CRUD on Django Model Objects

Skills Network

Estimated time needed: 30 minutes

In this lab, you will practice creating Django Models with relationships, saving objects into databases, querying, updating, and deleting objects.

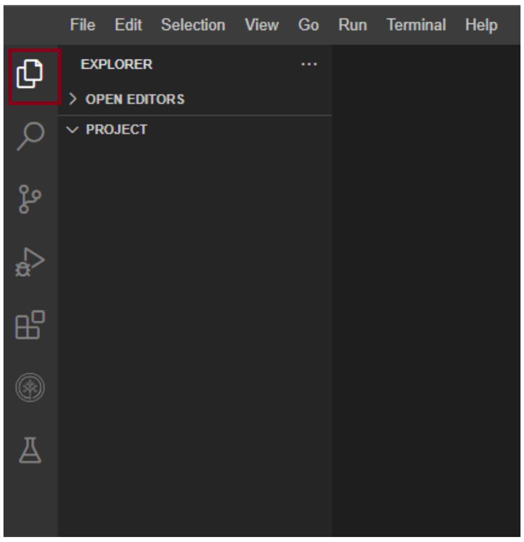
Learning Objectives

- Create Django Models with One-To-One, One-To-Many, and Many-To-Many relationships
- Query model objects with filters
- Delete and update objects

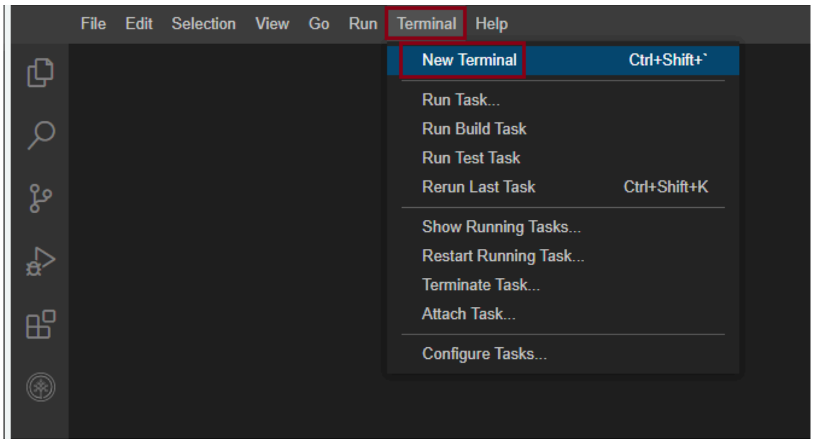
Working with files in Cloud IDE

If you are new to Cloud IDE, this section will show you how to create and edit files, which are part of your project, in Cloud IDE.

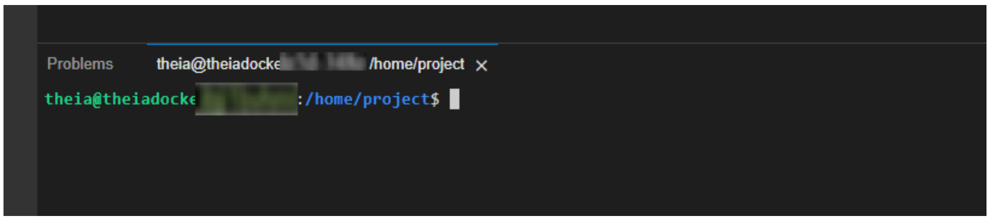
To view your files and directories inside Cloud IDE, click on this files icon to reveal it.



Click on `New`, and then `New Terminal`.



This will open a new terminal where you can run your commands.



```
theia@theiadocke[redacted]:/home/project$
```

