ICTACADEMY

Installing Django and REST Framework

⚠	ℹ	↺	TRY IT
Error/ Warning	Information	Flashback	Class Exercise

AGENDA

1. Understanding the directory structure for a virtual environment
2. Installing Django and Django REST frameworks in an isolated environment
3. Installing HTTPie and Installing the Postman REST client
4. Understanding the table generated by Django
5. Working with the Django shell and diving deeply into serialization and deserialization

1. Understanding the Directory Structure for a Virtual Environment

Understanding the directory structure for a virtual environment

- Virtual environment for Python development can be created in command prompt :

```
python –m venv ./myenv
```

- To activate this :

```
myenv\scripts\activate
```

Here “myenv” virtual environment is created and activated.

Understanding the directory structure for a virtual environment

Directory structure

When you create a virtual environment, it sets up a directory structure with specific subdirectories and files.

Directory structure of "myenv" virtual environment.

```
my_project/
|-- myvenv/
|   |-- Scripts
|   |   |-- activate
|   |   |-- python
|   |   |-- pip
|   |
|   |-- lib/
|   |   |-- pythonX.X/
|   |       |-- site-packages/
|   |
|   |-- include/
|
|-- my_project_files/
|   |-- main.py
|   |-- requirements.txt
|   |-- ...
```

Understanding the directory structure for a virtual environment

Explanation of directory structure

- **venv/**: The root directory of the virtual environment. This is usually named venv, but any name can be given.
- **Scripts/**: Contains the executables for the virtual environment.
- **lib/**: Contains the Python standard library and site-packages specific to the virtual environment.
- **pythonX.X/**: This directory corresponds to the Python version used to create the virtual environment.
- **site-packages/**: This is where all the installed packages for the virtual environment are stored.
- **include/**: Contains C headers that are used by Python packages needing to be compiled.

2. Installing Django and Django REST Frameworks in an Isolated Environment

Installing Django and Django REST frameworks in an isolated environment

Setting up a virtual environment (isolated environment)

1. Install Python
   - Ensure you have Python installed. Django supports Python 3.6 and above.
   - You can check your Python version using:

     ```
     python –version
     ```

2. Install virtualenv
   - If you're using Python 3.3 or newer, virtualenv is included by default.
   - For older versions, you might need to install virtualenv using:

     ```
     pip install virtualenv
     ```

Installing Django and Django REST frameworks in an isolated environment

Setting up a virtual environment (isolated environment)

3. Create a Virtual Environment
   - Create a directory for your project and navigate into it.
   - Then create a virtual environment:

```
mkdir myproject
cd myproject
python -m venv myenv
```

4. Activate the Virtual Environment

```
myenv\Scripts\activate
```

Installing Django and Django REST frameworks in an isolated environment

Installing Django in an isolated environment

1. Update pip

```
pip install --upgrade pip
```

2. Install Django

```
pip install Django
```

3. Install Django REST Framework

```
pip install djangorestframework
```

Installing Django and Django REST frameworks in an isolated environment

Verifying the installation

1. Create a Django Project

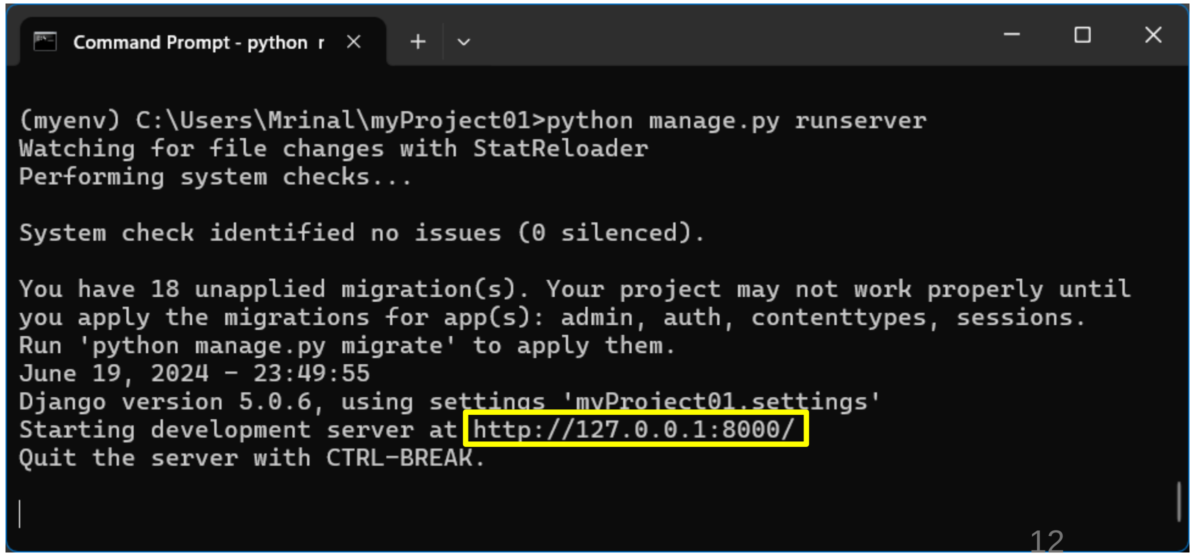
```
django-admin startproject myProject01
cd myproject
```

2. Run the Development Server

- Navigate into your project directory and run the development serve:

```
python manage.py runserver
```

After running the server, you will see similar output.

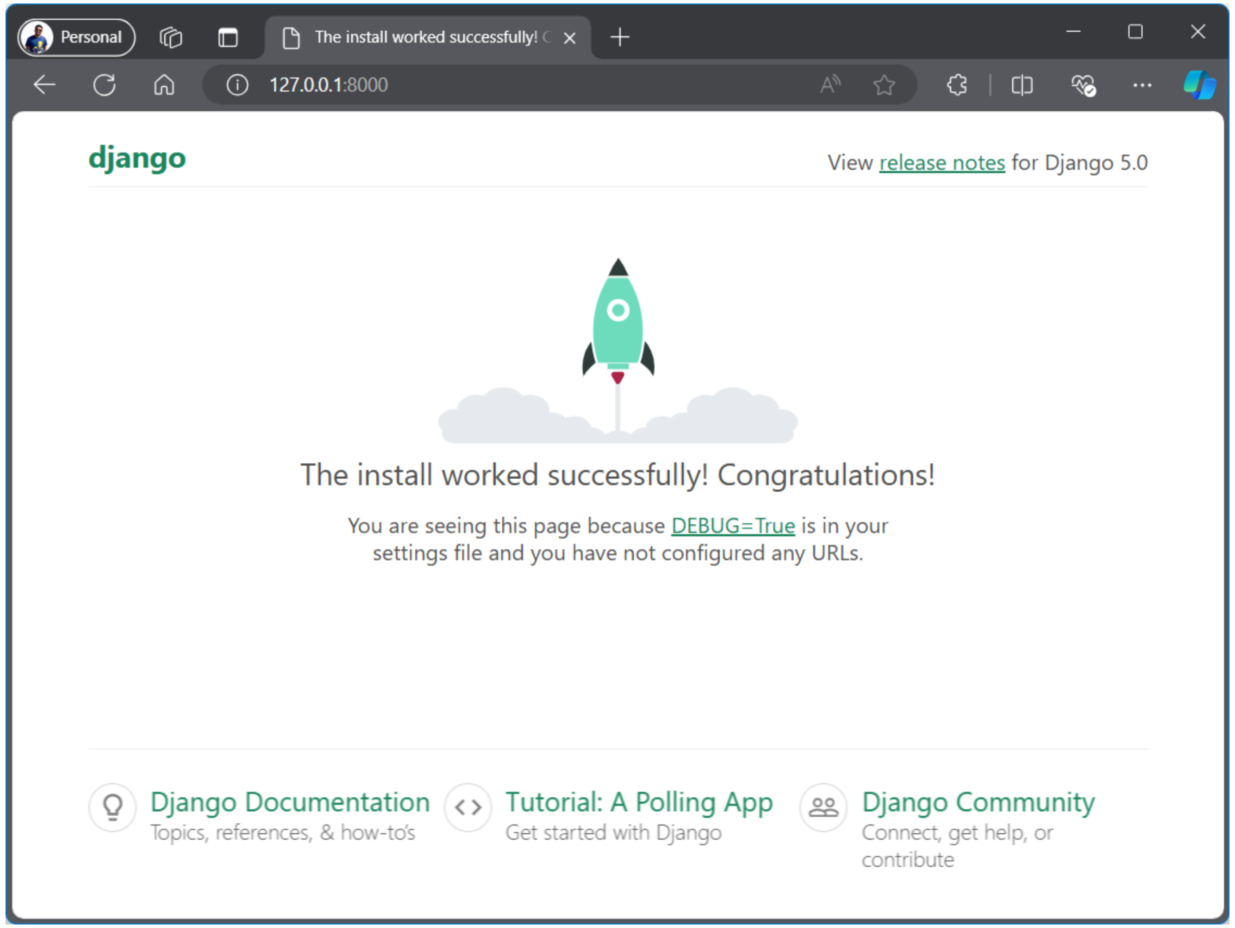


```
(myenv) C:\Users\Mrinal\myProject01>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
June 19, 2024 - 23:49:55
Django version 5.0.6, using settings 'myProject01.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Installing Django and Django REST frameworks in an isolated environment

Verifying the installation

Open the development server in browser, URL – http://127.0.0.1:8080



django

View release notes for Django 5.0

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's

Tutorial: A Polling App
Get started with Django

Django Community
Connect, get help, or contribute

3. Installing HTTPie and Installing the Postman REST Client

Installing HTTPie and installing the postman REST client

HTTPie

- HTTPie is a user-friendly command-line HTTP client.
- It is designed for testing APIs, making HTTP requests, and interacting with web services.
- It offers a simple syntax and provides formatted, colorized output for easy readability.

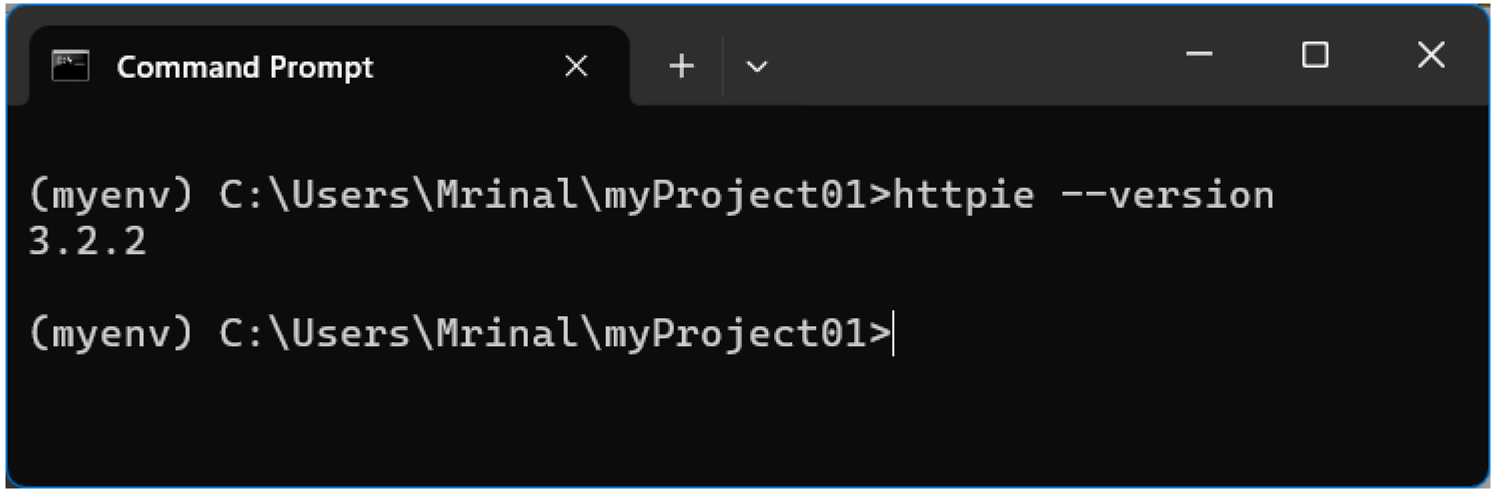
Install and verify HTTPie

- Use pip to install httpie:

  pip install httpie

- Verify httpie installation:

  httpie --version



```
(myenv) C:\Users\Mrinal\myProject01>httpie --version
3.2.2

(myenv) C:\Users\Mrinal\myProject01>
```

Installing HTTPie and installing the postman REST client

Postman

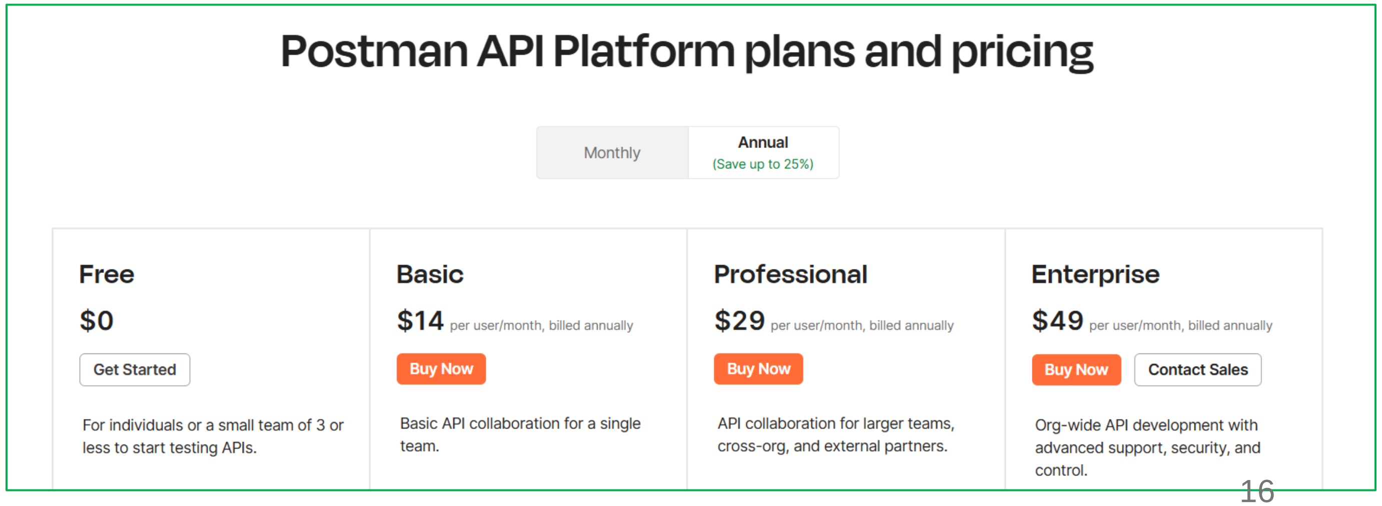
- Postman is a graphical tool for developing, testing, and debugging APIs.
- It provides a user-friendly interface to send HTTP requests, set up test cases, and manage environments.
- Postman supports a wide range of HTTP methods and allows for detailed configuration of requests.

Install Postman

Download postman from the link below and

Run the setup to install it.

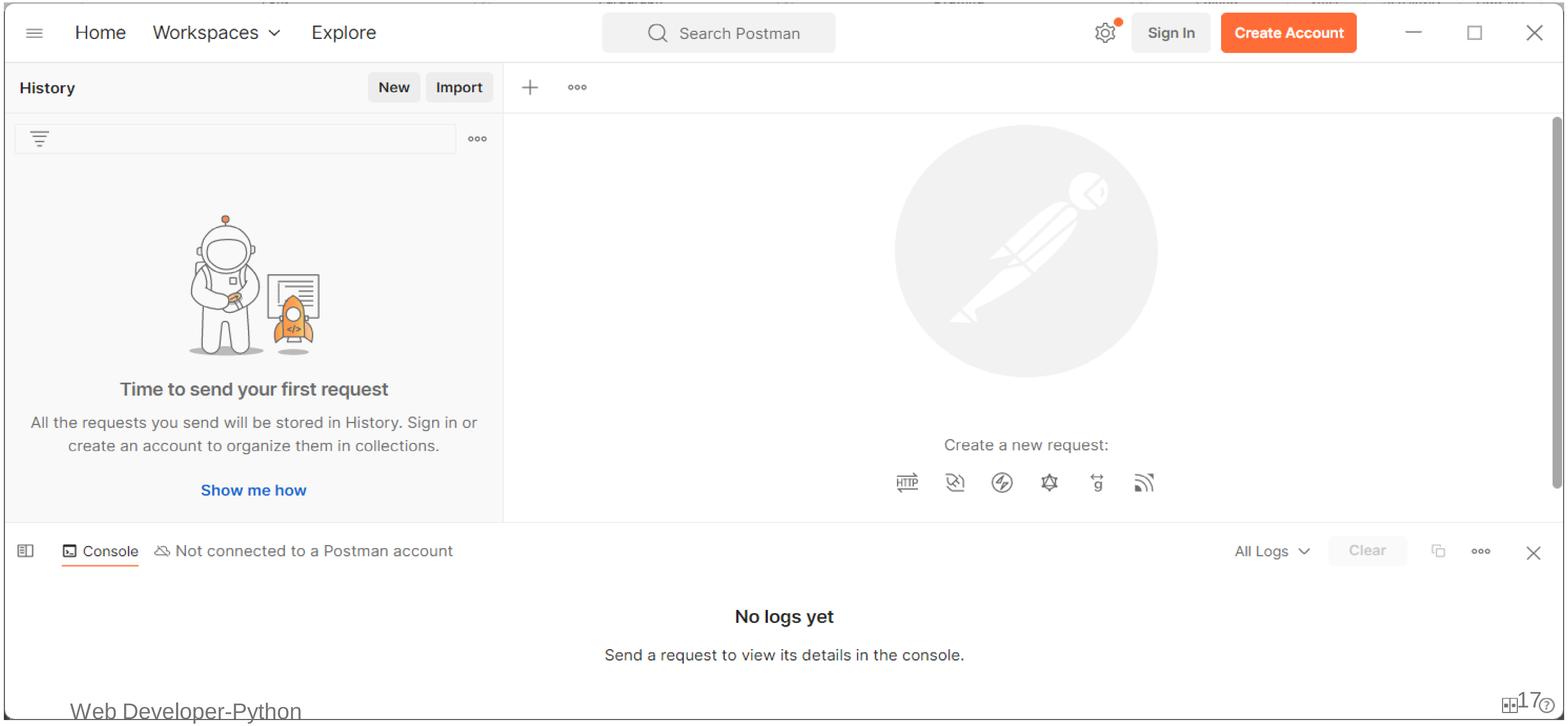
https://www.postman.com/pricing


Postman API Platform plans and pricing

Free	Basic	Professional	Enterprise
$0	$14 per user/month, billed annually	$29 per user/month, billed annually	$49 per user/month, billed annually
Get Started	Buy Now	Buy Now	Buy Now / Contact Sales
For individuals or a small team of 3 or less to start testing APIs.	Basic API collaboration for a single team.	API collaboration for larger teams, cross-org, and external partners.	Org-wide API development with advanced support, security, and control.

Installing HTTPie and installing the postman REST client

Postman



Home Workspaces Explore Search Postman Sign In Create Account

History New Import

Time to send your first request

All the requests you send will be stored in History. Sign in or create an account to organize them in collections.

Show me how

Create a new request:

Console Not connected to a Postman account All Logs Clear

No logs yet

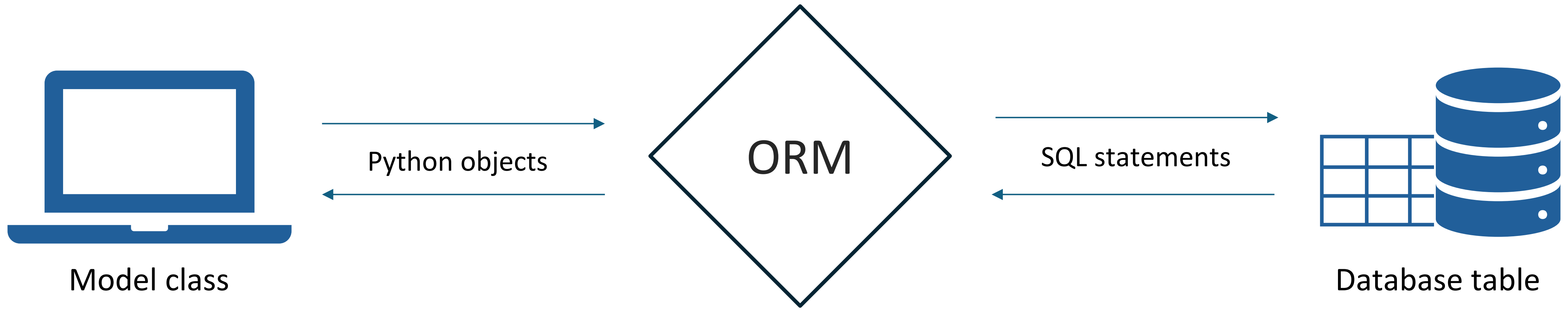
Send a request to view its details in the console.

4. Understanding the Table Generated by Django

Understanding the table generated by Django

Object-Relational Mapping (ORM)

- To understand the table generated by Django, it's crucial to delve into how Django's Object-Relational Mapping (ORM) works.
- Django ORM provides a high-level abstraction upon the relational database, enabling you to interact with your database using Python code rather than SQL queries.



Understanding the table generated by Django

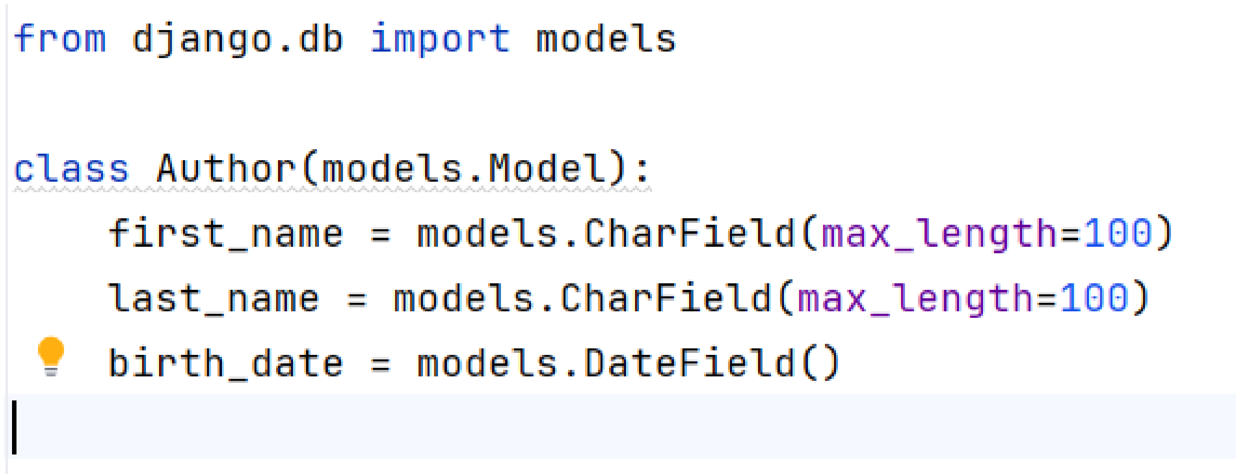
Defining models in Django

- A Django model is a class that inherits from django.db.models.Model.
- Each model represents a database table, and each attribute of the model represents a database field.

Understanding the table generated by Django

Defining models in Django

Example



