django-cookiecutter

Release 1.0.0

godfrey-ndungu

May 11, 2023

CONTENTS:

Welcome to django-cookiecutter's documentation!

Django cookie cutter is a template or starting point for creating a new Django project with a set of features and components. This cookie cutter is designed for developers who want to start with a basic Django project structure and add their own custom features as needed.

CHAPTER

ONE

FEATURES:

The Django cookie cutter includes the following features:

- AWS S3
- PostgreSQL
- TOX
- Python Celery
- Redis
- Ansible deployment playbooks
- Local deployment makefile
- CircleCI
- GitLab CI for security configs
- Python Decouple for env
- Django Rest Framework (DRF)
- DRF-Swagger
- Docker
- Kubernetes

1.1 Important:

Please note that the Django cookie cutter is intended to provide a basic project structure that can be extended to meet your specific requirements.

CHAPTER

TWO

HOW TO

2.1 Set Up Development with django-cookiecutter

This guide will walk you through the steps to set up a development environment for the *django-cookiecutter* project.

Prerequisites

Before you begin, make sure you have the following software installed on your system:

- Git
- Python 3
- virtualenv

Step 1: Clone the Project

First, clone the project repository by running the following command:

```
git clone https://github.com/Godfrey-Ndungu/django-cookiecutter.git
cd django-cookiecutter
```

Step 2: Make the Makefile Executable

Next, make the Makefile executable by running the following command:

```
chmod +x makefile.sh
```

Step 3: Execute the Makefile

Now, execute the Makefile by running the following command:

```
./makefile.sh
```

This will install the following software on your system:

- Redis
- PostgreSQL
- Ansible
- Docker
- Docker Compose
- Python
- Setuptools

It will also create a virtual environment and install the project requirements.

2.2 Change the Django Project Name from "cookiecutter" to a Desired Name

This guide will walk you through the steps to change the Django project name from "cookiecutter" to a desired name.

Step 1: Rename the Project Directory

First, rename the project directory from "cookiecutter" to your desired name:

```
mv cookiecutter/ myproject/
```

Step 2: Update the Project Name in Files

Next, update the project name in the following files:

- *myproject/settings/base.py*: Change the *ROOT_URLCONF* to the new URL conf module.
- *myproject/wsgi.py*: Change the *DJANGO_SETTINGS_MODULE* to the new settings module.
- *myproject/asgi.py*: Change the *DJANGO_SETTINGS_MODULE* to the new settings module.
- *manage.py*: Change the *DJANGO_SETTINGS_MODULE* to the new settings module.
- *myproject/settings.py*: Change the *DEFAULT_FILE_STORAGE* to the new settings module.

Step 3: Update the Project Name in Docker and Docker Compose Files

If you are using Docker and Docker Compose, update the project name in the following files:

- *docker-compose.yml*: Change the *services.app.container_name* to the new container name.
- *docker-compose.prod.yml*: Change the *services.app.container_name* to the new container name.
- *Dockerfile*: Change the *WORKDIR* and *COPY* commands to reflect the new project directory name.

Step 4: Update the Project Name in AWS Configs

If you are using AWS, update the project name in the following files:

- *backend/settings.py*: Change the *ROOT_URLCONF* to the new URL conf module.

Step 5: Conclusion

You have now successfully changed the Django project name from "cookiecutter" to your desired name.

2.3 Customize AWS File Storage in cookiecutter.storagebackend using boto3

This guide will walk you through the steps to customize AWS file storage in *cookiecutter.storagebackend* using *boto3*.

Step 1: Install Boto3

First, make sure you have *boto3* installed in your project environment:

```
pip install boto3
```

Step 2: Configure AWS Credentials

Next, configure your AWS credentials in your environment variables or AWS configuration files.