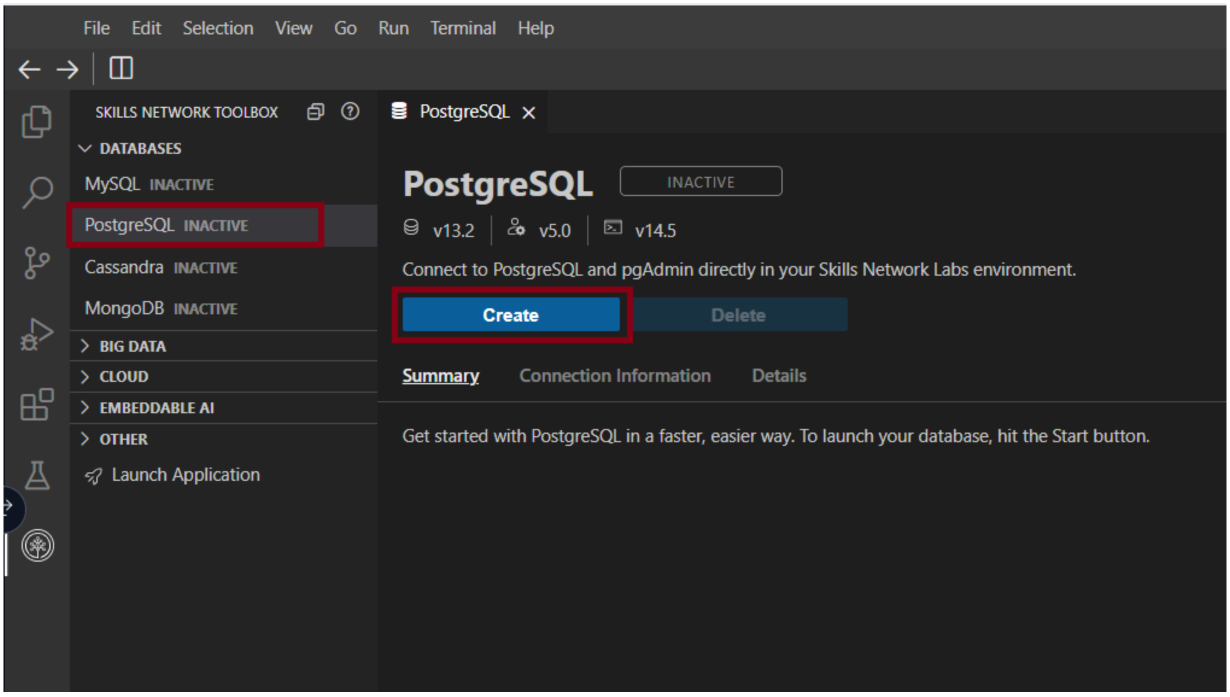
Concepts covered in the lab

1. `CRUD operations`: Create, Read, Update, and Delete operations
2. `QuerySet`: Represents a collection of records in a database, and is required to read objects using Django Model API.
3. `all()` method: Returns a QuerySet of all the objects in the database.

4. `filter()` method: Returns only the rows that match the search term. It can have lookup parameters such as **greater than**, **less than**, **contains**, or **is null**.

Start PostgreSQL in Theia

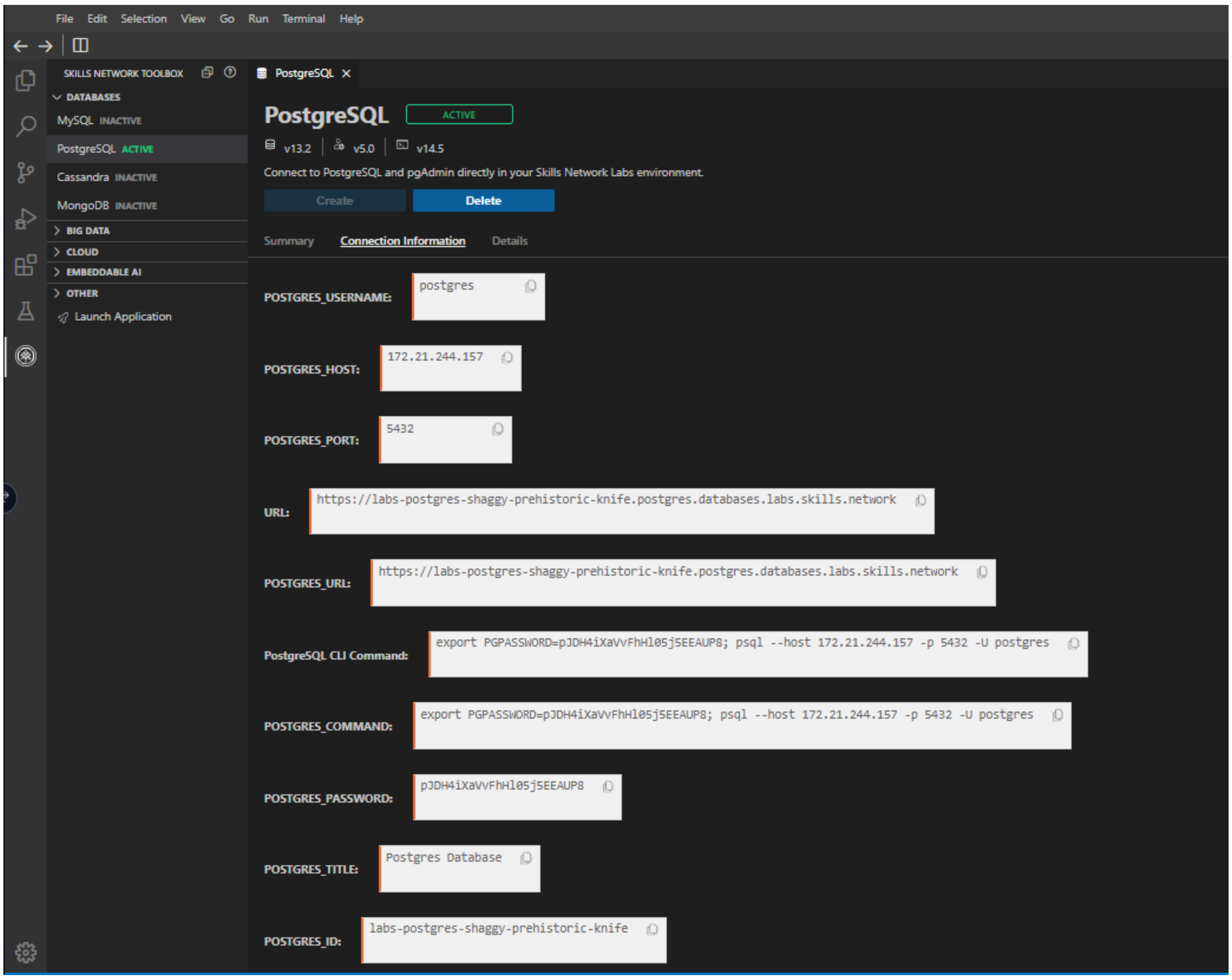
- Start PostgreSQL from UI by finding the SkillsNetwork icon on the left menu bar and selecting PostgreSQL from the DATABASES menu item:



Connect to PostgreSQL and pgAdmin directly in your Skills Network Labs environment.

Get started with PostgreSQL in a faster, easier way. To launch your database, hit the Start button.

- Once the PostgreSQL has been started, you can check the server connection information from the UI. Please markdown the connection information such as generated `username`, `password`, and `host`, etc, which will be used to configure Django app to connect to this database.



Connect to PostgreSQL and pgAdmin directly in your Skills Network Labs environment.

POSTGRES_USERNAME: `postgres`

POSTGRES_HOST: `172.21.244.157`

POSTGRES_PORT: `5432`

URL: `https://labs-postgres-shaggy-prehistoric-knife.postgres.databases.labs.skills.network`

POSTGRES_URL: `https://labs-postgres-shaggy-prehistoric-knife.postgres.databases.labs.skills.network`

PostgreSQL CLI Command: `export PGPASSWORD=pJDH4iXaVvFhHl05j5EEAUP8; psql --host 172.21.244.157 -p 5432 -U postgres`

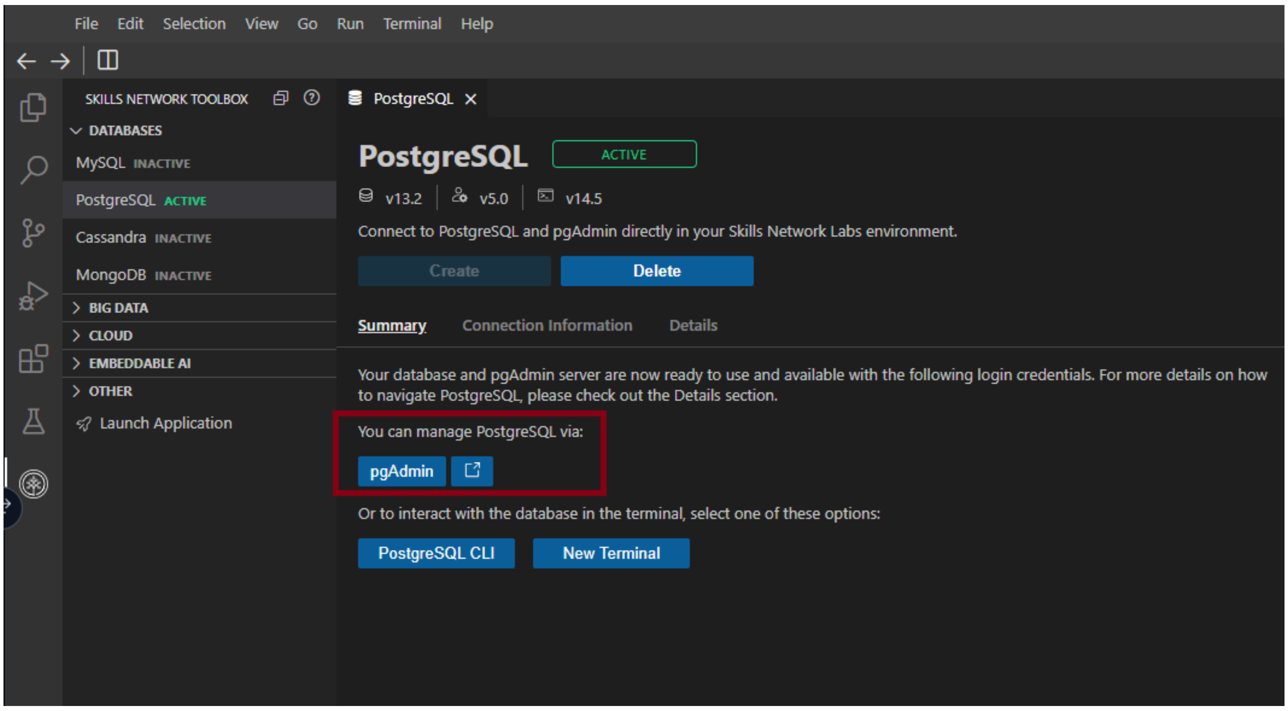
POSTGRES_COMMAND: `export PGPASSWORD=pJDH4iXaVvFhHl05j5EEAUP8; psql --host 172.21.244.157 -p 5432 -U postgres`

POSTGRES_PASSWORD: `pJDH4iXaVvFhHl05j5EEAUP8`

POSTGRES_TITLE: `Postgres Database`

POSTGRES_ID: `labs-postgres-shaggy-prehistoric-knife`

- You will also see a `pgAdmin` instance installed and started.



PostgreSQL ACTIVE

Connect to PostgreSQL and pgAdmin directly in your Skills Network Labs environment.

Your database and pgAdmin server are now ready to use and available with the following login credentials. For more details on how to navigate PostgreSQL, please check out the Details section.

You can manage PostgreSQL via:

Or to interact with the database in the terminal, select one of these options:

Import a standalone Django ORM project template

Before starting the lab, make sure your current Theia directory is `/home/project`.

First, we need to create a virtual environment and install Django and psycopg related packages.

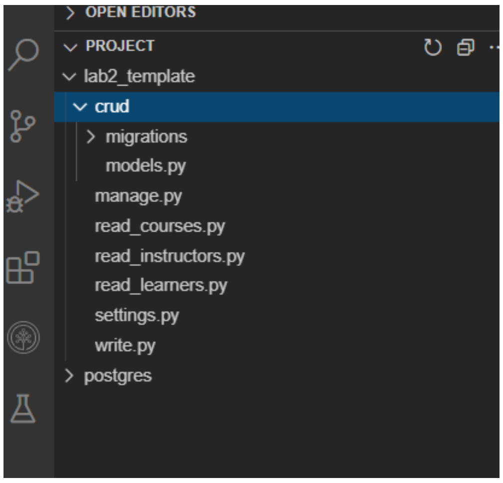
- If the terminal was not open, go to `Terminal > New Terminal` and run:

```
pip install --upgrade distro-info
pip3 install --upgrade pip==23.2.1
pip install virtualenv
virtualenv djangoenv
source djangoenv/bin/activate
pip install Django==4.2 psycopg2-binary
```

- Run the following command-lines to download a code template for this lab

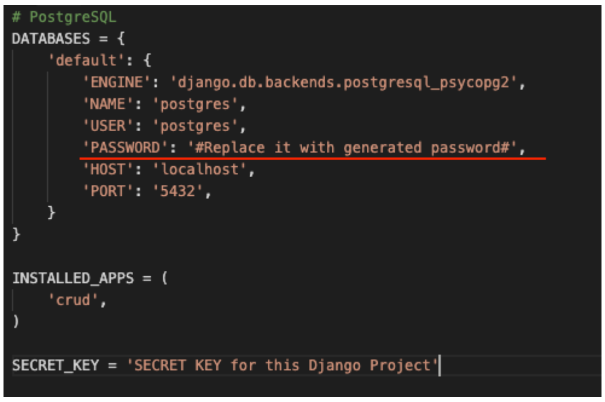
```
wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-CD0251EN-SkillsNetwork/labs/m3_django_orm/lab2_template.zip"
unzip lab2_template.zip
rm lab2_template.zip
```

Your Django project should look like the following:



- Open `settings.py` and find the `DATABASES` section. Replace the value of the `PASSWORD` to be the generated `PostgreSQL` password.

Your `settings.py` file now should look like the following:



```
# PostgreSQL
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': '#Replace it with generated password#',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}

INSTALLED_APPS = (
    'crud',
)

SECRET_KEY = 'SECRET KEY for this Django Project'
```

Create Models For an Online Course App

Next, you can start to create models for an online course app.

- Open `crud/models.py` and append first `User` model

```
# User model
class User(models.Model):
    first_name = models.CharField(null=False, max_length=30, default='john')
    last_name = models.CharField(null=False, max_length=30, default='doe')
    dob = models.DateField(null=True)

    # Create a toString method for object string representation
    def __str__(self):
        return self.first_name + " " + self.last_name
```

The `User` model contains common information about a user such as `first_name`, `last_name` as `CharField` and `dob` as `DateField`.

In addition to that, we override the `__str__(self):` method to create a string representation of a user object. This is convenient if you want to print a user object.

Also feel free to add as many primitive fields as you like to the `User` model such as `email` or `location`. You could find more details about model field definitions here: Django Model Fields

Next, let's add an `Instructor` model inherited from `User` model and make it as `One-To-One` relationship. `Instructor` is an extension of `User` which adds some more instructor specific fields such as `full_time` and `total_learners`.

- Append an `Instructor` model to `models.py`

```
# Instructor model
class Instructor(User):
    full_time = models.BooleanField(default=True)
    total_learners = models.IntegerField()

    # Create a toString method for object string representation
    def __str__(self):
        return "First name: " + self.first_name + ", " + \
               "Last name: " + self.last_name + ", " + \
               "Is full time: " + str(self.full_time) + ", " + \
               "Total Learners: " + str(self.total_learners)
```

Then, let's create a `Course` model which has a `Many-To-Many` relationship to `Instructor` model, defined by the reference field `instructors`.

- Append a `Course` model to `models.py`

```
# Course model
class Course(models.Model):
    name = models.CharField(null=False, max_length=100, default='online course')
    description = models.CharField(max_length=500)
    # Many-To-Many relationship with Instructor
    instructors = models.ManyToManyField(Instructor)

    # Create a toString method for object string representation
    def __str__(self):
        return "Name: " + self.name + "," + \
            "Description: " + self.description
```

Here we added a Many-To-Many relationship between `Course` and `Instructor` by creating a `ManyToManyField` field called `instructors`

A course normally contains several lessons thus has a `One-To-Many` relationship to a `Lesson` model, i.e., each course can have zero or many lessons but each lesson only belongs to one course.

- Append a `Lesson` model to `models.py`

```
# Lesson
class Lesson(models.Model):
    title = models.CharField(max_length=200, default="title")
    course = models.ForeignKey(Course, null=True, on_delete=models.CASCADE)
    content = models.TextField()
```

Coding Practice: Add a Learner Model

Complete the following code snippet to add a `Learner` model inherited from `User` with some learner related fields:

1. You could look at the comments with `<HINT>` for the missing parts.
2. You need to define `Learner` model before `Course` model in `models.py` so that the `Course` model knows the existence of `Learner`
3. Remember to save the updated files to make the changes effective.

```
# Learner model
class Learner(##<HINT> add a user parent model):
    STUDENT = 'student'
    DEVELOPER = 'developer'
    DATA_SCIENTIST = 'data_scientist'
    DATABASE_ADMIN = 'dba'
    OCCUPATION_CHOICES = [
        (STUDENT, 'Student'),
        (DEVELOPER, 'Developer'),
        (DATA_SCIENTIST, 'Data Scientist'),
        (DATABASE_ADMIN, 'Database Admin')
    ]
    occupation = models.CharField(
        null=False,
        max_length=20,
        choices=OCCUPATION_CHOICES,
        default=STUDENT
    )
    social_link = models.URLField(max_length=200)

    ##<HINT> Create a __str__ method returning a string presentation
    def __str__(self):
        ...
```

▼ Click here to see solution

```
# Learner model
class Learner(User):
    STUDENT = 'student'
    DEVELOPER = 'developer'
    DATA_SCIENTIST = 'data_scientist'
    DATABASE_ADMIN = 'dba'
    OCCUPATION_CHOICES = [
        (STUDENT, 'Student'),
        (DEVELOPER, 'Developer'),
        (DATA_SCIENTIST, 'Data Scientist'),
        (DATABASE_ADMIN, 'Database Admin')
    ]
    # Occupation Char field with defined enumeration choices
    occupation = models.CharField(
        null=False,
        max_length=20,
        choices=OCCUPATION_CHOICES,
        default=STUDENT
    )
    # Social link URL field
    social_link = models.URLField(max_length=200)

    # Create a toString method for object string representation
    def __str__(self):
        return "First name: " + self.first_name + ", " + \
               "Last name: " + self.last_name + ", " \
               "Date of Birth: " + str(self.dob) + ", " + \
               "Occupation: " + self.occupation + ", " + \
               "Social Link: " + self.social_link
```

Note for the `occupation` field, we added an enumerate choices for limiting the values of occupations

Coding Practice: Add an Enrollment Model

Append the following `Enrollment` class.
Update `Course` model to add a `Many-To-Many` relationship with `Learner` model via the `Enrollment` class.

```
# Enrollment model as a lookup table with additional enrollment info
class Enrollment(models.Model):
    AUDIT = 'audit'
    HONOR = 'honor'
    COURSE_MODES = [
        (AUDIT, 'Audit'),
        (HONOR, 'Honor'),
    ]
    # Add a learner foreign key
    learner = models.ForeignKey(Learner, on_delete=models.CASCADE)
    # Add a course foreign key
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    # Enrollment date
    date_enrolled = models.DateField(default=now)
    # Enrollment mode
    mode = models.CharField(max_length=5, choices=COURSE_MODES, default=AUDIT)
```

▼ Click here to see solution

Add a `Many-To-Many` relationship between `Course` and `Learner` by creating a `ManyToManyField` field called `learners` via `Enrollment`.

```
# Course model
class Course(models.Model):
    name = models.CharField(null=False, max_length=100, default='online course')
    description = models.CharField(max_length=500)
    # Many-To-Many relationship with Instructors
    instructors = models.ManyToManyField(Instructor)
    # Many-To-Many relationship with Learner via Enrollment relationship
    learners = models.ManyToManyField(Learner, through='Enrollment')
    def __str__(self):
        return "Name: " + self.name + "," + \
                "Description: " + self.description
    def __str__(self):
        return "Name: " + self.name + "," + \
                "Description: " + self.description
```

Similar to `instructors` reference field, we can add a `learners` reference field in `Course`.

Migrate Models

Now, you have defined:

1. `User` and `Instructor` models with One-To-One relationship
2. `Course` and `Lessons` models with One-To-Many relationship
3. `Course` and `Instructor` with Many-To-Many relationship.

Let's run migrations for the `crud` app to create those tables in our `PostgreSQL` database.

- If your current working directory is not `/home/project/lab2_template`, `cd` to the project folder

  ```
  cd lab2_template
  ```

- Then generate migration scripts for app `crud`

  ```
  python3 manage.py makemigrations crud
  ```

and you should see Django is about to create the following tables.

```
Migrations for 'crud':
  crud/migrations/0001_initial.py
    - Create model Course
    - Create model User
    - Create model Instructor
    - Create model Learner
    - Create model Lesson
    - Create model Enrollment
    - Add field instructors to course
    - Add field learners to course
```

- Run the migrations.

  ```
  python3 manage.py migrate
  ```

and you should see migration script `crud.0001_initial` was executed.

```
Operations to perform:
  Apply all migrations: crud
Running migrations:
  Applying crud.0001_initial... OK
```

Create and Delete Objects

At this point, you have defined all models for this lab.

Let's try to perform some create and delete operations on those models.

- Open `write.py` and append a `write_instructors()` method to save some instructors objects.

  ```
  def write_instructors():
      # Add instructors
      # Create a user
  ```

```
    user_john = User(first_name='John', last_name='Doe', dob=date(1962, 7, 16))
    user_john.save()
    instructor_john = Instructor(full_time=True, total_learners=30050)
    # Update the user reference of instructor_john to be user_john
    instructor_john.user = user_john
    instructor_john.save()

    instructor_yan = Instructor(first_name='Yan', last_name='Luo', dob=date(1962, 7, 16), full_time=True, total_learners=30050)
    instructor_yan.save()
    instructor_joy = Instructor(first_name='Joy', last_name='Li', dob=date(1992, 1, 2), full_time=False, total_learners=10040)
    instructor_joy.save()
    instructor_peter = Instructor(first_name='Peter', last_name='Chen', dob=date(1982, 5, 2), full_time=True, total_learners=2002)
    instructor_peter.save()
    print("Instructor objects all saved... ")
```

For the `instructor_john`, we first create his parent class model `user` and update `instructor_john.user` to be `user_john`.

For other instructors, Django will automatically assign values of `first_name`, `last_name`, `dob` to their parent user objects.

- Append a `write_courses()` method to add some course objects.

```
def write_courses():
    # Add Courses
    course_cloud_app = Course(name="Cloud Application Development with Database",
                              description="Develop and deploy application on cloud")
    course_cloud_app.save()
    course_python = Course(name="Introduction to Python",
                           description="Learn core concepts of Python and obtain hands-on "
                                       "experience via a capstone project")
    course_python.save()
    print("Course objects all saved... ")
```

- Append a `write_lessons()` method to add some lessons

```
def write_lessons():
    # Add lessons
    lession1 = Lesson(title='Lesson 1', content="Object-relational mapping project")
    lession1.save()
    lession2 = Lesson(title='Lesson 2', content="Django full stack project")
    lession2.save()
    print("Lesson objects all saved... ")
```

- To conveniently clean up your database tables, you can add a `clean_data()` method like the following code snippet.
  It uses the model manager `objects` to get all objects first and then delete them from database.

```
def clean_data():
    # Delete all data to start from fresh
    Enrollment.objects.all().delete()
    User.objects.all().delete()
    Learner.objects.all().delete()
    Instructor.objects.all().delete()
    Course.objects.all().delete()
    Lesson.objects.all().delete()
```

Next, let's call those populating methods to actually save the objects

- Append the following methods call to `write.py`

```
# Clean any existing data first
clean_data()
write_courses()
write_instructors()
write_lessons()
```

- At last, run the `write.py` in terminal

  ```
  python3 write.py
  ```

You should see objects all saved messages in the terminal indicating the save operations were done successfully.

```
Course objects all saved...
Instructor objects all saved...
Lesson objects all saved...
```

In the next step, let's try to query those saved objects.

Coding Practice: Create and Save More Learner Objects

Complete and the following code snippet `write_learners()` method by saving more learners objects into database:

1. You could look at the comments with `<HINT>` for the missing parts.

   ```
   def write_learners():
       # Add Learners
       learner_james = Learner(first_name='James', last_name='Smith', dob=date(1982, 7, 16),
                               occupation='data_scientist',
                               social_link='https://www.linkedin.com/james/')
       learner_james.save()
       learner_mary = Learner(first_name='Mary', last_name='Smith', dob=date(1991, 6, 12), occupation='dba',
                              social_link='https://www.facebook.com/mary/')
       learner_mary.save()
       learner_robert = Learner(first_name='Robert', last_name='Lee', dob=date(1999, 1, 2), occupation='student',
                                social_link='https://www.facebook.com/robert/')
       learner_robert.save()
       learner_david = Learner(first_name='David', last_name='Smith', dob=date(1983, 7, 16),
                               occupation='developer',
                               social_link='https://www.linkedin.com/david/')
       learner_david.save()
       learner_john = Learner(first_name='John', last_name='Smith', dob=date(1986, 3, 16),
                              occupation='developer',
                              social_link='https://www.linkedin.com/john/')
       learner_john.save()
       print("Learner objects all saved... ")
       #<HINT> Add more learners objects#
       #...
   ```

2. Add the `write_learners()` method call to `write.py`

   ```
   # Clean any existing data first
   clean_data()
   write_courses()
   write_instructors()
   write_lessons()
   write_learners()
   ```

3. Run the `write.py` file in terminal again

   ```
   python3 write.py
   ```

Query objects

We first read all `courses`.

- Open `read_courses.py` and add the following code snippet:

  ```
  #Find all courses
  courses = Course.objects.all()
  print(courses)
  ```

In the above code snippet, we call the model managers `objects` to return us all the courses.