```
from django.db import models

class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    birth_date = models.DateField()
```

This Author model translates to a table in the database with columns for first_name, last_name and birth_date.

Understanding the table generated by Django

Model fields and database columns

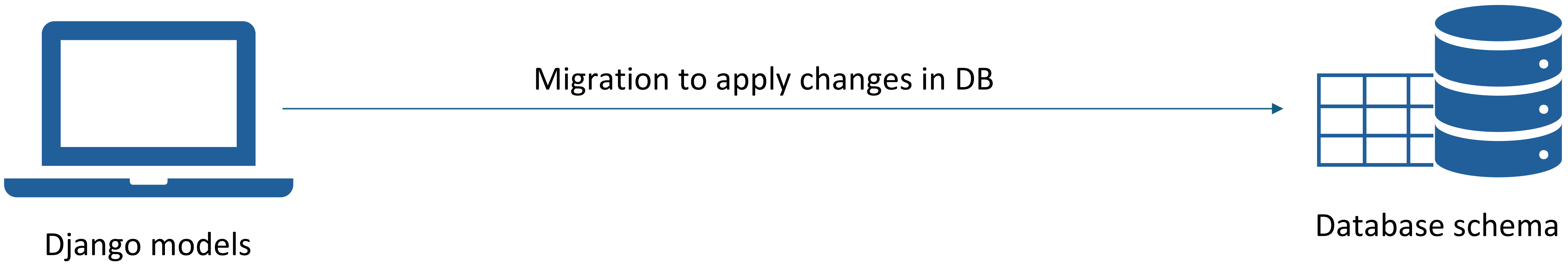
Django maps model fields to database columns with specific types.

- CharField is mapped to VARCHAR in SQL.
- DateField is mapped to DATE.
- Other fields like IntegerField, BooleanField, ForeignKey, etc. have their corresponding SQL datatypes.

Understanding the table generated by Django

Model fields and database columns

Django uses migrations to propagate changes made to models into the database schema.



Understanding the table generated by Django

Model fields and database columns

Migration commands

1. python manage.py makemigrations
2. python manage.py migrate

Running makemigrations generates migration files based on the model definitions and migrate applies these changes to the database.

Understanding the table generated by Django

Database table structure

Table structure generated by the Author model:

```sql
CREATE TABLE app_author (
    id SERIAL PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    birth_date DATE NOT NULL
);
```

Understanding the table generated by Django

Database table structure

Explanation

- id: An automatically added primary key field (SERIAL in PostgreSQL, AUTO_INCREMENT in MySQL).
- first_name: Mapped from CharField with max_length=100.
- last_name: Similar to first_name.
- birth_date: Mapped from DateField.

Understanding the table generated by Django

Relationships and foreign Key

Django handles relationships using foreign keys and other related fields.

Understanding the table generated by Django

Relationships and foreign Key

Example

Let's add a Book model that has a foreign key to Author.

```
class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    published_date = models.DateField()
```

This Book model translates to:

```
CREATE TABLE app_book (
    id SERIAL PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    author_id INTEGER NOT NULL,
    published_date DATE NOT NULL,
    FOREIGN KEY (author_id) REFERENCES app_author(id) ON DELETE CASCADE
);
```

Understanding the table generated by Django

Relationships and foreign Key

Explanation

- author_id : A foreign key referencing the Author table's id.
- on_delete=models.CASCADE : Ensures that when an author is deleted, all their books are also deleted.

Understanding the table generated by Django

Field options

Django model fields have several options to customize the database schema.

Example

```
class Publisher(models.Model):
    name = models.CharField(max_length=255, unique=True)
    established_year = models.IntegerField(null=True, db_index=True)
```

Explanation

- null=True: Allows NULL values in the database column.
- unique=True: Adds a unique constraint.
- db_index=True: Creates an index for faster lookups.

Understanding the table generated by Django

Field options

Resulting SQL

```
CREATE TABLE app_publisher (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    established_year INTEGER NULL,
    INDEX (established_year)
);
```

Understanding the table generated by Django

Meta options

Django's Meta class within models allows you to define additional options like –

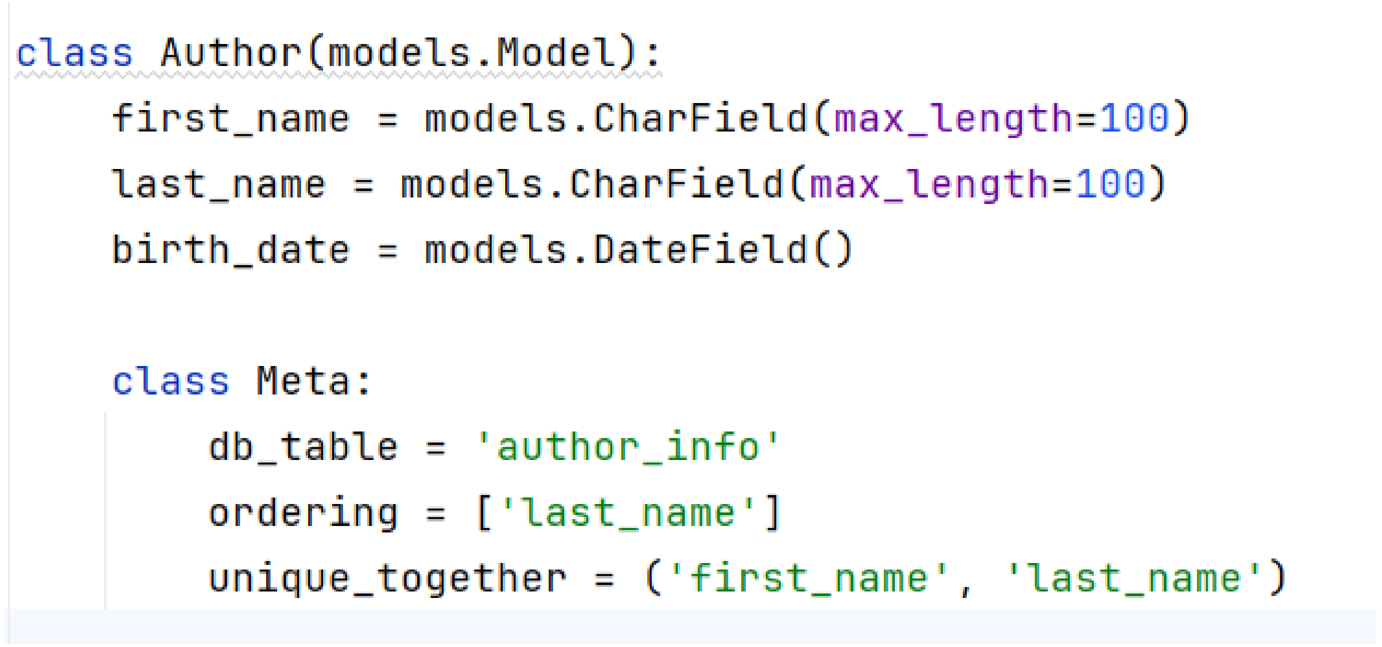
- table name
- rrdering
- unique constraints.

Understanding the table generated by Django

Meta options

Example - Change Meta options on author

- Alter unique_together for author
- Rename table for author to author_info



```python
class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    birth_date = models.DateField()

    class Meta:
        db_table = 'author_info'
        ordering = ['last_name']
        unique_together = ('first_name', 'last_name')
```

Understanding the table generated by Django

- ✓ Model Definition: Define models as Python classes.
- ✓ Field Mapping: Map model fields to database columns.
- ✓ Migrations: Generate and apply migrations to sync models with the database.
- ✓ Table Structure: Understand the SQL schema generated by Django ORM.
- ✓ Relationships: Handle relationships using foreign keys.
- ✓ Field Options: Customize fields with various options.
- ✓ Meta Options: Use Meta class for additional table settings.

5. Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Create a Django app

In windows Terminal navigate to your desired project directory and execute the following commands -

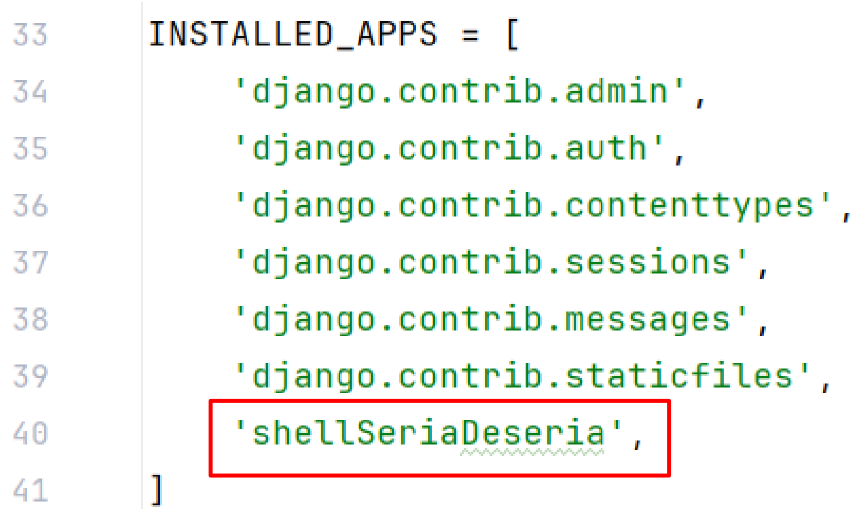
Step No.	Action	Command
1	Create a virtual environment	python –m venv myEnv
2	Activate virtual environment	myEnv\Sctripts\activate
3	Install Django	python –m pip install Django
4	Install Django REST framwork	python –m pip install djangorestframework
5	Create a Django project	django–admin startproject demoProj4
6	Create a Django app	cd demoProj4 python manage.py startapp shellSeriaDeseria

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

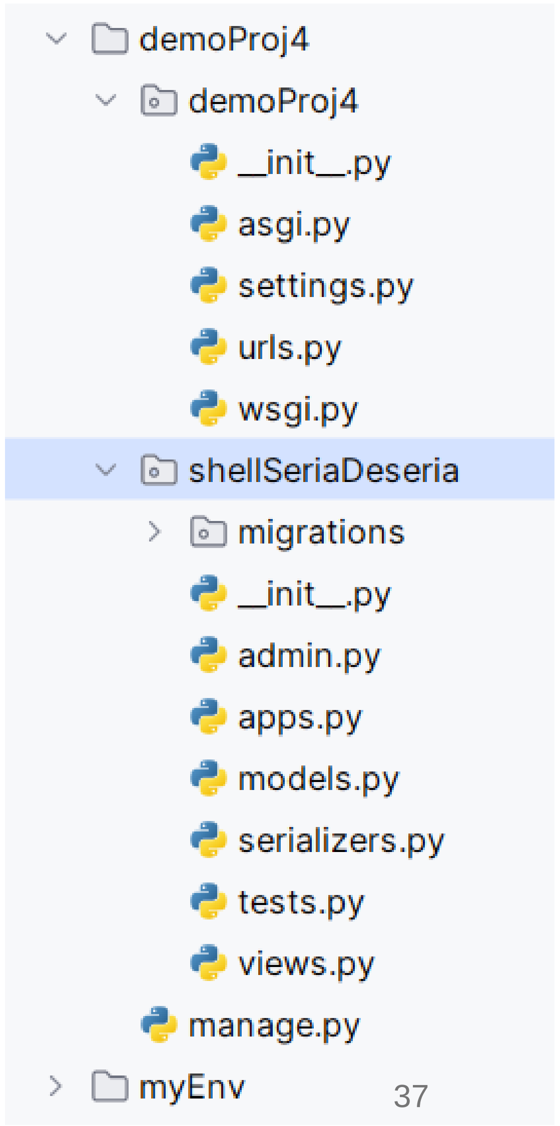
Registering the Django app

This will be directory structure after executing the all the previous commands.

Add the newly created app to the INSTALLED_APPS list in “demoProj4/settings.py”.



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

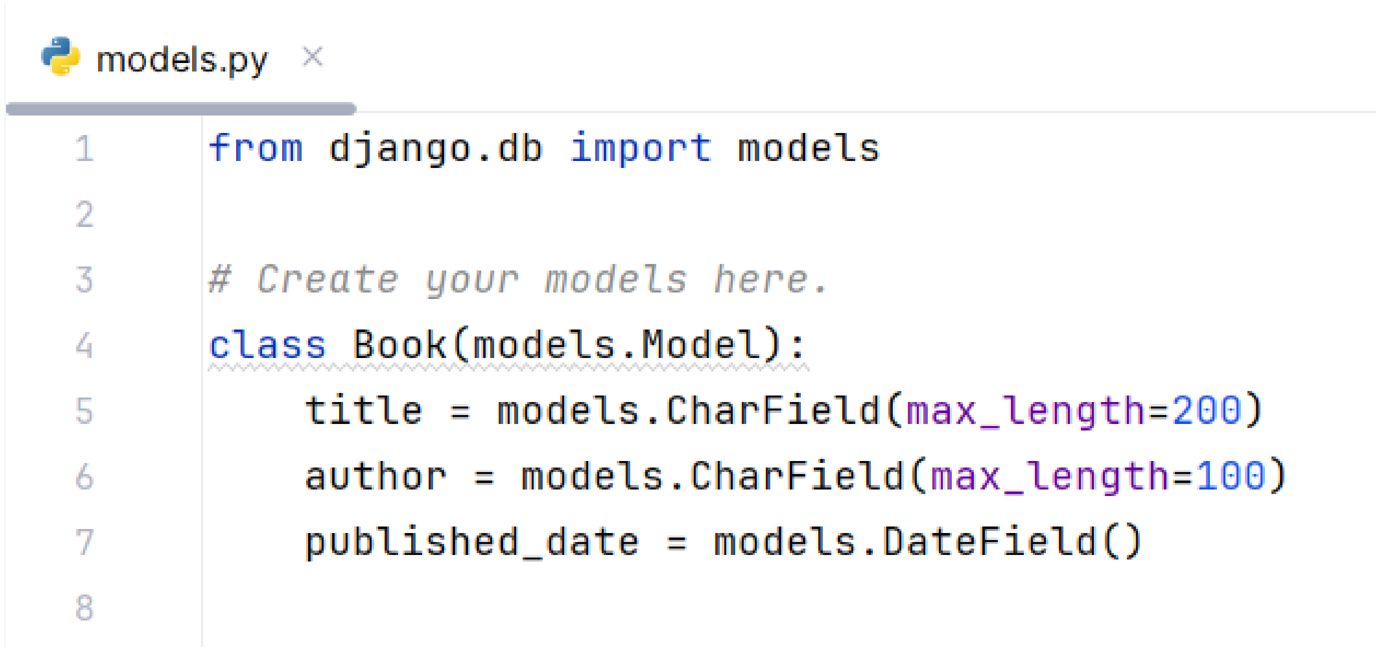


```
demoProj4
  demoProj4
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  shellSeriaDeseria
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    serializers.py
    tests.py
    views.py
  manage.py
myEnv
```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Define a model

Edit the “shellSeriaDeseria/models.py” file and add this code.


models.py

```python
from django.db import models

# Create your models here.
class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()

```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

ICTACADEMY

Define a model

Explanation

- This defines a Book model with three fields: title, author and published_date.
- models.CharField is used for short text fields.
- models.DateField is used for date fields.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

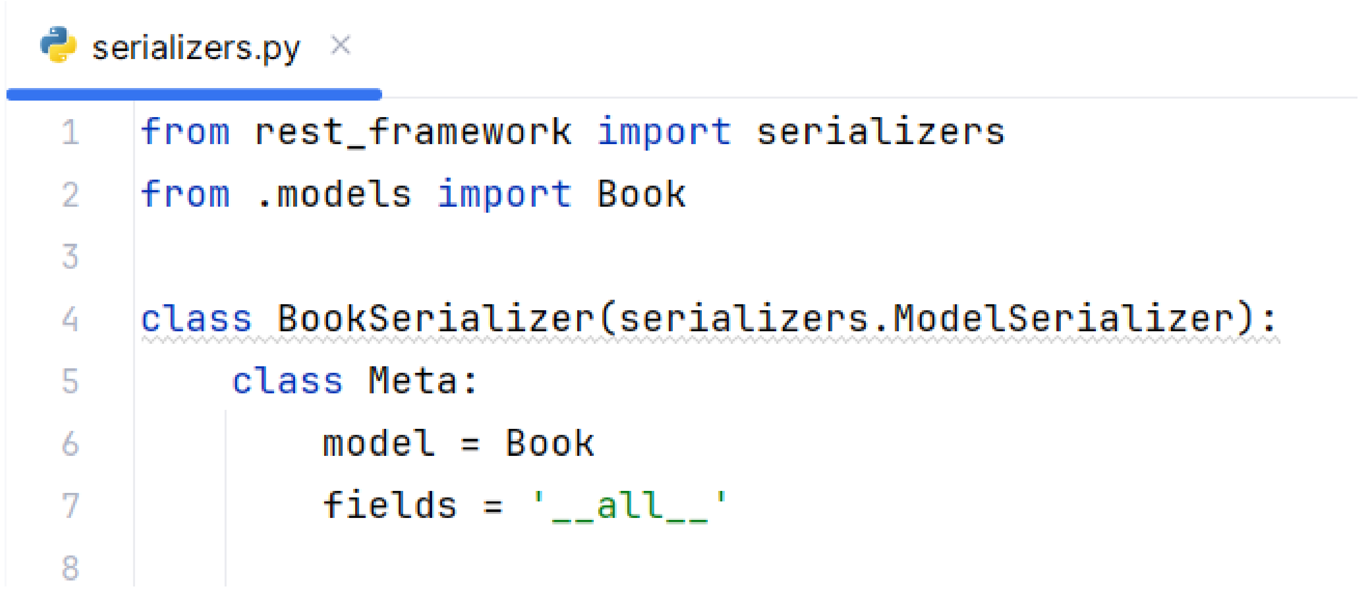
Create a serializer

A serializer in Django REST framework is used to convert a Django model instance or queryset into a native Python datatype, which can then be easily rendered into JSON, XML, or other content types.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Create a serializer

Create a new file “serializers.py” in “shellSeriaDeseria” directory.


serializers.py

```python
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = '__all__'

```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Create a serializer

Explanation

- This defines a serializer for the Book model.
- serializers.ModelSerializer automatically generates a serializer with fields that correspond to the model.
- The Meta class specifies the model to serialize and that all fields should be included.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Migrate models with database schemas

Run the following commands:

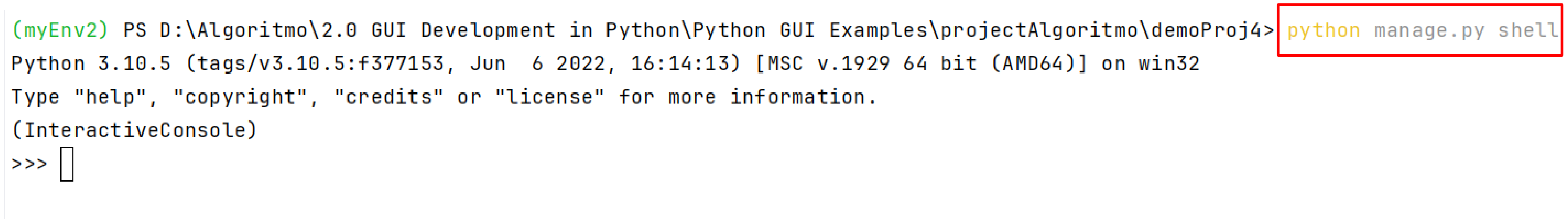
1. python manage.py makemigrations shellSeriaDeseria
2. python manage.py migrate

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Open Django shell

To start the Django shell, run :

```
python manage.py shell
```



```
(myEnv2) PS D:\Algoritmo\2.0 GUI Development in Python\Python GUI Examples\projectAlgoritmo\demoProj4> python manage.py shell
Python 3.10.5 (tags/v3.10.5:f377153, Jun  6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Interact with shell

In the Django shell or a view, you can create and serialize a “Book” instance.

Run the following code in shell interactively.

```
from shellSeriaDeseria.models import Book
from shellSeriaDeseria.serializers import BookSerializer

book = Book(title='Django for Beginners', author='William S. Vincent', published_date='2018-11-30')
book.save()

# Verify the book instance
print(book.title, book.author, book.published_date)

# Serialize the book instance
serializer = BookSerializer(book)
print(serializer.data)
```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Interact with shell

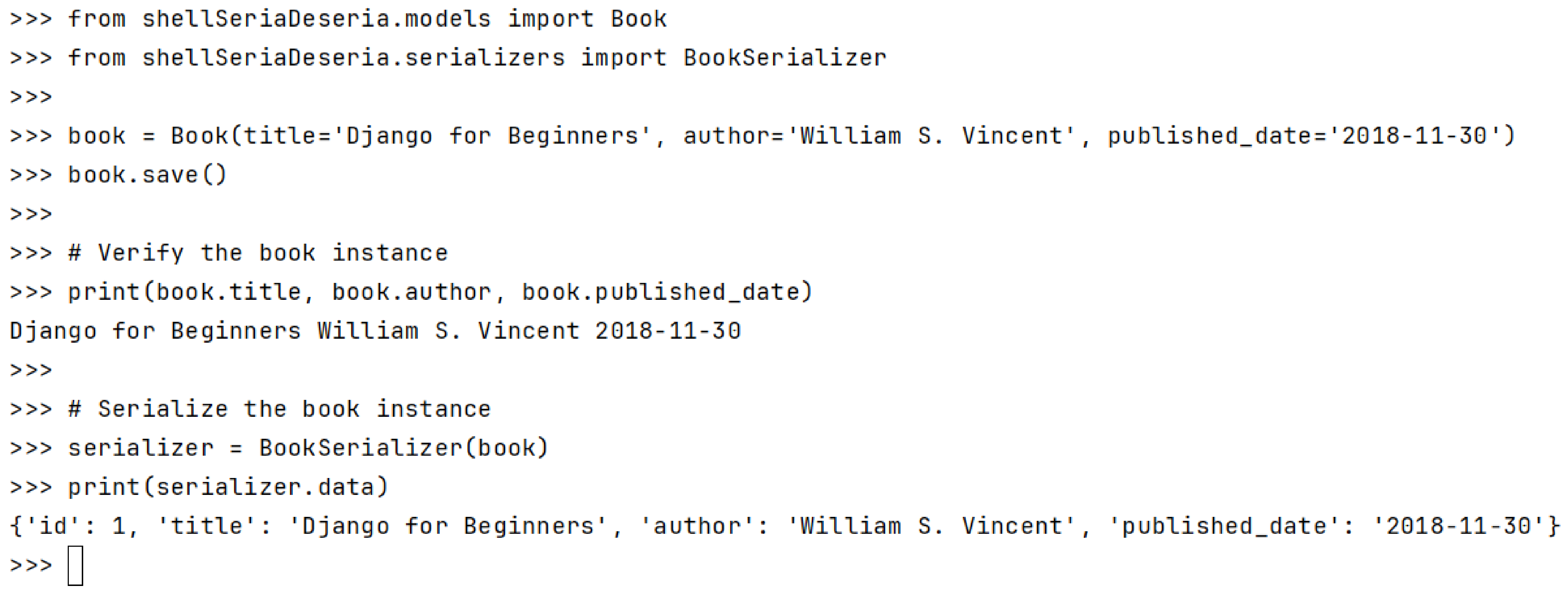
Explanation

- The BookSerializer converts the Book instance into a Python dictionary.
- The dictionary can then be rendered as JSON or another content type.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Interact with shell

Output from the shell.



```
>>> from shellSeriaDeseria.models import Book
>>> from shellSeriaDeseria.serializers import BookSerializer
>>>
>>> book = Book(title='Django for Beginners', author='William S. Vincent', published_date='2018-11-30')
>>> book.save()
>>>
>>> # Verify the book instance
>>> print(book.title, book.author, book.published_date)
Django for Beginners William S. Vincent 2018-11-30
>>>
>>> # Serialize the book instance
>>> serializer = BookSerializer(book)
>>> print(serializer.data)
{'id': 1, 'title': 'Django for Beginners', 'author': 'William S. Vincent', 'published_date': '2018-11-30'}
>>>
```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Create a deserializer

A deserializer in Django REST framework is used to convert parsed data back into complex data types, such as creating a Django model instance from JSON data.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Interact with shell

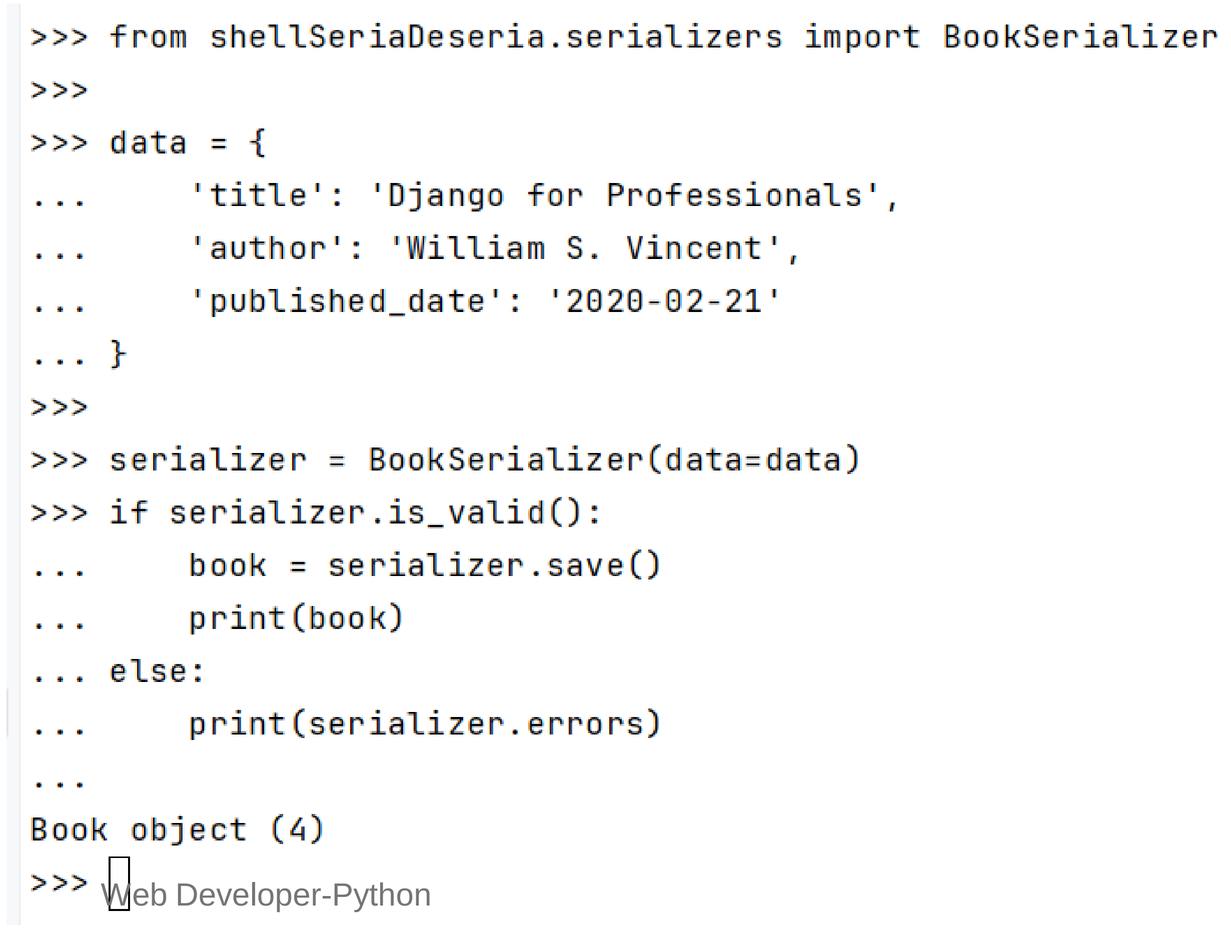
Run this code in shell to deserialize data to create a model instance of “BookSerializer”:

```
from shellSeriaDeseria.serializers import BookSerializer

data = {
    'title': 'Django for Professionals',
    'author': 'William S. Vincent',
    'published_date': '2020-02-21'
}

serializer = BookSerializer(data=data)
if serializer.is_valid():
    book = serializer.save()
    print(book)
else:
    print(serializer.errors)
```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Interact with shell

Output from the shell.



```
>>> from shellSeriaDeseria.serializers import BookSerializer
>>>
>>> data = {
...     'title': 'Django for Professionals',
...     'author': 'William S. Vincent',
...     'published_date': '2020-02-21'
... }
>>>
>>> serializer = BookSerializer(data=data)
>>> if serializer.is_valid():
...     book = serializer.save()
...     print(book)
... else:
...     print(serializer.errors)
...
Book object (4)
>>>
```

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Create a deserializer

Explanation

- The BookSerializer is used to validate the incoming data.
- If the data is valid, a Book instance is created and saved to the database.
- If the data is invalid, error messages are printed.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

Use cases

- Serializers: Often used in APIs to render database records as JSON responses.
- Deserializers: Often used in APIs to parse incoming JSON requests and save the data to the database.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

TRY IT

Case Study: Library Management System

- You are working on a Library Management System using Django.
- The system needs to handle the creation, serialization and deserialization of books in the library.
- You will use the Django shell to interact with the models, create serializers to convert data to and from JSON, and handle deserialization to create and update instances from JSON data.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

TRY IT

Case Study: Library Management System

Task 1: Working with the Django Shell

Objective: Use the Django shell to create, update, and delete instances of the Book model.

- Create a new book instance in the Django shell with the following details:

  Title: "Learning Python“

  Author: "Mark Lutz“

  Published Date: "2013-07-06“

- Retrieve and display all book instances in the database.
- Verify that the newly created book is listed.
- Update the book instance with the title "Learning Python" to have a new author name "David Ascher".
- Delete the book instance with the title "Learning Python".

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

TRY IT

Case Study: Library Management System

Task 2: Serialization

Objective: Create a serializer for the Book model and serialize book instances to JSON.

- Create a serializer for the Book model in myApp3/serializers.py.
- Serialize a book instance with the following details:

  Title: "Python Crash Course"

  Author: "Eric Matthes"

  Published Date: "2015-11-01"
- Print the serialized data to the console.

Working with the Django Shell and Diving Deeply into Serialization and Deserialization

TRY IT

Case Study: Library Management System

Task 3: Deserialization

Objective: Deserialize JSON data to create and update Book model instances.

- Deserialize JSON data to create a new book instance with the following details:

    Title: "Fluent Python"

    Author: "Luciano Ramalho"

    Published Date: "2015-08-20"

- Deserialize JSON data to update an existing book instance with the following details:

    Existing Title: "Python Crash Course"

    New Author: "Eric Matthes (Updated)"

    New Published Date: "2019-05-03"

Question?

Thank you