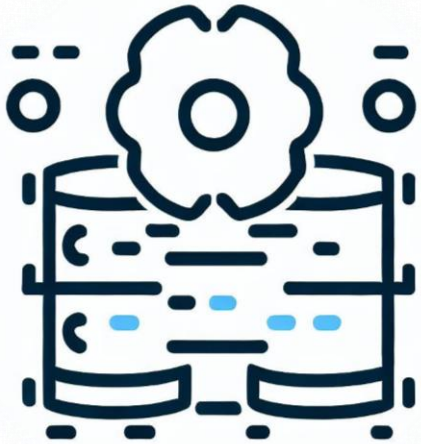
Advanced Django Model Techniques



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-db

Table of Contents

Validation in Models

Built-in & Custom Validators

Built-in Field Validators

Software University

- Django provides **built-in field validators** allowing to
  - **validate** the **data** entered in the model fields
  - **ensure** that the data **meets** **specific requirements** before saving it
- Common **built-in validators**
  - `MaxValueValidator`, `MinValueValidator`
  - `MaxLengthValidator`, `MinLengthValidator`
  - `RegexValidator`

More at: *https://docs.djangoproject.com/en/5.0/ref/validators/#built-in-validators*

Built-in Field Validators

- **Min/Max Length/Value Validators** accept **two** arguments
  - `limit_value`
    - A **required**, **first positional** argument that **specifies** the **limit value**
  - `message`
    - A **default** argument
    - `message=None` is being passed **by default**
- Validators raise a `ValidationError`
  - If the **field value** does **not meet** the **requirements**

Built-in Field Validator Example

- "validators" **option** is used to **apply** a list of **validators** to a **specific** field

```
from django.core.validators import MinLengthValidator
from django.db import models

class Employee(models.Model):

    first_name = models.CharField(max_length=100,
        validators=[MinLengthValidator(2, message='First name
should be at least 2 chars long')])

    ...
```

Import a built-in validator

Limit value

Message will be None by default if you do not pass one

One or more validators can be passed as a list or a tuple

Problem: Restaurant

- You are given an empty **ORM project skeleton** (download it from **here**). Your task is to **create** a **Restaurant Review System**
- First, in the **`main_app`** create a model called **`"Restaurant"`**, with the fields: **`"name"`**, **`"location"`**, **`"description"`**, and **`"rating"`**
- A full description of the problem can be found in the Lab document **here**

Solution: Restaurant

```
class Restaurant(models.Model):
    name = models.CharField(max_length=100, validators=[
        validators.MinLengthValidator(2, "Name must be at least
2 characters long."),
        validators.MaxLengthValidator(100, "Name cannot exceed
100 characters.")])
    location = models.CharField(max_length=200, validators=[
        validators.MinLengthValidator(2, "Location must be at
least 2 characters long."),
        validators.MaxLengthValidator(200, "Location cannot
exceed 200 characters.")])
    ...
```

Solution: Restaurant

```
class Restaurant(models.Model):
    ...
    description = models.TextField(blank=True, null=True)
    rating = models.DecimalField(
        max_digits=3,
        decimal_places=2,
        validators=[
            validators.MinValueValidator(0.00, "Rating must be
at least 0.00"),
            validators.MaxValueValidator(5.00, "Rating cannot
exceed 5.00"),
        ])
```

Custom Validators

- Create **custom validators** when you need to implement a **custom validation logic**
- How to create a **custom validator**
  - Define a **function** that
    - **takes** the **field's value**
    - **applies** some **validation logic**
    - **raises** a `ValidationError`
      - if the **value** does **not meet** the **requirements**

Custom Validator Example

```
from django.core.exceptions import ValidationError

def validate_even(value):


    if value % 2 != 0:
        raise ValidationError('Value must be an even number!')


class MyModel(models.Model):
    number = models.IntegerField(validators=[validate_even])
    ...
```

A Custom function that accepts the field value

Some custom validation logic

Raises a ValidationError

Error message

Use the custom validator

Problem: Menu

- In the **`main_app`** create a model called **`"Menu"`**, with the fields: **`"name"`**, **`"description"`**, and **`"restaurant"`**
- A full description of the problem can be found in the Lab document here

Solution: Menu

main_app/validators.py

```
from django.core.exceptions import ValidationError

def validate_menu_categories(value):
    required_categories = ["Appetizers", "Main Course", "Desserts"]

    for category in required_categories:
        if category.lower() not in value.lower():
            raise ValidationError('The menu must include each of
the categories "Appetizers", "Main Course", "Desserts".')
```

