters (Y, J and H)

The spectroscopic channel is equipped with 4 different low resolution near infrared grisms (three red and one blue) but no slit

The image on the right shows a NISP frame composed by its 16 detectors (photometric channel, 1 band)

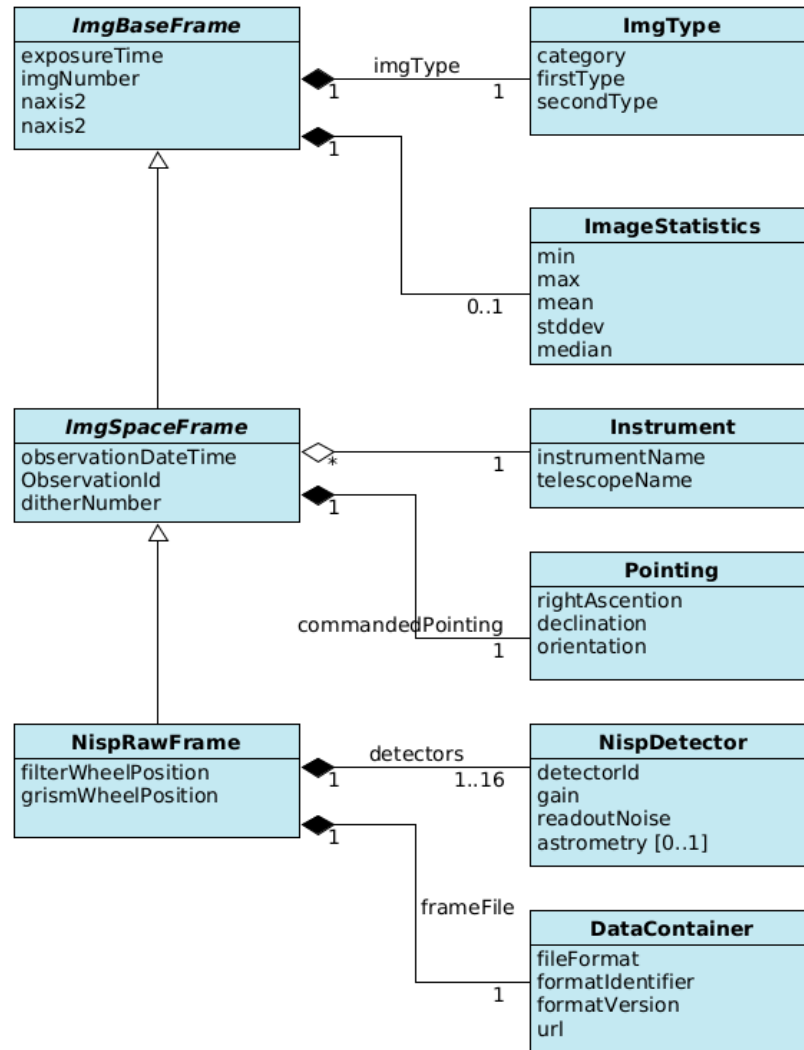


Metadata content (simplified)



- We need to define the metadata associated to a NISP image (a single exposure)
- Since Euclid also needs images from **ground-based telescopes**, the **dictionary** of types used to model the metadata information should be homogeneous among them and **reuse a common base set** of type definitions
- All images have a common set of information
 - **Exposure time, image category** and purpose (is it a simulation, a calibration image, a sky image, etc.) and image **dimensions**, some statistics on the image, to quickly check if there are anomalies, and we need to keep the information about the **instrument** used to acquire a given image
 - However, for ground-based telescopes we need also the geographical location of the telescope, so the telescope information requires more properties.
 - Space telescopes can perform **surveys** of the sky, hence the observation can be identified by the **observation ID**. Moreover, for a given field, they can execute a **dithering** pattern, in order to increase the signal-to-noise ratio and reduce cosmic-ray hits. So we need also to store the dither number. Additional information needed are the **observation date and time** and the **commanded pointing** (right ascension, declination and telescope orientation)
- Then we have information specific to the Euclid instruments. The NISP instrument has both a filter wheel and a grism wheel. The images from **all detectors** should be stored in a single file, to simplify its retrieval and the analysis. However, each detector has some specific properties: **gain, readout noise**. Then, for each detector we need to compute the mapping from pixel indexes to sky coordinates (RA, DEC), i.e. its own **astrometric solution**.

The NISP image data model (simplified)



Django Implementation (1)



- Each class inherits `models.Model`
- All fields use a Django Model Data Type
<https://www.webforefront.com/django/modeldatatypesandvalidation.html>
 - `models.CharField(max_length = 20)`
 - `models.BooleanField()`
 - `models.FloatField()`
 - `models.DateTimeField()`
 - ...

- Attributes in the Data Model Type are used to set options for fields
 - `null = True`
 - `primary_key = True`

- Foreign keys
<https://docs.djangoproject.com/en/4.1/ref/models/fields/#django.db.models.ForeignKey>

```
instrument = models.ForeignKey(Instrument)
```

- Related names
<https://docs.djangoproject.com/en/4.1/topics/db/queries/#backwards-related-objects>

```
rawFrame = models.ForeignKey('NispRawFrame', related_name='detectors', on_delete=models.CASCADE)
```

Django Implementation (2)



- By **enumerated type** we mean a type that provides a set of possible values through the **choices** parameter (option) available to all field types

```
IMAGE_CATEGORY = ( 'SCIENCE', 'CALIBRATION', 'SIMULATION' )
```

```
category = models.CharField( max_length=20, choices=[(d, d) for d in IMAGE_CATEGORY])
```

- Model Meta options is “anything that’s not a field”

```
class Meta:  
    abstract = True
```

- Abstract class

```
class Meta:  
    ordering = ['surname']
```

- Ordering

```
class Meta:  
    unique_together = (("fiscalCode1", "fiscalCode2"),)
```

- Candidate key of multiple columns

- ...