Step 3: Customize cookiecutter.storagebackend

Now, customize *cookiecutter.storagebackend* to use AWS S3 file storage by modifying the *backend.py* file as follows:

```
import boto3
from django.core.files.storage import get_storage_class

s3 = boto3.resource('s3')
storage_class = get_storage_class('storages.backends.s3boto3.S3Boto3Storage')

class CustomS3Storage(storage_class):
    def __init__(self, *args, **kwargs):
        kwargs['bucket'] = 'my-bucket-name'  # Replace with your S3 bucket name
        super().__init__(*args, **kwargs)

    def _save(self, name, content):
        """
        Save a file to AWS S3.
        """
        self._create_bucket_if_not_exists()
        return super()._save(name, content)

    def _create_bucket_if_not_exists(self):
        """
        Create an S3 bucket if it does not already exist.
        """
        if self.bucket_name not in [bucket.name for bucket in s3.buckets.all()]:
            s3.create_bucket(Bucket=self.bucket_name)

DEFAULT_FILE_STORAGE = 'path.to.CustomS3Storage'
```

Step 4: Test the Custom Storage Backend

Finally, test the custom storage backend by uploading a file to S3 using Django's *default_storage*:

```
from django.core.files.storage import default_storage

file = open('/path/to/file.jpg', 'rb')
default_storage.save('file.jpg', file)
```

2.4 Change Celery Task Name in cookiecutter.celery and Updating Docker Compose celery_worker Command

This guide will walk you through the steps to change the Celery task name in *cookiecutter.celery* and update Docker Compose *celery_worker* command.

Step 1: Rename the Celery Task Name

First, rename the Celery task name from *tasks.example* to your desired name in *celery.py* file:

```
app.conf.task_default_queue = 'default'
app.conf.task_default_exchange_type = 'topic'
app.conf.task_default_routing_key = 'default'

app.conf.task_routes = {
    'new_task': {'queue': 'new_task_queue', 'routing_key': 'new_task'},
    'example': {'queue': 'example_queue', 'routing_key': 'example'}
    # Change 'example' to your desired task name
}
```

Step 2: Update Docker Compose celery_worker Command

Next, update the Docker Compose *celery_worker* command in *docker-compose.yml* file to use the new Celery task name:

```
version: '3'

services:
  app:
    build:
      context: .
    command: >
      sh -c "python manage.py migrate &&
      python manage.py runserver 0.0.0.0:8000"
    ports:
      - "8000:8000"
    depends_on:
      - db
      - redis
      - celery_worker
    environment:
      - DB_HOST=db
      - REDIS_URL=redis://redis:6379/0

  celery_worker:
    build:
      context: .
    command: >
      sh -c "celery -A myproject worker -l info -Q new_task_queue,example_queue"  #␣
↪Change 'example' to your new task name
    depends_on:
      - db
      - redis
    environment:
      - DB_HOST=db
      - REDIS_URL=redis://redis:6379/0

  db:
    image: postgres:12-alpine
    environment:
      POSTGRES_USER: myproject
      POSTGRES_PASSWORD: myproject
      POSTGRES_DB: myproject
```

(continues on next page)

(continued from previous page)

```
  redis:
    image: redis:5-alpine
```

Step 3: Restart Docker Compose Services

Finally, restart the Docker Compose services to apply the changes:

```
docker-compose down
docker-compose up --build
```

2.5 Replace Redis with RabbitMQ

This guide will walk you through the steps to replace Redis with RabbitMQ as the message broker in your Django project.