Solution: Menu

main_app/models.py

```
from main_app.validators import validate_menu_categories

class Menu(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField(
        validators=[validate_menu_categories]
    )
    restaurant = models.ForeignKey(Restaurant,
                                   on_delete=models.CASCADE)
```

Meta Options & Meta Inheritance

Table name, Default order, Unique constraints

Meta Class and Meta Options

- **Meta class** allows you to provide **additional information** about a **model**
- Used to specify **model-level options** like
  - database **table name**
  - default **ordering**
  - **unique** constraints
  - **abstract**
  - **proxy**

Meta Options in Details

- `abstract`
  - If `True`, indicates that the model will be an **Abstract Base Class**
- `proxy`
  - If `True`, indicates that the model will be a **Proxy Model**
- `db_table`
  - **Overrides** the **default** database **table name** for the model
- `ordering`
  - Defines a **default order** when a collection of model objects is obtained

Meta Options in Details

- `unique_together`
  - Defines a **set** of **field names** that their values together must be unique
- `verbose_name`
  - Defines a **human-readable name** for the object (**singular**)
- `verbose_name_plural`
  - Defines a **plural name** for the object
  - By default, Django uses the **model** or **verbose name** (if given) + **"s"**

More at: *https://docs.djangoproject.com/en/5.0/ref/models/options/#model-meta-options*

Meta Options Example

```
class MyModel(models.Model):
    name = models.CharField(max_length=100)
    ...

    class Meta:
        db_table = 'custom_table_name'

        ordering = ['-name']

        unique_together = ['name', 'id']

        # A unique constraint will be applied to the combination of
"name" and "id" fields
```

Defines a custom name for the database table

Defines a default objects order

Defines a unique constraint

Problem: Restaurant Review

- In the **`main_app`** create an additional model called **`"RestaurantReview"`**, with the fields: **`"reviewer_name"`**, **`"restaurant"`**, **`"review_content"`**, and **`"rating"`**
- For this model, **define additional options**
- A full description of the problem can be found in the Lab document **here**

Solution: Restaurant Review

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

    class Meta:
        ordering = ['-rating']
        verbose_name = "Restaurant Review"
        verbose_name_plural = "Restaurant Reviews"
        unique_together = ['reviewer_name', 'restaurant']
```

Meta Inheritance

- **Meta inheritance** refers to the ability to **inherit Meta options**
  - From a **parent abstract model** to its **child models**
- When an **abstract** model (parent) is defined in Django
  - It can include an **inner Meta class**
    - where you specify various **options** related to the **behavior** and **configuration** of that **model**
  - It can be **subclassed** so the **child** will **inherit**
    - the **same Meta options** if it **does not define** its **own Meta class**

Meta Inheritance

- When you create a **child** model that **inherits** from an **abstract** model
  - It can also define its **own** **Meta class**
    - **Own Meta class** **completely overrides** the **parent's** one **unless** it **extends** it by **subclassing**
  - If the **child** model does **not** have its **own Meta class**
    - Django will look for the **Meta class** in the **parent abstract** model and **inherit** its **options**

Meta Inheritance Example

```
from django.db import models

class BaseModel(models.Model):
    name = models.CharField(max_length=100)

    class Meta:
        abstract = True
        ordering = ['name']

class ChildModel(BaseModel):
    description = models.TextField()

    # ChildModel inherits the Meta options
```

Meta options in the Abstract Base Class

Meta class has not been defined by the child

Meta Inheritance Example - Overriding

Software University

```
from django.db import models

class BaseModel(models.Model):
    name = models.CharField(max_length=100)

    class Meta:
        abstract = True
        ordering = ['name']

class ChildModel(BaseModel):
    description = models.TextField()

    class Meta:
        verbose_name_plural = 'Child Models'
        ordering = ['name']
```

Meta options in the Abstract Base Class

Child's Meta class overrides the inherited parent's one

Child's own Meta options

Meta Inheritance Example - Extending

Software University