- It is a good practice to override the default name of objects

```
def __str__(self):  
    return self.name
```

Prerequisites



- The simplest way to install Django is to download and install the Python Anaconda Distribution, with Python version 3.x:
<https://www.anaconda.com/download>
- Then you need to install some additional python packages for the following exercise/hands-on:

- To install the Django framework use the following command line:

```
conda create -n orm_django django
```

- Additional packages are needed, not available in Anaconda but installed with the “pip” command:

```
pip install django-extensions djangorestframework  
pip install django-composite-field django-ufilter  
pip install django-phonenumbers-field phonenumbers  
pip install Pillow
```

- The full Anaconda distribution already provides Jupyter notebooks, used in one example

ORM project example - Euclid



- The entire examples can be retrieved at the following link:

https://www.ict.inaf.it/gitlab/odmc/orm_example

- You can clone the project with the git version control system, i.e. with the command:

```
git clone https://www.ict.inaf.it/gitlab/odmc/orm_example.git
cd orm_example/django_example_euclid
```

- The Django project for the Euclid example has been already created using the following commands (so you **don't need** to execute them):

```
django-admin startproject euclid_example ./
python manage.py startapp imagedb
```

which creates a project folder, named **euclid_example**, with additional files and then an application, named **imagedb**, inside the project.
It automatically creates skeleton files needed by a django project and application

Project structure



django_example_euclid/

imagedb
— admin.py
— apps.py
— migrations
— models.py
— tests.py
— views.py

manage.py
euclid_example
— settings.py
— urls.py
— wsgi.py

File containing the app data model

Views on the data model classes

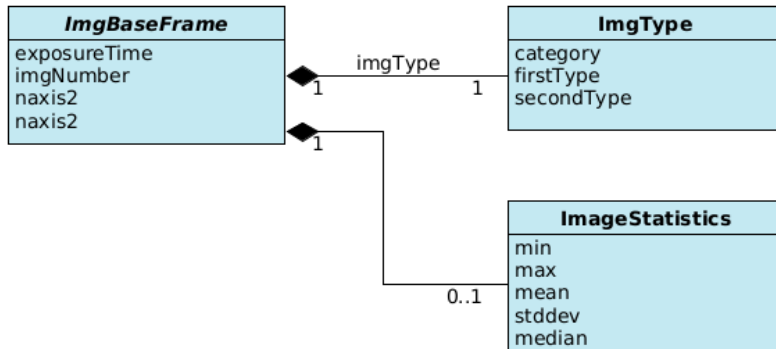
Project settings: app list and configuration

Site urls declaration

The Django ORM



- From the data model class to a Django ORM model class



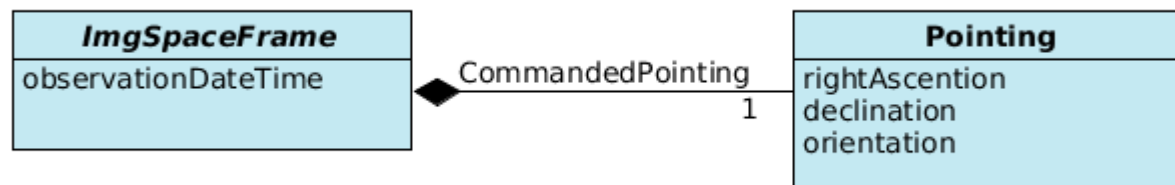
- Each model is represented by a class that subclasses **django.db.models.Model**
- ImageBaseFrame here is **abstract**: no table instantiated
 - That's why we define the stats attribute as a Foreign Key to the ImageStatistics class and not vice versa

```
from django.db import models
```

```
class ImageBaseFrame(models.Model):
    exposureTime = models.FloatField()
    imgNumber = models.PositiveSmallIntegerField()
    naxis1 = models.PositiveIntegerField()
    naxis2 = models.PositiveIntegerField()
    imgType = ImageType()
    stats = models.OneToOneField(
        ImageStatistics,
        models.SET_NULL,
        blank=True,
        null=True,
    )
```

```
class Meta:
    abstract = True
```

- Sometime we would like to define a model class attribute as a **multi-column field** in the same table (i.e. a non-atomic type) instead of creating a 1-to-1 relation (a second table with the attribute columns and a foreign key)
- Many ORM systems provide such feature:
 - JPA: named as **embeddable classes**
 - odb: named as **Composite Value Types**
 - SQLAlchemy: named as **Composite Column Types**
- Django ORM does not provide directly this feature. However there is a package provided by the community, called `django-composite-field`, which provides an “acceptable” solution
- Composite fields provide an implementation of a “part-of” relationship, i.e. what in the UML class diagram is called **composition**



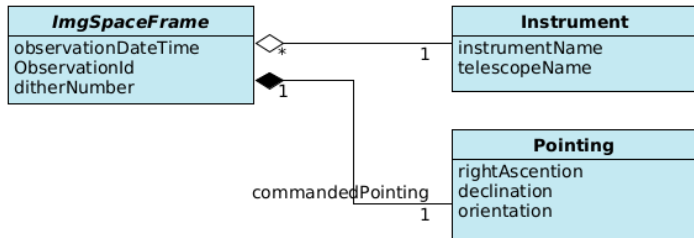
The ImageType class



```
IMAGE_CATEGORY = (  
    'SCIENCE',  
    'CALIBRATION',  
    'SIMULATION'  
)  
  
IMAGE_FIRST_GROUP = (  
    'OBJECT',  
    'STD',  
    'BIAS',  
    'DARK',  
    'FLAT',  
    'LINEARITY',  
    'OTHER'  
)  
  
IMAGE_SECOND_GROUP = (  
    'SKY',  
    'LAMP',  
    'DOME',  
    'OTHER'  
)
```

```
from composite_field import CompositeField  
  
class ImageType(CompositeField):  
  
    category = models.CharField(  
        max_length=20,  
        choices=[(d, d) for d in IMAGE_CATEGORY]  
    )  
  
    firstType = models.CharField(  
        max_length=20,  
        choices=[(d,d) for d in IMAGE_FIRST_GROUP]  
    )  
  
    secondType = models.CharField(  
        max_length=20,  
        choices=[(d,d) for d in IMAGE_SECOND_GROUP]  
    )
```

The ImageSpaceFrame class



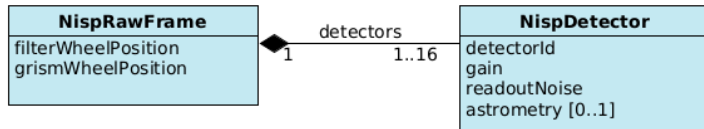
```
class Instrument(models.Model):
    instrumentName = models.CharField(max_length=100)
    telescopeName = models.CharField(max_length=100)

class Pointing(CompositeField):
    rightAscension = models.FloatField()
    declination = models.FloatField()
    orientation = models.FloatField()

class ImageSpaceFrame(ImageBaseFrame):
    observationDateTime = models.DateTimeField()
    observationId = models.PositiveIntegerField()
    ditherNumber = PositiveSmallIntegerField()
    instrument = models.ForeignKey(Instrument,
                                   on_delete=models.CASCADE)
    commandedPointing = Pointing()

class Meta:
    abstract = True
```

- The same Instrument is associated to many images, hence here we use a Foreign Key from ImageSpaceFrame to Instrument
- If the Instrument instance is deleted, also all images referring to it are automatically deleted (option **on_delete** set to `models.CASCADE` in `ForeignKey`)

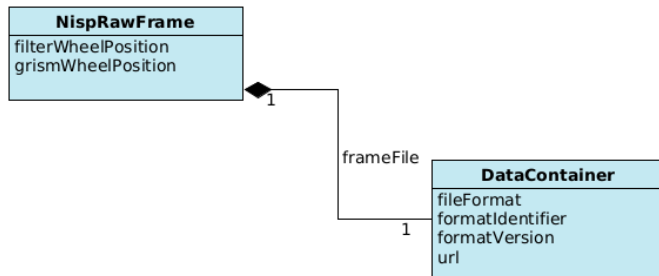


```
NISP_DETECTOR_ID = (
    '11', '12', '13', '14',
    '21', '22', '23', '24',
    '31', '32', '33', '34',
    '41', '42', '43', '44'
)
```

```
class NispDetector(models.Model):
    detectorId = models.CharField(
        max_length=2,
        choices = [(d,d) for d in NISP_DETECTOR_ID]
    )
    gain = models.FloatField()
    readoutNoise = models.FloatField()
    rawFrame = models.ForeignKey('NispRawFrame',
                                related_name='detectors',
                                on_delete=models.CASCADE)
```

- Many detectors (up to 16) associated to the same raw frame
- Since NispRawFrame is not yet defined, we pass the class name as a string to models.ForeignKey
- But we want to access the detector data using the NispRawFrame class, i.e. the reverse relation.
- This is the purpose of the related_name parameter. For instance we can access the detector data using NispRawFrame.detectors

NispRawFrame class



```
NISP_FILTER_WHEEL = (
    'Y',
    'J',
    'H',
    'OPEN',
    'CLOSE'
)

NISP_GRISM_WHEEL = (
    'BLUE0',
    'RED0',
    'RED90',
    'RED180',
    'OPEN',
    'CLOSE'
)
```

```
class DataContainer(models.Model):
    fileFormat = models.CharField(
        max_length=10
    )
    formatIdentifier = models.CharField(
        max_length=20
    )
    formatVersion = models.CharField(
        max_length=20
    )
    url = models.URLField()

class NispRawFrame(ImageSpaceFrame):
    filterWheelPosition = models.CharField(
        max_length=10,
        choices = [(d,d) for d in NISP_FILTER_WHEEL]
    )

    grismWheelPosition = models.CharField(
        max_length=10,
        choices = [(d,d) for d in NISP_GRISM_WHEEL]
    )
    frameFile = models.OneToOneField(DataContainer,
                                    on_delete=models.CASCADE)
```

- A `models.OneToOneField` is analogous to `models.ForeignKey` with the option `unique=True` but the reverse side of the relation will directly return a single object

DB Schema creation 1/2



- Once we have defined our data model in **imagedb/models.py** we need Django to create the corresponding DB schema
- First let's check the the project settings includes the imagedb application, i.e. that the file **orm_example/settings.py** contains the the strings highlighted in red in the box on the bottom left
- To do the first migration, i.e. generation of the DB schema, **run the following command**

command `python manage.py makemigrations imagedb`

output

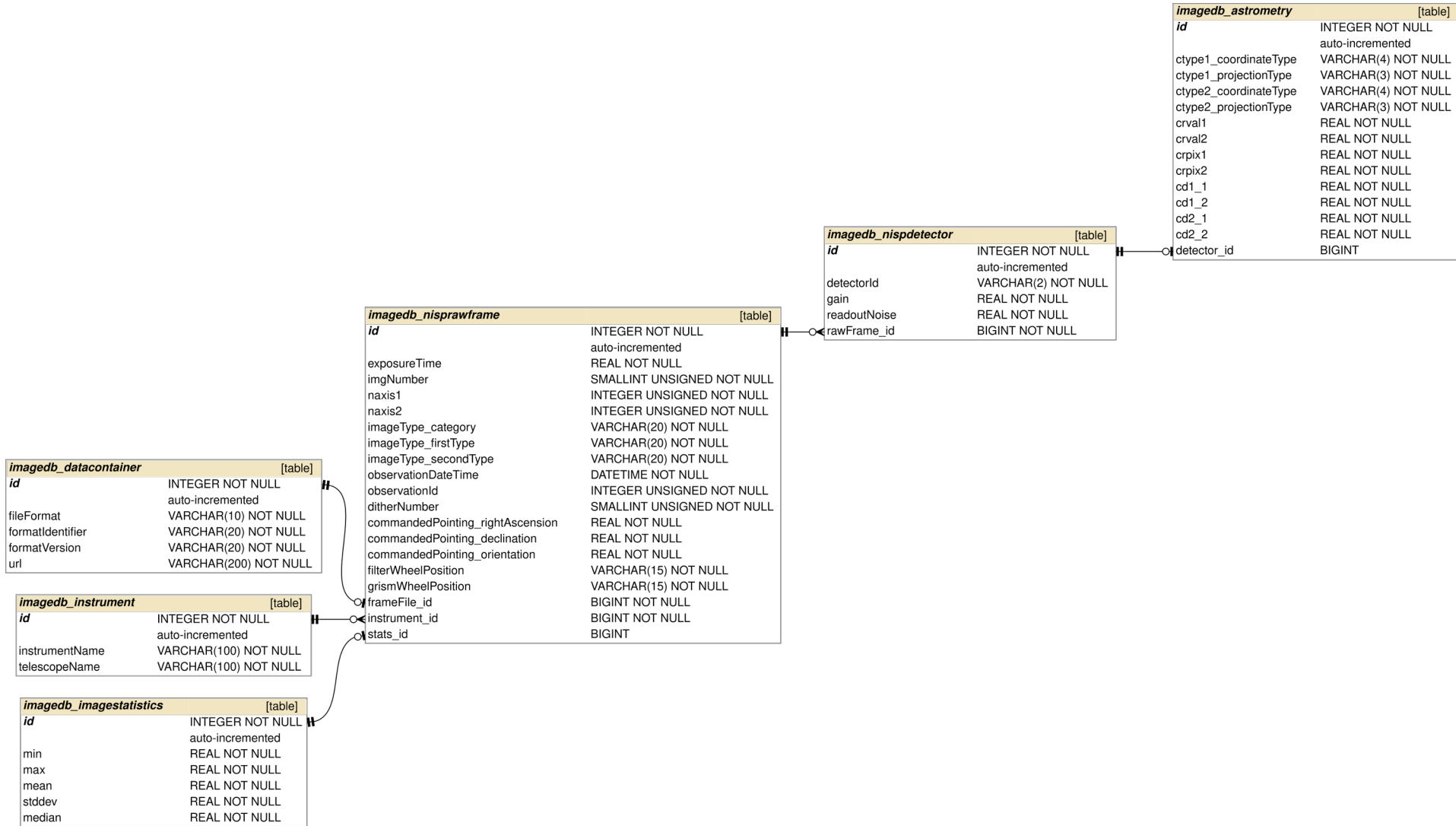
```
Migrations for 'imagedb':
  imagedb/migrations/0001_initial.py
    - Create model DataContainer
    - Create model ImageStatistics
    - Create model Instrument
    - Create model NispRawFrame
    - Create model NispDetector
    - Create model Astrometry
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_extensions',
    'imagedb',
    'rest_framework',
    'django_ufilter',
]
```

