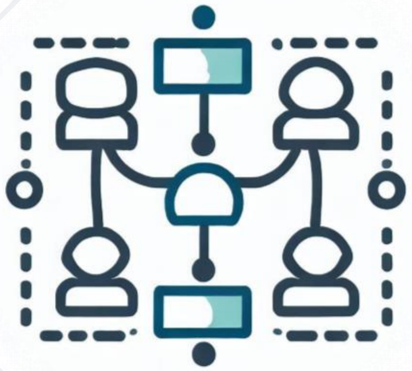
Django Models Relations



SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-db

Table of Contents

1. Database Normalization
   - Introduction
   - Key Benefits
2. Relationships in Django Models
   - Introduction
   - Foreign Key, Related Name
3. Types of Relationships
   - One-to-One, One-to-Many, Many-to-Many

Database Normalization

Database Normalization

- A **process** that helps **organize** and **structure** relational databases **efficiently**
- A set of **guidelines** or **rules** that are applied
  - **Designing** the database **schema**
  - **Minimizing** data **redundancy**
  - **Ensuring** data **integrity**
  - **Eliminating** data **anomalies**
  - **Improving** the **efficiency** and **maintainability**

Database Normalization

- **Database normalization** is **crucial** for
  - maintaining data **integrity**
  - **improving** data **consistency** and **accuracy**
  - **simplifying maintenance**
  - **enhancing** query **performance**
- Follow **normalization principles** to
  - design a **well-structured** and **efficient** database schema
  - ensure the **long-term viability** and **usability** of applications

Elimination of Data Redundancy

- **Redundancy** refers to storing the **same piece of data** in multiple places within a database
  - Leading to **inconsistencies** and update **anomalies**
  - **Normalization** helps **eliminate** or **minimize** data **redundancy** by
    - **breaking down** data into **smaller**, more **atomic** units
    - storing data only **once**, **reducing** storage requirements
    - ensuring **consistency** **throughout** the database

Data Integrity

- **Normalization** helps **maintain** data **integrity** by
  - enforcing **rules** and **constraints** on the **relationships** between tables
  - **ensuring** that each piece of data is stored in **one place**
  - **preventing** **inconsistencies** or **contradictory** information
  - adhering to the **normalization rules**
    - **minimizes** the risk of **data** **corruption**
    - **maintains** the **accuracy** and **reliability**

Improved Data Consistency and Accuracy

- Normalization promotes consistency by
  - ensuring that **updates** or **modifications** are made in **one place only**
    - **reducing** the likelihood of inconsistencies
  - ensuring that **changes propagate correctly** throughout the database
    - avoiding **duplicate** or **conflicting** data
    - enhancing the accuracy and reliability

Simplified Database Maintenance

- **Normalized databases** are typically **easier** to **maintain** and **modify**
  - Data is stored in a **structured** and **organized** manner
  - Making **changes** or **adding** new data becomes **more straightforward**
  - **Relationships** between tables are **well-defined**
  - **Easier** to **understand** and **work** with the database **schema**

Enhanced Query Performance

- **Normalization** can **improve** the **performance** of database **queries**
  - Breaking down data into **smaller tables**
  - Establishing **relationships** to **optimize** the way queries **retrieve** and **join** data
  - Properly **indexed normalized tables** can
    - **speed up** data retrieval
    - lead to more **efficient query execution**

Scalability and Flexibility

- **Normalized** databases are **more scalable** and **flexible**
- When **properly normalized**, databases **become**
  - **easier** to accommodate **changes**
  - **scalable** as data requirements **evolve**
    - adding **new** tables or **modifying** existing ones
  - **less complicated**, allowing for greater **flexibility** in **adapting** to **future needs**

Relationships in Django Models

Foreign Keys, Related Names

Model Relationships

- **Model relationships** allow **defining**
  - how different database **tables** (**models**) **relate** to each other
- **Relationships** are established using **fields**
  - `ForeignKey`
  - `ManyToManyField`

Foreign Key

- **Foreign key** is a **field** in a model that
  - refers to the **primary key** of **another model**
  - represents a **one-to-many relationship** between models
  - establishes **a link** between **two models**
    - **one** model has **a reference** to **another** model's **primary key**

Related Name

- When **defining** a **foreign key** from one model to another, you can
  - specify a **related name** for the **reverse relationship**
- The **related name** allows you to **access related objects** from the **other** model **conveniently**
- By **default**, the **related name** is generated **automatically**
  - by appending **_set** to the lowercase **name** of the **model** that defines the **foreign key**

Related Name

- **Customize** the **related name**
  - by specifying the `related_name` **attribute** when defining the **foreign key**
- By **explicitly** setting the **related name**
  - you have **more control** over the **naming** of the **reverse relationship**
- The **related name** provides a way to
  - access **related objects** in the **reverse direction**
  - **simplify** querying and **traversal** of **relationships**

Types of Relationships

One-to-Many, Many-to-Many, One-to-One

Many-to-One (One-to-Many) Relationship

- Use the **field** `ForeignKey` to define it
  - Requires **two** positional **arguments**
    - The **class to** which the model is **related**
    - The `on_delete` option
  - `related_name` is optional

```
class Department(models.Model):...

class Employee(models.Model):
    ...
    department = models.ForeignKey(to=Department,
on_delete=models.CASCADE, related_name='employees')
```

On Delete Option

- You can reproduce the behavior of the SQL constraint **ON DELETE** using **Python code**

```
class Employee(models.Model):
    ...
    manager = models.ForeignKey(to=Manager,
                                on_delete=models.SET_NULL, null=True)

    department = models.ForeignKey(to=Department,
                                on_delete=models.RESTRICT)
```

More at: *https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.ForeignKey.on_delete*

Problem: The Lecturer

- You are given an empty **ORM project skeleton** (you can download it from here) needed to **create a University Management System**
- First, in the **`main_app`** create two models called **`"Lecturer"`** and **`"Subject"`**
- A full description of the problem can be found in the Lab document **here**

Solution: The Lecturer

```
# models.py
class Lecturer(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    # TODO: Implement a __str__ method

class Subject(models.Model):
    name = models.CharField(max_length=100)
    code = models.CharField(max_length=10, unique=True)
    lecturer = models.ForeignKey(to='Lecturer',
on_delete=models.SET_NULL, null=True)
    # TODO: Implement a __str__ method
```

Many-to-Many Relationship

- ManyToManyField
  - Requires **one** positional **argument**
    - The **class to** which the model is **related**

```
class Project(models.Model):...

class Employee(models.Model):
    ...
    projects = models.ManyToManyField(Project)
```

- Doesn't matter **which model** has the field, but it should be only put in **one** of the models

Problem: The Student

- In the **`main_app`** create one **additional** model called **`"Student"`**
- A full description of the problem can be found in the Lab document **here**

Solution: The Student

```
# models.py

class Student(models.Model):
    student_id = models.CharField(max_length=10, primary_key=True)
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    birth_date = models.DateField()
    email = models.EmailField(unique=True)
    subjects = models.ManyToManyField(to='Subject')
```

Through Option

- When creating **many-to-many** relationship, Django **automatically** creates an **intermediary** (a.k.a **junction** or **join**) **table**
- To **manually specify** the **intermediary model**, use the **`through`** option
  - It creates a Django **intermediary table** that represents it
  - Django will use that **model** to **manage** the **relationship**
- Mostly used when associating **extra data** with a many-to-many relationship
  - Gives more control
  - Allows adding extra fields

Through Option

- Using the **`through`** option
  - Allows performing **various queries**
  - Gives access to the **extra fields** in the intermediary model
  - Provides more **flexibility** when dealing with **many-to-many relationships** in Django

Example: Through Option

```
class Employee(models.Model):...

class Project(models.Model):
     ...
    employees = models.ManyToManyField(
        Employee, through='ProjectAssignment'
    )

class ProjectAssignment(models.Model):
    employee = models.ForeignKey(Employee,
on_delete=models.CASCADE)
    project = models.ForeignKey(Project, on_delete=models.CASCADE)
    start_date = models.DateField()
    role = models.CharField(max_length=30)
```

Problem: The Enrollment

- Improve the management system by adding a **`"through"`** option to the **`Student's`** "**`subjects`**" field and a **`"through"`** table
- A full description of the problem can be found in the Lab document **here**

Solution: The Enrollment

```
class Student(models.Model):
    ...
    subjects = models.ManyToManyField(to='Subject',
                through='StudentEnrollment')

class StudentEnrollment(models.Model):
    student = models.ForeignKey(to='Student',
                                on_delete=models.CASCADE)
    subject = models.ForeignKey(to='Subject',
                                on_delete=models.CASCADE)
    enrollment_date = models.DateField(default=date.today)
    grade = models.CharField(max_length=1, choices=GRADE_CHOICES,
blank=True, null=True)
    # Add grade choices
```

One-to-One Relationship

- OneToOneField
  - Requires **two** positional **arguments**
    - the class **to** which the model is **related**
    - on_delete option
- Most useful **on the primary key** of an object when that object "**extends**" another object in some way

```
class Address(models.Model):...

class BusinessBuilding(models.Model):
    address = models.OneToOneField(
        Address, on_delete=models.CASCADE, primary_key=True)
    ...
```

Problem: The Lecturer Profile

- In the **`main_app`** create one more **additional** model called **`"LecturerProfile"`**
- A full description of the problem can be found in the Lab document **here**

Solution: The Lecturer Profile

```
# models.py

class LecturerProfile(models.Model):
    lecturer = models.OneToOneField(to=Lecturer,
                                    on_delete=models.CASCADE)
    email = models.EmailField(unique=True)
    bio = models.TextField(blank=True, null=True)
    office_location = models.CharField(max_length=100,
                                       blank=True, null=True)
```

Self-referential Foreign Key

- When creating a **relation with instances of the same model**
  - Used to establish **hierarchical** or **recursive** relationships **within** a **single** model

```
class Employee(models.Model):
    ...
    manager = models.ForeignKey('self',
        on_delete=models.SET_NULL, null=True,
        blank=True, related_name='employees')
```

Lazy Relationships

- When resolving **circular dependencies** between two models
  - Using **strings** to **define** model **relationships** **without direct imports**

```
class Manager(models.Model):
    ...
    team = models.ManyToManyField('Employee')

class Employee(models.Model):
    ...
    team_leader = models.ForeignKey('Manager', ...)
```

Summary

- Database Normalization
- Relationships in Django Models
  - Foreign Key, Related Name
- Types of Relationships
  - One-to-Many
  - Many-to-Many
  - One-to-One

Questions?

Software University

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

СофтУни БУДИТЕЛ

Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License