```
from django.db import models

class BaseModel(models.Model):
    name = models.CharField(max_length=100)

    class Meta:
        abstract = True
        ordering = ['name']

class ChildModel(BaseModel):
    description = models.TextField()

    class Meta(BaseModel.Meta):
        verbose_name_plural = 'Child Models'
    # ChildModel inherits all parent's Meta options and adds its own ones
```

Meta options in the Abstract Base Class

Child's Meta class extends the parent's one by subclassing it

Child's extra Meta option

Problem: Restaurant Review Types

- We decided to differentiate **two types of restaurant reviews** - Regular Reviews and Food Critic Reviews
- In this case, we do not want the base **`"RestaurantReview"`** model to have a database table and save data on its own
- In the **`main_app`** create **two additional models** called **`"RegularRestaurantReview"`** and **`"FoodCriticRestaurantReview"`**
- A full description of the problem can be found in the Lab document **here**

Solution: Restaurant Review Types

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

    class Meta:
        ...
        abstract = True

class RegularRestaurantReview(RestaurantReview):
    ...
```

Solution: Restaurant Review Types

```
class FoodCriticRestaurantReview(RestaurantReview):
    food_critic_cuisine_area = models.CharField(max_length=100)

    class Meta(RestaurantReview.Meta):
        verbose_name = "Food Critic Review"
        verbose_name_plural = "Food Critic Reviews"
```

Meta Inheritance Summary

- **Meta Inheritance** is possible **only** if the **parent** is an **abstract base class**
- If the **child** defines its **own Meta class**, it **overrides** the **inherited** one from the **abstract parent**
  - **Unless** it **extends** the parent's **Meta class** by subclassing it
- When a model **inherits** from a **non-abstract parent** with a **Meta class**, the **child's Meta class** is **independent**, and it will **not inherit** or be **affected** by the **Meta options** of the **parent** model

Indexing

Indexing in Models

- In Django models, **indexing** is used to
  - **optimize** database **queries** for specific fields
- By adding an **index** to a **field**, you can
  - **speed up search operations** on that field
- Database uses **indexes** to locate rows **much faster**
  - significantly **reducing** the **time** to retrieve the data

Indexing in Models

- By **default**, the database creates an **index**
  - For the **primary key**
- It is **possible** to specify additional **indexes manually** on **other fields** by using
  - The `db_index` **attribute**
  - **Meta** class **option** `indexes`

Indexing Example

```
from django.db import models

class MyModel(models.Model):
    title = models.CharField(max_length=200, db_index=True)
    author = models.CharField(max_length=100)
    publication_date = models.DateField(db_index=True)
    genre = models.CharField(max_length=50)
```

Additional index on a field

Additional index on a field

Indexing Example – Meta Option

```
from django.db import models

class MyModel(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    publication_date = models.DateField()
    genre = models.CharField(max_length=50)

    class Meta:
        indexes=[
models.Index(fields=["title", "author"]),
models.Index(fields=["publication_date"])
        ]
```

A list of indexes to define on the model

A composite index on both fields

A single-field index

Problem: Menu Review

- In the **`main_app`** create an additional model called **`"MenuReview"`**, with the fields: **`"reviewer_name"`**, **`"menu"`**, **`"review_content"`**, and **`"rating"`**
- For this model, **define additional options**
- A full description of the problem can be found in the Lab document **here**

Solution: Menu Review

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

    class Meta:
        ordering = ['-rating']
        verbose_name = "Menu Review"
        verbose_name_plural = "Menu Reviews"
        unique_together = ['reviewer_name', 'menu']
        indexes = [models.Index(
            fields=["menu"],
            name="main_app_menu_review_menu_id")]
```

Django Model Mixins

Model Mixins

- **Mixins** are a way to **extend** the functionality of Django **models**
  - by creating **reusable** pieces of **code** that can be **mixed** into **multiple models**
- A **Mixin** is essentially a **Python class** that
  - contains additional **fields**, **methods**, or **behavior**
  - can be **combined** with **other** Django **models**
- By using **Mixins**, you can **avoid** code **duplication** and keep your models **clean** and **organized**

Model Mixins

- Model **Mixins** are **abstract classes** that can be added as a **parent class** of a **model**
- Python supports **multiple inheritances**
  - You can list **any number** of **parent classes** for a model
- **Mixins** have to be **simple** and **easily composable**
- **Smaller Mixins** are better
  - If a **Mixin** becomes large and **violates** the **single responsibility principle**, consider **refactoring** it into **smaller** classes
  - Let a **Mixin** do **one thing** and do it well

Model Mixin Example