Then run the command

```
python manage.py migrate
```


DB Schema creation 2/2



- We can now open a python shell and interact with the data model API

```
python manage.py shell
```

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)  
Type 'copyright', 'credits' or 'license' for more information  
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]: from imagedb.models import Instrument
```

```
In [2]: instrument = Instrument(telescopeName='Euclid', instrumentName='VIS')
```

```
In [3]: instrument.save()
```

```
In [4]: quit()
```

- We can pass a Python script to insert data

```
python manage.py shell < data_ingestion.py
```

```
2025-06-21T17:14:03.000001  
2025-06-21T12:20:43.000001  
2025-06-22T17:42:53.000001  
...
```

Django ORM and Jupyter notebook



- For didactic purpose, we can use a Django extension to start a Jupyter notebook. The `orm_example` project example already provides one notebook. To use it, issue the following command:

```
env DJANGO_ALLOW_ASYNC_UNSAFE=true python manage.py shell_plus --notebook
```

a browser page will be opened. In this page, select the file **`imagedb_objects.ipynb`** and execute each cell.

The screenshot shows the Jupyter web interface in a browser window. The address bar shows `localhost:8888/tree`. The interface includes a 'Quit' and 'Logout' button in the top right. Below the navigation tabs (Files, Running, Clusters), there are 'Upload', 'New', and refresh buttons. A table lists the files and folders in the current directory:

	Name	Last Modified	File size
<input type="checkbox"/>	/		
<input type="checkbox"/>	euclid_example	21 hours ago	
<input type="checkbox"/>	imagedb	19 minutes ago	
<input type="checkbox"/>	resources	a day ago	
<input type="checkbox"/>	imagedb_objects.ipynb	a day ago	11.2 kB
<input type="checkbox"/>	data_ingestion.py	a day ago	3.27 kB
<input type="checkbox"/>	db.sqlite3	5 minutes ago	492 kB
<input type="checkbox"/>	manage.py	a day ago	546 B
<input type="checkbox"/>	README.md	a day ago	790 B

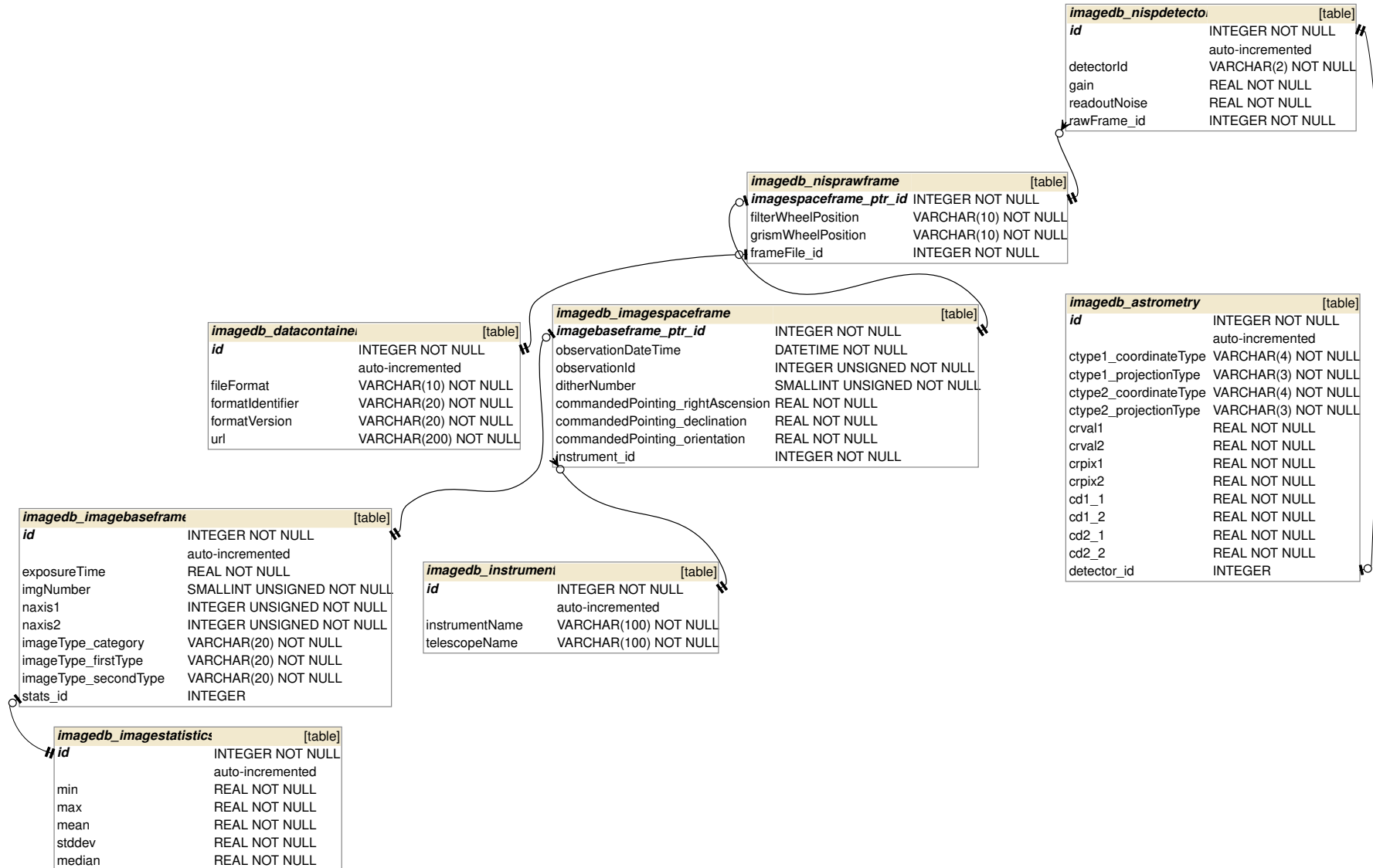
Multi-table inheritance 1/3



- With django model abstract base classes, we cannot define foreign keys referencing such base class (since no table is created for abstract classes)
- A solution is the **Multi-table inheritance** of Django models. In this case the abstraction is removed from such base classes and each class in the inheritance hierarchy will have a corresponding table in the DB schema.
- To obtain a multi-table inheritance version of the previous data model, remove the statements

```
class ImageBaseFrame(models.Model):  
    ...  
class Meta:  
    abstract = True  
  
class ImageSpaceFrame(ImageBaseFrame):  
    ...  
class Meta:  
    abstract = True
```

Multi-table inheritance 2/3



Multi-table inheritance 3/3



- Each model corresponds to its own database table and can be queried and created individually
- The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created **OneToOneField**)
- With the multi-table inheritance, all fields of **ImageBaseFrame** will still be available also in **ImageSpaceFrame** and **NispRawFrame**
- If we have an ImageBaseFrame instance that is also an ImageSpaceFrame instance, we can get from ImageBaseFrame object to ImageSpaceFrame object by using the lower-case version of the model name

```
from imagedb.models import ImageBaseFrame
```

```
obj = ImageBaseFrame.objects.get(pk=2)  
obj.imagespaceframe.nisprawframe
```

```
<NispRawFrame: NispRawFrame object (2)>
```

Serializing Django objects



- Django's serialization framework provides a mechanism for “translating” Django models into other formats.
- Usually these other formats will be text-based and used for sending Django data over a wire, but it's possible for a serializer to handle any format (text-based or not).
- Django supports a number of serialization formats, including XML and JSON.

```
from django.core import serializers

serializers.serialize('json', NispRawFrame.objects.filter(observationId=53877,
                                                         filterWheelPosition='Y').order_by('ditherNumber'))
```

- The Django serialize function requires, as one of the inputs, a **QuerySet**
- However, the **Django REST framework**, external to the Django framework, provides a more flexible serialization mechanism

The Django REST serializers



- In particular, the Django REST framework provides a **ModelSerializer** class which can be a useful shortcut for creating serializers that deal with model instances and querysets
- See 'imagedb/serializers.py' to check some examples

```
from rest_framework import serializers
from composite_field.rest_framework_support import CompositeFieldSerializer

...

class NispRawFrameSerializer(serializers.ModelSerializer):
    detectors = NispDetectorSerializer(many = True, read_only = True)
    commandedPointing = CompositeFieldSerializer()
    imageType = CompositeFieldSerializer()

    class Meta:
        model = NispRawFrame
        exclude = [f.name for g in NispRawFrame._meta.get_fields()
                  if hasattr(g, 'subfields')
                  for f in g.subfields.values()]
        depth = 2
```


The Django REST framework



- We need an Application Programming Interface (API) that let us perform CRUD operations on the database without directly connecting to the database
- A REST (Representational State Transfer) API provides such operations through HTTP methods:
 - GET, to request to a server a specific dataset
 - POST, to create a new data object in the database
 - PUT, to update an existing object in the database or create it if it does not exist
 - DELETE, to request the removal of a given data object
- Such methods can be applied to a specific set of endpoints (URLs) provided by our API
- The Django REST framework provides software tools to build a REST API on top of our models

Django REST framework ViewSets



- The actions provided by the **ModelViewSet** class are `.list()`, `.retrieve()`, `.create()`, `.update()`, `.partial_update()`, and `.destroy()` of instances of a specific model we have defined
- The **ReadOnlyModelViewSet** only provides the 'read-only' actions, `.list()` and `.retrieve()`
 - In practice it returns a list of instances of a specific model or it retrieves a single instance by its primary key value
- In our `orm_example` projects, we have few examples in **`imagedb/views.py`**

```
from rest_framework import viewsets
from imagedb.serializers import NispRawFrameSerializer

class NispRawFrameViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = NispRawFrame.objects.all()
    serializer_class = NispRawFrameSerializer
```

- More advanced filtering capabilities can be added with additional parameters: <https://www.django-rest-framework.org/api-guide/filtering/>

- Once we have defined viewsets on our models, we have to create endpoints (urls) to access those views
- The Django REST framework provides the so called **routers**, which generate automatically url patterns based on the views we have defined
- An example is found in **imagedb/urls.py**

```
from django.urls import re_path, include
from rest_framework.routers import DefaultRouter

from imagedb import views

router = DefaultRouter()
router.register(r'nisprowframes', views.NispRawFrameViewSet)

urlpatterns = [
    re_path(r'^$', include(router.urls))
]
```

will generate automatically the following url patterns:

[/nisprowframes/](#) : it will return, in json format, all the NispRawFrame objects in the database

[/nisprowframes/\[pk\]/](#) : it will return only the NispRawFrame object with primary key pk

Starting the Django development server



- In order to test the REST API, you can start the Django server with the following command

```
python manage.py runserver
```

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

```
System check identified no issues (0 silenced).
```

```
October 15, 2018 - 21:30:57
```

```
Django version 2.1.1, using settings
```

```
'orm_example.settings'
```

```
Starting development server at
```

```
http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

- Now with the browser you can open the following link:
<http://127.0.0.1:8000/imagedb/nisprawframes/1/>

The browsable REST API



← → ↻ ⓘ 127.0.0.1:8000/imagedb/nisprawframes/1/ 🔍 ☆ 🖨️ ↺ 🌐 📱 🗂️ 🔄 🏠 👤 ⋮

Django REST framework

[Api Root](#) / [Nisp Raw Frame List](#) / Nisp Raw Frame Instance

Nisp Raw Frame Instance

[OPTIONS](#) [GET](#) ▾

GET /imagedb/nisprawframes/1/

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "id": 1,
  "detectors": [
    {
      "id": 1,
      "astrometry": {
        "id": 1,
        "ctype1": {
          "coordinateType": "RA",
          "projectionType": "TAN"
        },
        "ctype2": {
          "coordinateType": "DEC",
```

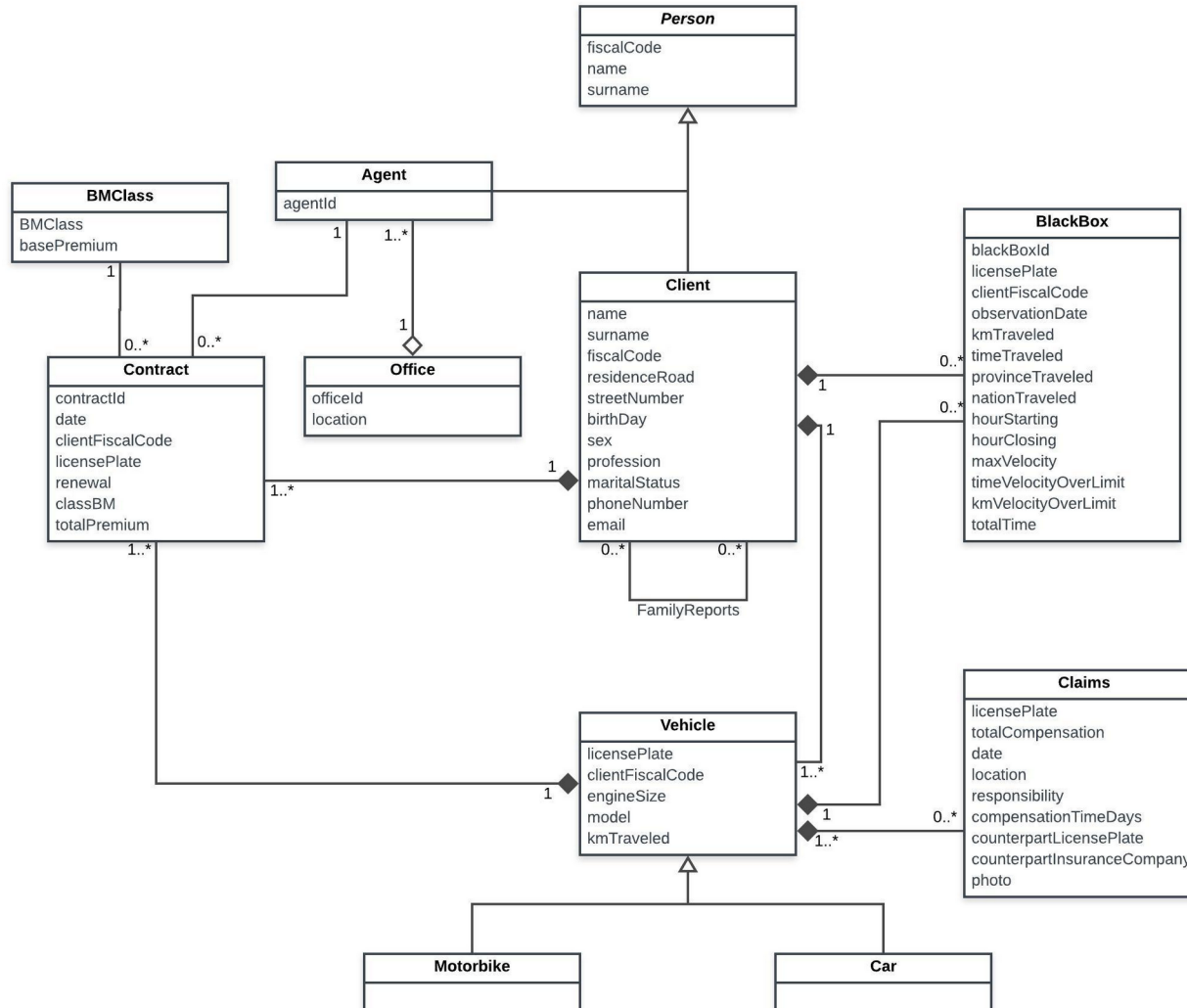
More advanced filtering criteria



- In order to use more advanced filtering criteria through the REST API, rather than just the primary key, in the `orm_example` project we have added the **django-ufilter** (<https://github.com/Qu4tro/django-ufilter/>)
- With this filter, we can specify filtering condition directly in the url, e.g. :

```
http://127.0.0.1:8000/imagedb/nisprawframes/?observationId__in=53877,54349&filterWheelPosition=Y
```

Data Model for Insurance Company



Credit to Andrea Pesce

Identifier for Contracts

Element of interest	Value
name	First three letters
surname	First three letters
date	Day, Month, Year (e.g.: 130394 for 13 March 1994)
renewal number	0 (for first contract),1,2,3,...
province	ISO Code
uniqueness	Random character

Identifier for FamilyReports

Element of interest	Value
first relative	fiscalCode
second relative	fiscalCode

ORM project example



- The entire example can be retrieved at the following link:

https://www.ict.inaf.it/gitlab/odmc/orm_example

- You can clone the project with the git version control system, i.e. with the command:

```
git clone https://www.ict.inaf.it/gitlab/odmc/orm_example.git
cd orm_example/django_example_insurance
```

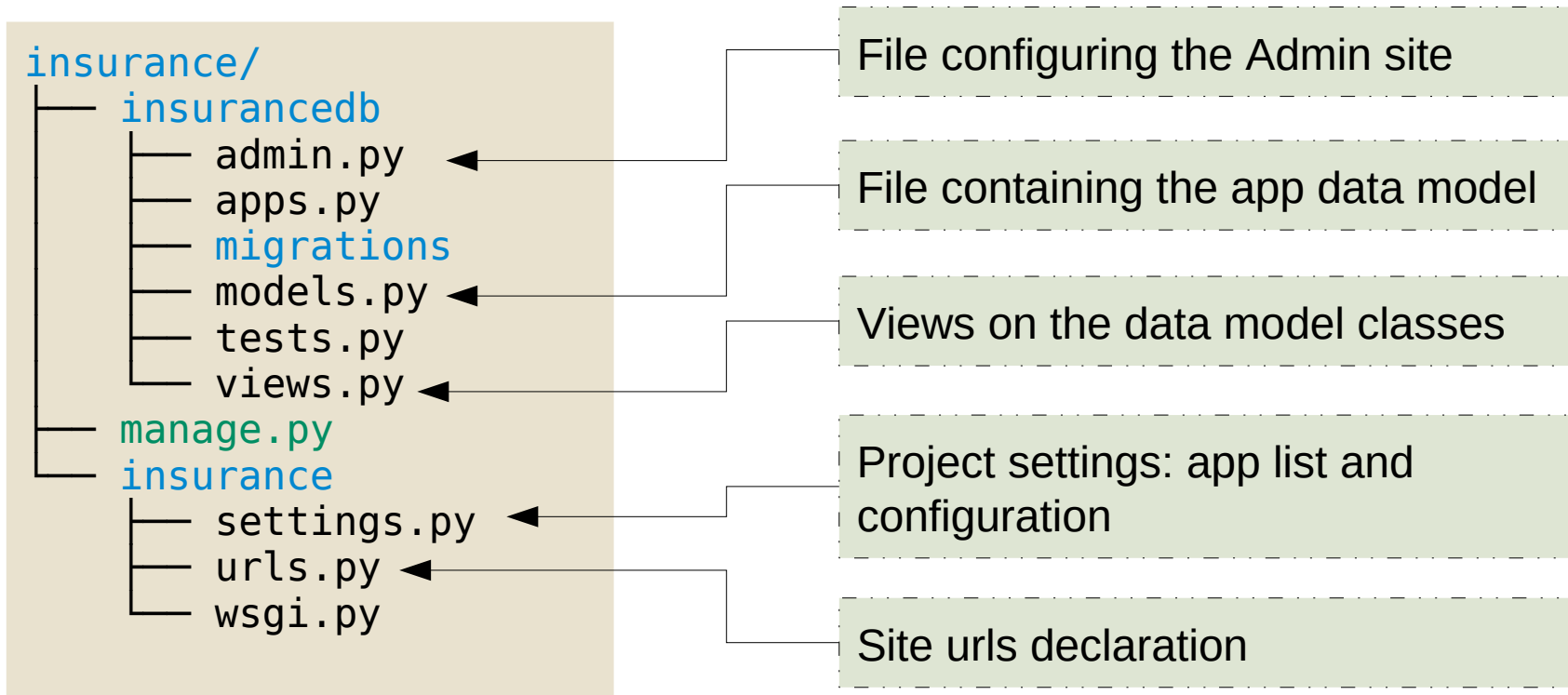
- Create the Django project from scratch using the following commands

```
django-admin startproject insurance
cd insurance
python manage.py startapp insurancedb
```

which creates a project folder, named **insurance**, with additional files and then an application, named **insurancedb**, inside the project.

It automatically creates skeleton files needed by a Django project and application

Project structure



- For `admin.py`, `models.py`, `urls.py` and `views.py` files we are going to use the ones in the git repository
- We must edit the `settings.py`

DB Schema creation



- Once we have defined our data model in **insurancedb/models.py** we need Django to create the corresponding DB schema
- First let's check the the project settings includes the insurancedb application, i.e. that the file **insurance/settings.py** contains the strings highlighted in red in the box on the bottom left
- To do the first migration, i.e. generation of the DB schema, **run the following command**

command `python manage.py makemigrations insurancedb`

output

```
Migrations for 'insurancedb':
  insurancedb/migrations/0001_initial.py
    - Create model BMClass
    - Create model Client
    - Create model Office
    - Create model Vehicle
    - Create model Contract
    - Create model Claims
    - Create model BlackBox
    - Create model Agent
    - Create model FamilyReports
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_extensions',
    'insurancedb',
]
```

Then run the command

```
python manage.py migrate
```

- We can now open a python shell and interact with the data model API

```
python manage.py shell
```

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from insurancedb.models import BMClass

In [2]: bonus = BMClass(BMClass=1, basePremium=100.00)

In [3]: bonus.save()

In [4]: quit()
```

- You can pass a Python script to insert data

```
python manage.py shell < ../insert.py
```

Django urls.py and views.py



- A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations
- To design URLs for an app, you create a Python module informally called a URLconf (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views)
- A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This response can be:
 - HTML contents
 - A redirect
 - A 404 error
 - An XML document
 - An image
 - ...

```
from django.http import HttpResponseRedirect
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponseRedirect(html)
```

Django admin.py



- Django provides an automatic admin interface
- It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site
- You can customize the admin interface editing the admin.py
- Setup an admin user

```
python manage.py createsuperuser
```

- Run the Django web server

```
python manage.py runserver
```

- Access to <http://127.0.0.1:8000/>



ADDITIONAL MATERIAL

SQLAlchemy (1)



- The SQLAlchemy SQL Toolkit and Object Relational Mapper is a comprehensive set of tools for working with databases and Python
- It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access
- SQLAlchemy has dialects for many popular database systems including Firebird, Informix, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, or Sybase
- The SQLAlchemy has four ways of working with database data:
 - Raw SQL
 - SQL Expression Language
 - Schema Definition Language
 - ORM

- SQLAlchemy ORM consists of several components
 - **Engine**
 - It manages the connection with the database
 - It is created using the `create_engine()` function
 - Declarative **Base** class
 - It maintains a catalog of classes and tables
 - It is created using the `declarative_base()` function and is bound to the engine
 - **Session** class
 - It is a container for all conversations with the database
 - It is created using the `sessionmaker()` function and is bound to the engine
- <https://docs.sqlalchemy.org/en/14/orm/tutorial.html>

Prerequisites



- Download and install the Python Anaconda (or Miniconda) Distribution, with Python version 3.x:

<https://www.anaconda.com/download>

- Then you need to install some additional python packages for the following exercise/hands-on:

- To install the Django framework use the following command line:

```
conda create -n orm_sqlalchemy sqlalchemy  
conda activate orm_sqlalchemy
```

- Clone the GIT repository and enter the directory of SQLAlchemy examples

```
git clone https://www.ict.inaf.it/gitlab/odmc/orm_example.git  
cd orm_example/sqlalchemy_example
```

ORM with SQLAlchemy: Example 1

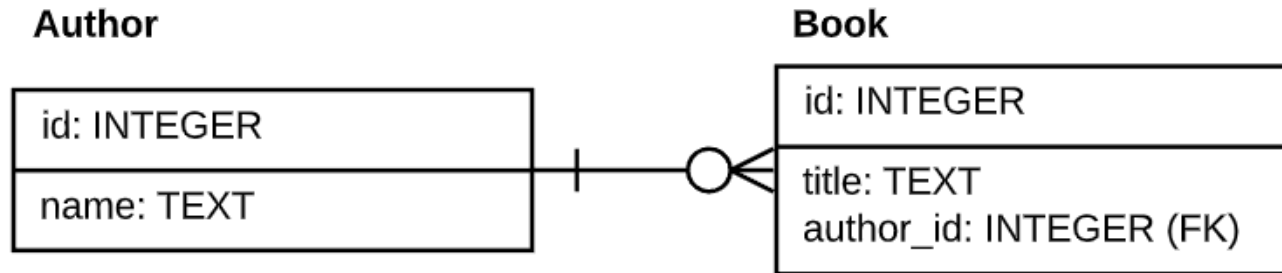


Car

id: INTEGER
name: TEXT price: INTEGER

- Engines <https://docs.sqlalchemy.org/en/14/core/engines.html>
- Declarative Base
<https://docs.sqlalchemy.org/en/14/orm/extensions/declarative/>
- Session <https://docs.sqlalchemy.org/en/14/orm/session.html>
- Query <https://docs.sqlalchemy.org/en/14/orm/query.html>

ORM with SQLAlchemy: Example 2

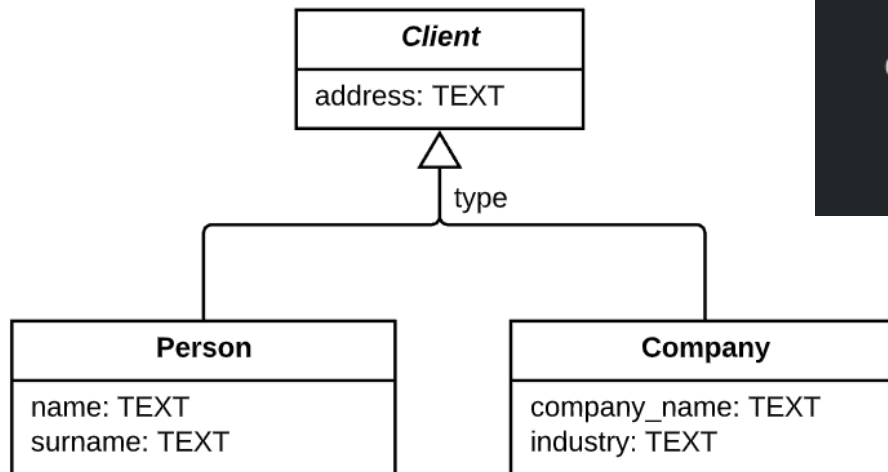


- Foreign keys in SQLite
<https://docs.sqlalchemy.org/en/14/dialects/sqlite.html#foreign-key-support>
- Relationship
https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html

Inheritance in Python



- This is a simple example of inheritance in UML and how can be implemented in Python



UML

```
class Client(object):
    """docstring for Client"""

    def __init__(self, address):
        super(Client, self).__init__()
        self.address = address

class Person(Client):
    """docstring for Person"""

    def __init__(self, name, surname, address):
        #super(Person, self).__init__(address)
        self.name = name
        self.surname = surname

class Company(Client):
    """docstring for Company"""

    def __init__(self, company_name, industry, address):
        super(Company, self).__init__(address)
        self.company_name = company_name
        self.industry = industry
```

Inheritance in a Relational Database



Client

id: INTEGER
address: TEXT type: CHAR name: TEXT surname: TEXT company_name: TEXT industry: TEXT

IE

Single table inheritance

- Unique ID
- No JOIN necessary
- Many NULL attributes

Concrete table inheritance

- Not unique ID
- No JOIN necessary
- No NULL attributes

Person

id: INTEGER
address: TEXT name: TEXT surname: TEXT

Company

id: INTEGER
address: TEXT company_name: TEXT industry: TEXT

IE

Client

id: INTEGER
address: TEXT type: CHAR



type

Person

id: INTEGER (FK)
name: TEXT surname: TEXT

IE

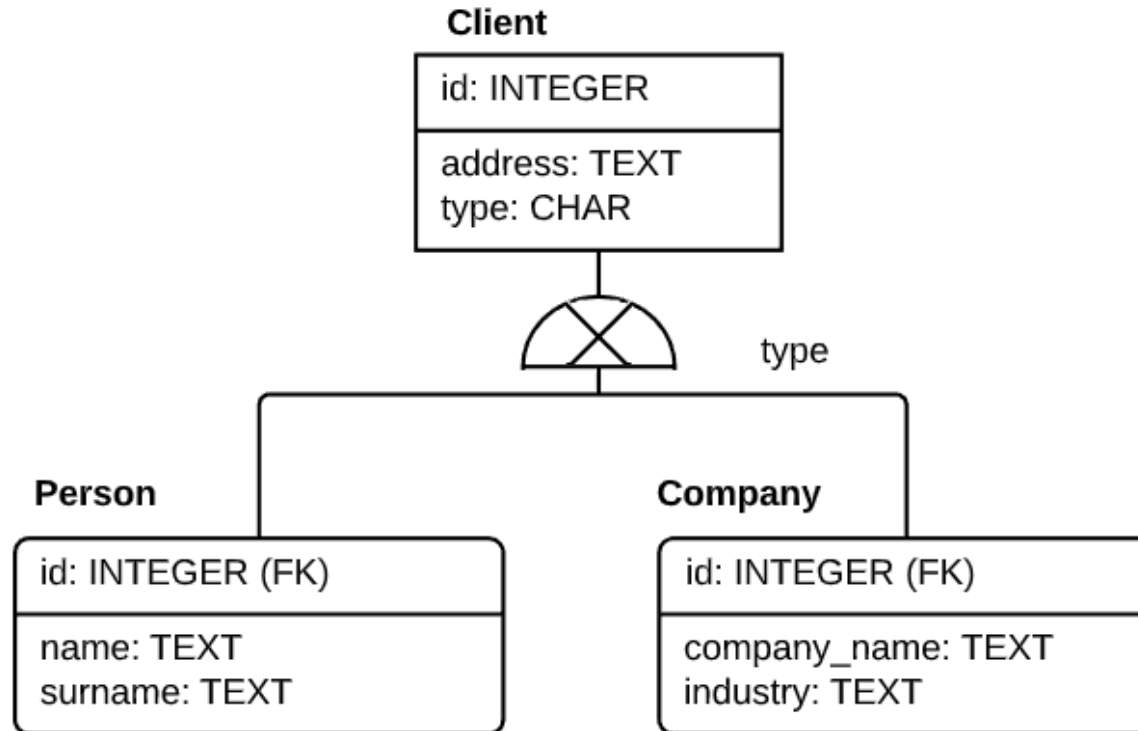
Company

id: INTEGER (FK)
company_name: TEXT industry: TEXT

Joined table inheritance

- Unique ID
- JOIN necessary
- No NULL attributes

ORM with SQLAlchemy: Example 3



- Inheritance <https://docs.sqlalchemy.org/en/14/orm/inheritance.html>