- Let's run the Python script file to test the result

  ```
  python3 read_courses.py
  ```

You should see a `QuerySet` object containing the two courses we created in the previous step.

```
<QuerySet [
<Course: Name: Cloud Application Development with Database,Description: Develop and deploy application on cloud>,
<Course: Name: Introduction to Python,Description: Learn core concepts of Python and obtain hands-on experience via a capstone project>
]>
```

- Next, let's query instructors with filters to select subsets of instructors meeting following criterions:

1. Find a single instructor with first name `Yan`
2. Try to find a non-existing instructor with first name `Andy`
3. Find all part time instructors
4. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000
5. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000 using multiple parameters

- Open `read_instructor.py`, and add the following code snippets to perform queries:

  ```
  instructor_yan = Instructor.objects.get(first_name="Yan")
  print("1. Find a single instructor with first name `Yan`")
  print(instructor_yan)
  print("\n")
  # Note that there is no instructor with first name `Andy`
  # So the manager will throw an exception
  try:
      instructor_andy = Instructor.objects.get(first_name="Andy")
  except Instructor.DoesNotExist:
      print("2. Try to find a non-existing instructor with first name `Andy`")
      print("Instructor Andy doesn't exist")
  print("\n")
  part_time_instructors = Instructor.objects.filter(full_time=False)
  print("3. Find all part time instructors: ")
  print(part_time_instructors)
  print("\n")
  full_time_instructors = Instructor.objects.exclude(full_time=False).filter(total_learners__gt=30000).\
          filter(first_name__startswith='Y')
  print("4. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000")
  print(full_time_instructors)
  print("\n")
  full_time_instructors = Instructor.objects.filter(full_time=True, total_learners__gt=30000,
                                                    first_name__startswith='Y')
  print("5. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000")
  print(full_time_instructors)
  ```

Review the above code examples to understand how each filter and parameters were made.

- Run `read_instructors.py` in the terminal

  ```
  python3 read_instructors.py
  ```

Query results:

```
1. Find a single instructor with first name `Yan`
First name: Yan, Last name: Luo, Is full time: True, Total Learners: 30050
2. Try to find a non-existing instructor with first name `Andy`
Instructor Andy doesn't exist
3. Find all part time instructors:
<QuerySet [<Instructor: First name: Joy, Last name: Li, Is full time: False, Total Learners: 10040>]>
4. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000
<QuerySet [<Instructor: First name: Yan, Last name: Luo, Is full time: True, Total Learners: 30050>]>
5. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000
<QuerySet [<Instructor: First name: Yan, Last name: Luo, Is full time: True, Total Learners: 30050>]>
```

Coding practice: Query Learners with Filters

Open `read_learners.py`, complete and append the code snippet to query subset learners based on the following criterions:

1. Find learners with last name `Smith`
2. Find two youngest learners (ordered by `dob`)

```
 # Find students with last name "Smith"
learners_smith = Learner.objects.filter(#<HINT> add last_name check)
print("1. Find learners with last name `Smith`")
print(learners_smith)
print("\n")
# Order by dob descending, and select the first two objects
learners = Learner.objects.order_by(#<HINT> add dob with - as descending )[#<HINT> add index 0:2]
print("2. Find top two youngest learners")
print(learners)
```

▼ Click here to see solution

```
 # Find students with last name "Smith"
learners_smith = Learner.objects.filter(last_name="Smith")
print("1. Find learners with last name `Smith`:")
print(learners_smith)
print("\n")
# Order by dob descending, and select the first two objects
learners = Learner.objects.order_by('-dob')[0:2]
print("2. Find top two youngest learners:")
print(learners)
```

- Run the `read_learners.py` file:

  ```
  python3 read_learners.py
  ```

Query results:

```
1. Find learners with last name `Smith`
<QuerySet [<Learner: First name: James, Last name: Smith, Date of Birth: 1982-07-16, Occupation: data_scientist, Social Link: https://www.linkedin.com/james/>,
<Learner: First name: Mary, Last name: Smith, Date of Birth: 1991-06-12, Occupation: dba, Social Link: https://www.facebook.com/mary/>,
<Learner: First name: David, Last name: Smith, Date of Birth: 1983-07-16, Occupation: developer, Social Link: https://www.linkedin.com/david/>,
<Learner: First name: John, Last name: Smith, Date of Birth: 1986-03-16, Occupation: developer, Social Link: https://www.linkedin.com/john/>]>
```

```
2. Find top two youngest learners
<QuerySet [<Learner: First name: Robert, Last name: Lee, Date of Birth: 1999-01-02, Occupation: student, Social Link: https://www.facebook.com/robert/>,
<Learner: First name: Mary, Last name: Smith, Date of Birth: 1991-06-12, Occupation: dba, Social Link: https://www.facebook.com/mary/>]>
```

Summary

In this lab, you have imported a standalone Django ORM project template. Based on the template, you
learned and practiced creating Django Models with relationships, saving those Django Models objects into databases, performing update and delete operations, and querying them with filters.

Next, you will learn how to access related objects.

Author(s)

Yan Luo