```
class TimestampMixin(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True

class MyModel(TimestampMixin):
    name = models.CharField(max_length=100)
    ...
```

An abstract model Mixin class, defining two reusable fields

Child class inherits all fields from the Mixin class

Problem: Rating and Review Content

- As you can see the **rating** and the **review content** fields are part of each type of review
- Your task is to **create a reusable component** called **`"ReviewMixin"`** that can be **mixed** into the models **`"RestaurantReview"`** and **`"MenuReview"`** to add the review-related **fields `"rating"`** and **`"review_content"`** and their **validation rules**
- Do not forget to **add the common Meta options** of the models, related to the review parts

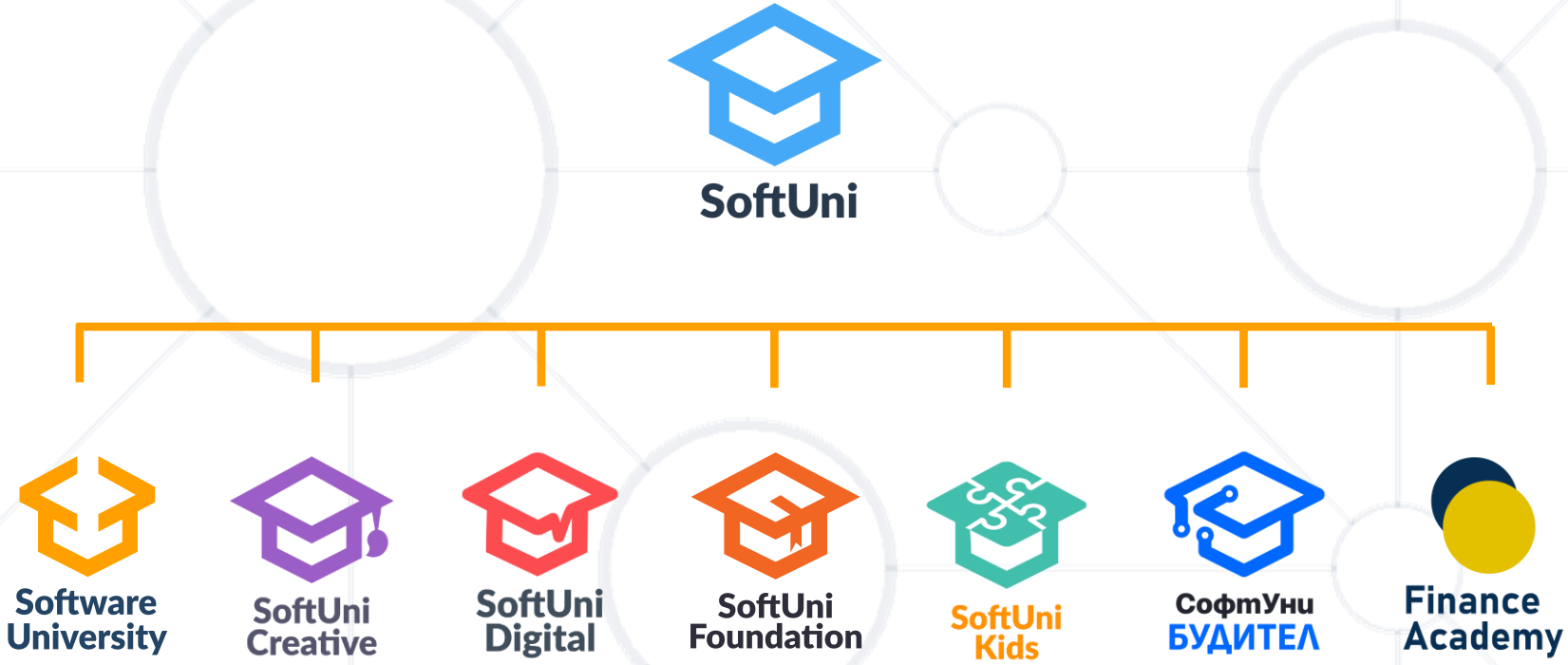
Solution: Rating and Review Content

```
class ReviewMixin(models.Model):
    rating = ...
    review_content = ...

    class Meta:
        abstract = True
        ordering = ['-rating']

class RestaurantReview(ReviewMixin):
    ...
    class Meta(ReviewMixin.Meta):...
class MenuReview(ReviewMixin):...
```

Summary

- Validation in Models
  - Built-in and Custom Validators
- Meta Options and Meta Inheritance
- Indexing
- Django Model Mixins

Questions?



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

- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg