Dyna Pajo Buna **BSCPE-2A**

Laboratory Activity No. 2:

Laboratory Activity No. 2:

Topic belongs to: **Software Design and Database Systems**

Title: *Designing the Database Schema for the Library Management System*

Introduction: In this activity, you will design the database schema for the Library Management System. The database will include tables for books, authors, users, and borrowing records. You will also learn how to use Django's ORM (Object-Relational Mapping) to define the models.

Objectives:

- Design the database schema for the Library Management System.
- Create Django models to represent the schema.
- Use Django's ORM to interact with the database.

Theory and Detailed Discussion: Django uses an ORM (Object-Relational Mapping) system to map Python objects to database tables. By defining models in Python code, Django automatically creates the corresponding database tables. We will start by designing the database schema with the necessary relationships between entities like books, authors, and users.

Django Program or Code: Write down the summary of the code for models that has been provided in this activity.

- python manage.py startapp books
- python manage.py startapp users
- from django.db import models

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

def __str__(self):
        return self.name

class Book(models.Model):
```

```
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    isbn = models.CharField(max_length=13)
    publish_date = models.DateField()

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

- ```
  from django.db import models
  from books.models import Book

  class User(models.Model):
      username = models.CharField(max_length=100)
      email = models.EmailField()

  def __str__(self):
      return self.username

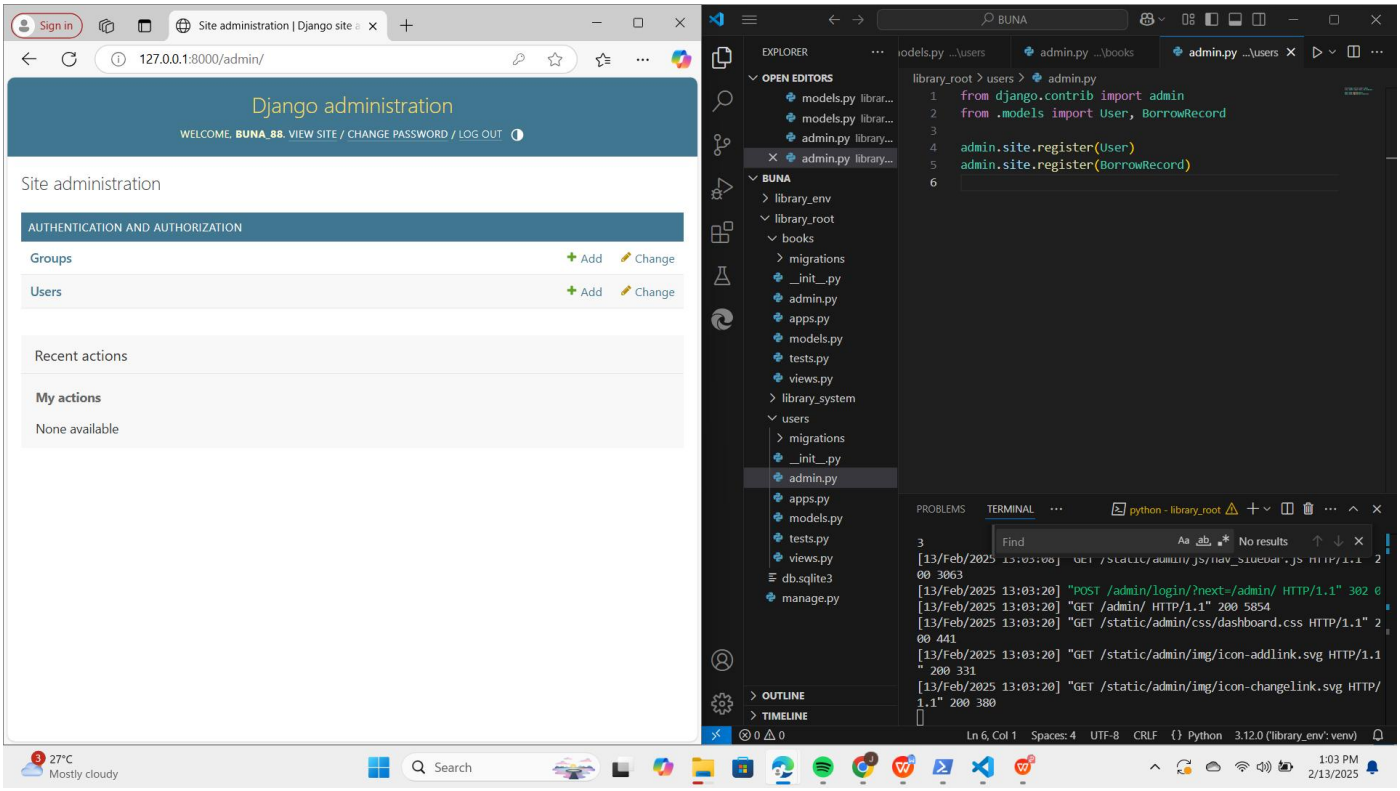
  class BorrowRecord(models.Model):
      user = models.ForeignKey(User, on_delete=models.CASCADE)
      book = models.ForeignKey(Book, on_delete=models.CASCADE)
      borrow_date = models.DateField()
      return_date = models.DateField(null=True, blank=True)
  ```
- python manage.py makemigrations
- python manage.py migrate
- python manage.py createsuperuser
- ```
  from django.contrib import admin
  from .models import Author, Book

  admin.site.register(Author)
  admin.site.register(Book)
  ```
- ```
  from django.contrib import admin
  from .models import User, BorrowRecord
  ```
  ```

admin.site.register(User)

admin.site.register(BorrowRecord)

Results: By the end of this activity, you will have successfully defined the database schema using Django models, created the corresponding database tables, and registered the models in the admin panel. (print screen the result and provide the github link of your work)



Follow-Up Questions:

1. What is the purpose of using ForeignKey in Django models?

Answer: A **ForeignKey** in Django establishes a **many-to-one** relationship between two models, allowing one model to reference another while maintaining **data integrity** and **relational consistency**.

Purposes:
Defines Relationships – Links models in a relational database (e.g., a book is associated with one author, but an author can have multiple books).
Ensures Data Integrity – Prevents referencing non-existent objects (e.g., a loan record must link to an existing book and user).
Optimizes Queries – Enables efficient data retrieval using Django's ORM (e.g., book.author.name fetches the author's name directly).
Handles Cascade Deletions – The on_delete parameter determines what happens when a referenced object is deleted:

- models.CASCADE: Deletes related objects when the referenced object is removed.
- models.SET_NULL: Sets the foreign key field to NULL instead of deleting it.
- models.PROTECT: Prevents deletion if related objects exist.

2. How does Django's ORM simplify database interaction?

Answer: Django's **ORM (Object-Relational Mapping)** streamlines database interactions by eliminating the need for **raw SQL**, providing a **Pythonic approach** to data handling.

Key Advantages:
No Raw SQL Required – Perform CRUD (Create, Read, Update, Delete) operations using Python methods instead of SQL queries.
Automated Database Schema Creation – Defining models in models.py generates the necessary database tables automatically.
Optimized Queries – The ORM enhances efficiency by optimizing database queries in the background.
Seamless Relationship Handling – Supports relationships like ForeignKey, ManyToManyField, and OneToOneField with built-in management.
Security – Prevents SQL injection by automatically escaping query parameters.
Cross-Database Compatibility – Works with multiple databases (SQLite, PostgreSQL, MySQL, etc.) without modifying queries—just update the DATABASES setting in settings.py.

Findings:

- The **ForeignKey** field connects models in a **many-to-one** relationship (e.g., a book is linked to one author, but an author can have multiple books).
- Django's **ORM** allows developers to work with databases using Python, eliminating the need for raw SQL.
- ORM **automates SQL generation, enforces relationships, and maintains data integrity**, reducing manual database management.
- CRUD operations become more **seamless** and **efficient** with ORM.

Summary:

Django's **ForeignKey** establishes structured relationships between models, ensuring efficient database management. Meanwhile, Django's **ORM** abstracts complex SQL operations, making database interactions simpler, more readable, and consistent.

Conclusion:

By utilizing **ForeignKey relationships** and **Django's ORM**, database management becomes more **organized, scalable, and secure**. ORM enhances productivity by simplifying queries, reducing complexity, and ensuring data consistency, making Django an efficient framework for web development.