Step 1: Install RabbitMQ

First, you need to install RabbitMQ. You can follow the official documentation for installation instructions: https://www.rabbitmq.com/download.html

Step 2: Install the Required Packages

Next, you need to install the required packages to use RabbitMQ as the message broker. You can install the *pika* package with pip:

```
pip install pika
```

Step 3: Update the Celery Configuration

Update the Celery configuration in *celery.py* file to use RabbitMQ as the message broker:

```
from kombu import Exchange, Queue

broker_url = 'amqp://guest:guest@localhost:5672//' # Change this to your RabbitMQ URL

task_queues = (
    Queue('default', Exchange('default'), routing_key='default'),
    Queue('new_task_queue', Exchange('new_task'), routing_key='new_task'),
)

task_routes = {
    'new_task': {'queue': 'new_task_queue', 'routing_key': 'new_task'},
}
```

Step 4: Update the Docker Compose Configuration

Update the Docker Compose configuration in *docker-compose.yml* file to use RabbitMQ instead of Redis:

```
version: '3'

services:
  app:
    build:
```

(continues on next page)

(continued from previous page)

```
    context: .
  command: >
    sh -c "python manage.py migrate &&
    python manage.py runserver 0.0.0.0:8000"
  ports:
    - "8000:8000"
  depends_on:
    - db
    - rabbitmq
    - celery_worker
  environment:
    - DB_HOST=db
    - BROKER_URL=amqp://guest:guest@rabbitmq:5672// # Change this to your RabbitMQ URL

celery_worker:
  build:
    context: .
  command: >
    sh -c "celery -A myproject worker -l info -Q new_task_queue"
  depends_on:
    - db
    - rabbitmq
  environment:
    - DB_HOST=db
    - BROKER_URL=amqp://guest:guest@rabbitmq:5672// # Change this to your RabbitMQ URL

db:
  image: postgres:12-alpine
  environment:
    POSTGRES_USER: myproject
    POSTGRES_PASSWORD: myproject
    POSTGRES_DB: myproject

rabbitmq:
  image: rabbitmq:3.9-alpine
```

Step 5: Restart Docker Compose Services

Finally, restart the Docker Compose services to apply the changes:

```
docker-compose down
docker-compose up --build
```

2.6 Customize Django Rest Errors

This guide will walk you through the steps to customize the error handling in Django Rest using a custom exception handler.

Step 1: Create a Custom Exception Handler

Create a new file *custom_error_handlers.py* in your *core/views/* directory, and add the following code:

```
from django.core.exceptions import ObjectDoesNotExist
from rest_framework.exceptions import PermissionDenied
from rest_framework.views import exception_handler
from rest_framework.response import Response
from rest_framework import status

def custom_exception_handler(exc, context):
    """
    Custom exception handler to handle PermissionDenied and ObjectDoesNotExist␣
↪exceptions.
    """
    if isinstance(exc, PermissionDenied):
        return Response({"detail": "You do not have permission to perform this action."},
↪ status=status.HTTP_403_FORBIDDEN)

    if isinstance(exc, ObjectDoesNotExist):
        return Response({"detail": "The requested resource does not exist."},␣
↪status=status.HTTP_404_NOT_FOUND)

    # Let DRF handle other exceptions
    response = exception_handler(exc, context)

    if response is not None and response.status_code == status.HTTP_500_INTERNAL_SERVER_
↪ERROR:
        # Customize error message for 500 errors
        response.data = {"detail": "Internal server error occurred."}

    return response
```

Step 2: Register the Custom Exception Handler

Add the *custom_exception_handler* to the *EXCEPTION_HANDLER* setting in your Django settings file:

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'core.views.custom_error_handlers.custom_exception_handler'
}
```

Step 3: Test the Custom Exception Handler

Test the custom exception handler by triggering an exception in your Django Rest API. For example, if you try to access a non-existent object, you should see the custom error message "The requested resource does not exist."

2.7 Add Logs

Logging is a crucial aspect of any software system, as it allows developers to keep track of what's happening in their code and quickly diagnose and fix issues when they arise. The *logging* module in Python provides a powerful and flexible way to manage logging in your application.

Step 1: Import the logging module

Start by importing the *logging* module in your Python file:

```
import logging
```

Step 2: Get a logger instance

To use the *logging* module, you first need to get an instance of the logger. You can create a logger with a specific name by calling the *getLogger()* method on the *logging* module:

```
db_logger = logging.getLogger('db')
```

Here, we're creating a logger with the name 'db', which we'll use to log messages related to our database operations.

Step 3: Log messages

Once you have a logger instance, you can use it to log messages at different levels of severity, including *debug*, *info*, *warning*, *error*, and *critical*. Here's an example of logging an *info* message:

```
db_logger.info('info message')
```

You can also log *warning* messages:

```
db_logger.warning('warning message')
```

Step 4: Log exceptions

In addition to logging messages, you can also log exceptions using the *exception()* method of the logger. Here's an example:

```
try:
    1/0
except Exception as e:
    db_logger.exception(e)
```

This code will log the exception message and stack trace at the *ERROR* level.

Step 5: Configure logging settings

By default, the *logging* module will write log messages to the console. However, you can customize the logging settings by configuring a logging handler. For example, you can write log messages to a file or send them to a remote server. You can also customize the log format and level.

2.8 Change the database in *cookiecutter* settings and development files from PostgreSQL

cookiecutter is a popular tool for creating templates for Python projects. By default, it comes with settings and development files that use PostgreSQL as the database. However, you may want to use a different database, such as MySQL or SQLite. In this guide, we'll show you how to change the database in the *cookiecutter* settings and development files.

Step 1: Open the *settings* file

The first step is to open the *settings* file in your *cookiecutter* project. This file is located in the *config* directory of your project. You can open it in your favorite text editor.

Step 2: Change the database settings

In the *settings* file, you'll find a section that specifies the database settings. By default, it looks like this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'cookiecutter',
        'USER': 'cookiecutter',
        'PASSWORD': 'cookiecutter',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

To change the database, you'll need to modify the *ENGINE*, *NAME*, *USER*, *PASSWORD*, *HOST*, and *PORT* settings to match your desired database. For example, if you want to use MySQL, you would change the *ENGINE* setting to *'django.db.backends.mysql'* and modify the other settings accordingly.

Step 3: Open the *development* file

Next, open the *development* file in your *cookiecutter* project. This file is located in the root directory of your project.

Step 4: Change the database settings

In the *development* file, you'll find a section that specifies the database settings. By default, it looks like this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'cookiecutter',
        'USER': 'cookiecutter',
        'PASSWORD': 'cookiecutter',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

To change the database, you'll need to modify the *ENGINE*, *NAME*, *USER*, *PASSWORD*, *HOST*, and *PORT* settings to match your desired database. For example, if you want to use MySQL, you would change the *ENGINE* setting to *'django.db.backends.mysql'* and modify the other settings accordingly.

Step 5: Save your changes

Once you've made your changes to the *settings* and *development* files, save your changes and exit your text editor.

Step 6: Test your changes

Finally, you'll want to test your changes to make sure everything is working as expected. You can run your *cookiecutter* project as you normally would and verify that your database is being used correctly.

2.9 Customize Ansible Variables

Ansible is an open-source automation tool that can be used for a wide range of tasks, including configuration management, deployment, and orchestration. The *django-cookiecutter* project includes an Ansible playbook that can be used to set up your development environment. In this guide, you'll learn how to customize the Ansible variables in this playbook to suit your needs.

Step 1: Modify the PostgreSQL variables

The Ansible playbook includes variables for setting up a PostgreSQL database and user. You can modify these variables to use your own PostgreSQL database and user. Open the *ansible/vars/main.yml* file and modify the following variables:

- *postgres_database*: Change this variable to the name of your desired PostgreSQL database.
- *postgres_user*: Change this variable to the name of your desired PostgreSQL user.
- *postgres_password*: Change this variable to the password for your PostgreSQL user.

For example, to use a PostgreSQL database named *mydb* and a user named *myuser* with the password *mypassword*, you would modify the variables as follows:

```
postgres_database: mydb
postgres_user: myuser
postgres_password: mypassword
```

Step 2: Modify the project name

The *project_name* variable in the *ansible/vars/main.yml* file specifies the name of your project. You can modify this variable to use your own project name:

```
project_name: myproject
```

Step 3: Modify the Docker Compose version

The *docker_compose_version* variable in the *ansible/vars/main.yml* file specifies the version of Docker Compose that will be installed. You can modify this variable to use a different version of Docker Compose:

```
docker_compose_version: 1.29.2
```

Step 4: Modify the Nginx template file

The *ansible/roles/nginx/tasks/main.yml* file includes a task for configuring Nginx. This task uses a Jinja2 template to generate the Nginx configuration file. You can modify the *src* variable in this task to point to the template file for your own Nginx configuration.

For example, if you've created a custom Nginx configuration file called *myproject.conf.j2*, you would modify the task as follows:

```
- name: Generate Nginx configuration file
  template:
    src: "path/to/myproject.conf.j2"
    dest: "/etc/nginx/sites-available/{{ project_name }}"
```

2.10 Adding Items to the Docs/Source Folder

To add items to the *docs/source* folder, follow these steps:

1. Navigate to the *docs/source* folder in your project directory.
2. Create a new file for your item using a descriptive name, such as *my_item.rst*.
3. Open the file in a text editor and add your content. Use the reStructuredText syntax to format your content, and be sure to include a title for your item.
4. Save the file and commit it to your project's version control system.
5. If you want your item to appear in the table of contents, update the *index.rst* file in the *docs/source* folder. Add a new entry for your item using the following format:

   Replace *My Item* with the title of your item, and *my_item* with the name of the file you created in step 2.
6. Build the documentation using Sphinx. You can do this by running the following command from the root of your project directory:

CHAPTER

THREE

CORE

The apps.core.models module contains several models that are used across the application. In this document, we will describe the TrackableModel, TimestampedModel, and Task models.

3.1 TrackableModel

The TrackableModel is an abstract model that provides fields to automatically track changes and keep a record of who made them. It has the following attributes:

- **created_at (DateTimeField)**: The timestamp when the model instance was first created.
- **updated_at (DateTimeField)**: The timestamp of the most recent update to the model instance.
- **created_by (ForeignKey)**: The user who created the model instance.
- **updated_by (ForeignKey)**: The user who last updated the model instance.
- **history (JSONField)**: A JSON field that stores a complete history of changes to the model instance, including timestamps and the user who made each change.

To use this model, you can create a new model and inherit from it:

```
from django.db import models
from apps.accounts.models import TrackableModel

class MyModel(TrackableModel):
   my_field = models.CharField(max_length=255)
```

3.2 TimestampedModel

The TimestampedModel is an abstract model that builds on the functionality of TrackableModel by adding extra fields for timestamping. It has the following attributes:

- **date_added (DateTimeField)**: The timestamp when the model instance was first added.
- **date_updated (DateTimeField)**: The timestamp of the most recent update to the model instance.
- **date_deleted (DateTimeField)**: The timestamp of when the model instance was deleted (if applicable).

To use this model, you can create a new model and inherit from it:

```
from django.db import models
from apps.core.models import TimestampedModel

class MyModel(TimestampedModel):
   my_field = models.CharField(max_length=255)
```

3.3 Task

The Task model is used for tracking tasks. It has the following attributes:

- **name (str)**: The name of the task.
- **task_ingestor (str)**: The ingestor responsible for the task.
- **datetime (datetime)**: The date and time the task was created.
- **status (str)**: The current status of the task. Allowed values are 'pending', 'processing', 'processed', and 'failed'.

It also has the following methods:

- **start_processing()**: Transition the task from 'pending' to 'processing'.
- **complete_processing()**: Transition the task from 'processing' to 'processed' and delete the record.
- **fail_processing()**: Transition the task from 'processing' to 'failed'.

To use this model, you can create a new model and inherit from it:

```
from django.db import models
from apps.core.models import Task

class MyTaskModel(Task):
 my_field = models.CharField(max_length=255)
```

3.4 apps.core.models

class apps.core.models.**Task**(**args*, ***kwargs*)

Bases: Model

A model for tracking tasks.

name

The name of the task.

Type
str

task_ingestor

The ingestor responsible for the task.

Type
str

datetime

The date and time the task was created.

Type
datetime

status

The current status of the task. Allowed values are 'pending', 'processing', 'processed', and 'failed'.

Type
str

exception DoesNotExist

Bases: ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

TASK_STATUS_CHOICES = (('pending', 'Pending'), ('processing', 'Processing'), ('processed', 'Processed'), ('failed', 'Failed'))

TASK_STATUS_FAILED = 'failed'

TASK_STATUS_PENDING = 'pending'

TASK_STATUS_PROCESSED = 'processed'

TASK_STATUS_PROCESSING = 'processing'

complete_processing()

Transition the task from 'processing' to 'processed' and delete the record.

datetime

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

fail_processing()

Transition the task from 'processing' to 'failed'.

get_next_by_datetime(*, *field=<django.db.models.fields.DateTimeField: datetime>*, *is_next=True*, ***kwargs*)

get_previous_by_datetime(*, *field=<django.db.models.fields.DateTimeField: datetime>*, *is_next=False*, ***kwargs*)

get_status_display(*, *field=<django.db.models.fields.CharField: status>*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

save(**args*, ***kwargs*)

Save the task

start_processing()

Transition the task from 'pending' to 'processing'.

status

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

task_ingestor

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class apps.core.models.**TimestampedModel**(**args*, ***kwargs*)

Bases: TrackableModel

Abstract model that builds on the functionality of
TrackableModel by adding extra fields for timestamping.

date_added

The timestamp when the model instance was first added.

Type
DateTimeField

date_updated

The timestamp of the most recent update to the model instance.

Type
DateTimeField

date_deleted

The timestamp of when the model instance was deleted (if applicable).

Type
DateTimeField

class Meta

Bases: object

abstract = False

created_by

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Child.parent is a ForwardManyToOneDescriptor instance.

date_added

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_deleted

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_updated

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_date_added(*, *field=<django.db.models.fields.DateTimeField: date_added>*, *is_next=True*, ***kwargs*)

get_next_by_date_updated(*, *field=<django.db.models.fields.DateTimeField: date_updated>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_date_added(*, *field=<django.db.models.fields.DateTimeField: date_added>*, *is_next=False*, ***kwargs*)

get_previous_by_date_updated(*, *field=<django.db.models.fields.DateTimeField: date_updated>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

updated_by

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

class apps.core.models.**TrackableModel**(**args*, ***kwargs*)

Bases: `Model`

Abstract model that provides fields to automatically track changes and keep a record of who made them.

created_at

The timestamp when the model instance was first created.

Type
DateTimeField

updated_at

The timestamp of the most recent update to the model instance.

Type
DateTimeField

created_by

The user who created the model instance.

Type
ForeignKey

updated_by

The user who last updated the model instance.

Type
ForeignKey

history

A JSON field that stores a complete history of changes to the model instance, including timestamps and the user who made each change.

Type
JSONField

class Meta

Bases: `object`

abstract = False

created_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

created_by

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

created_by_id

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

updated_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

updated_by

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

updated_by_id

CHAPTER

FOUR

ACCOUNTS

4.1 Models

4.1.1 CustomUserManager and CustomUser

The **CustomUserManager** and **CustomUser** models in the **apps.accounts.models** file are designed to extend the default Django user model with additional functionality and customizations.

To understand how these models work, let's start by examining the **CustomUserManager** model. This model is responsible for creating and managing instances of the **CustomUser** model. It overrides the default Django BaseUserManager to provide custom functionality. Specifically, the **CustomUserManager** model provides two additional methods:

- **create_user()**: This method creates a new user instance using the provided email and password. It sets the is_staff and is_superuser attributes to False, indicating that this is a regular user account.
- **create_superuser()**: This method creates a new user instance with the provided email and password, but sets the is_staff and is_superuser attributes to True. This indicates that this is an administrator account with elevated privileges.

The CustomUser model itself extends the default Django **AbstractBaseUser** and **PermissionsMixin** models to provide additional customizations. It defines several fields, including email, first_name, and last_name. It also defines two boolean fields, is_active and is_staff, which determine whether the user account is currently active and whether the user has staff permissions, respectively.

To use these models in your Django project, you'll first need to import **CustomUSer** into your file. You can do this using the following code:

```
from apps.accounts.models  import CustomUser

user = CustomUser.objects.create_user(email = 'user@example.com', password = 'password*␣
↪**23')
admin = CustomUser.objects.create_superuser(email = 'admin@example.com', password =
↪'password* **23')
```

4.1.2 UserVisitHistory

The **UserVisitHistory** model in the **apps.accounts.models** module is used to store the visit history of users in your application. This model is particularly useful if you want to keep track of which pages or sections of your app users are visiting, how often they visit, and when they last visited.

The structure of the UserVisitHistory model consists of the following fields:

- **user**: A foreign key to the user who made the visit.
- **timestamp**: The date and time of the visit.
- **url**: The URL of the page visited.
- **referer**: The URL of the referring page, if any.
- **user_agent**: The user agent string for the browser or other client used to make the visit.

To use these models in your Django project, you'll first need to import **CustomUSer** into your file. You can do this using the following code:

```
from apps.accounts.models import UserVisitHistory

user = CustomUser.objects.get(id=* **)
UserVisitHistory.objects.create(
    user=user,
    url='/example',
    referer=None,
    user_agent='Mozilla/5.0 (Windows NT * **0.0; Win64; x64) AppleWebKit/537.36 (KHTML,␣
↪like Gecko) Chrome/58.0.3029.* **0 Safari/537.3'
)
```

To retrieve user visit history records, you can use the objects manager of the UserVisitHistory model:

```
from django.utils import timezone
from apps.accounts.models import CustomUSer
from apps.accounts.models import UserVisitHistory

# Get all user visit history records for a specific user
user = CustomUser.objects.get(id=* **)
visit_history = UserVisitHistory.objects.filter(user=user)

# Get all user visit history records for a specific URL
visit_history = UserVisitHistory.objects.filter(url='/example')

# Get all user visit history records for a specific time range
start_time = timezone.now() - timezone.timedelta(days=7)
end_time = timezone.now()
visit_history = UserVisitHistory.objects.filter(timestamp__range=(start_time, end_time))
```

4.1.3 LoginHistoryTrail

LoginHistoryTrail is a model that stores a trail of login attempts made by users. It is a part of the apps.accounts.models module. This model has the following fields:

- **user**: A foreign key to the user who made the login attempt.
- **timestamp**: The date and time of the login attempt.
- **successful**: A boolean field indicating whether the login attempt was successful or not.
- **ip_address**: The IP address used to make the login attempt.
- **user_agent**: The user agent string for the browser or other client used to make the login attempt.
- **location**: The location (city, country) of the IP address used to make the login attempt, if available.

To use this model in your Django project, you need to follow these steps:

```
from apps.accounts.models import LoginHistoryTrail

LoginHistoryTrail.objects.create(user=user, ip_address=ip_address, user_agent=user_agent,
↪ location=location, successful=successful)
```

To retrieve login history trail records, you can use the objects attribute of the LoginHistoryTrail model:

```
from apps.accounts.models import LoginHistoryTrail

LoginHistoryTrail.objects.create(user=user, ip_address=ip_address, user_agent=user_agent,
↪ location=location, successful=successful)
```

4.1.4 LoginAttemptHistory

The **LoginAttemptHistory** model is a Django model used to store a history of login attempts made by users . This model is located in the **apps.accounts.models**

Fields:

- **user**: A foreign key to the CustomUser model representing the user who made the login attempt.
- **timestamp**: The date and time of the login attempt, automatically generated when a new instance is created.
- **successful**: A boolean field indicating whether the login attempt was successful or not.
- **ip_address**: The IP address used to make the login attempt, stored as a GenericIPAddressField.
- **user_agent**: The user agent string for the browser or other client used to make the login attempt, stored as a TextField.
- **location**: The location (city, country) of the IP address used to make the login attempt, if available, stored as a CharField.

Usage: To use the LoginAttemptHistory model, you can import it in any Django file using the from apps.accounts.models import LoginAttemptHistory statement.

To create a new LoginAttemptHistory instance, you can call its constructor and pass the necessary arguments as follows:

```
from apps.accounts.models import LoginAttemptHistory
from apps.accounts.models import CustomUser
from django.utils import timezone
```

(continues on next page)

(continued from previous page)

```
user = CustomUser.objects.get(email='example@example.com')
ip_address = '* **27.0.0.* **'
user_agent = 'Mozilla/5.0 (Windows NT * **0.0; Win64; x64) AppleWebKit/537.36 (KHTML,␣
↪like Gecko) Chrome/58.0.3029.* **0 Safari/537.3'

# create new instance
login_attempt = LoginAttemptHistory(
    user=user,
    successful=False,
    ip_address=ip_address,
    user_agent=user_agent,
    location=None
)

# save instance to database
login_attempt.save()
```

To retrieve all login attempts for a particular user, you can use Django's related manager:

```
from apps.accounts.models import CustomUser
from apps.accounts.models import LoginAttemptHistory

user = CustomUser.objects.get(email='example@example.com')
login_attempts = user.loginattempthistory_set.all()
```

To retrieve all successful login attempts for a particular user:

```
from apps.accounts.models import CustomUser
from apps.accounts.models import LoginAttemptHistory

user = CustomUser.objects.get(email='example@example.com')
successful_login_attempts = user.loginattempthistory_set.filter(successful=True)
```

4.1.5 ExtraData

The **ExtraData** model is used to store additional information related to user activity, such as browser information, IP address, device details, operating system, and location. This information can be useful for tracking user activity and analyzing user behavior on your website or application.

Fields

- **user (ForeignKey)**: A foreign key to the CustomUser model, indicating which user this extra data belongs to.
- **timestamp (DateTimeField)**: A date and time field indicating when this extra data was recorded. This field is set to auto_now_add, meaning it will automatically be set to the current date and time when a new record is created.
- **browser (CharField)**: A string field that stores the user's browser information.
- **ip_address (GenericIPAddressField)**: A field that stores the user's IP address. This field automatically validates the input to ensure it is a valid IP address.
- **device (CharField)**: A string field that stores the user's device details.

- **os (CharField)**: A string field that stores the user's operating system.
- **location (CharField)**: A string field that stores the user's location information.

To use the ExtraData model, you can create a new record whenever you want to store additional information related to user activity.

```
from apps.accounts.models import ExtraData
from apps.accounts.models import CustomUser

# Assume that we have a user object representing the current user
current_user = CustomUser.objects.get(id=1)

# Create a new ExtraData record
extra_data = ExtraData.objects.create(
    user=current_user,
    browser='Chrome',
    ip_address='192.168.0.1',
    device='Desktop',
    os='Windows 10',
    location='New York'
)

# The timestamp field will be set automatically by the auto_now_add argument in the␣
↪model definition.
```

4.1.6 OTP

The **OTP** model is used to store one-time passwords (OTPs) associated with a user in the CustomUser model. This model is defined in the apps.accounts.models module.

Fields

- **user (ForeignKey)**: The user associated with the OTP. This field is required.
- **code (CharField)**: The OTP code. This field is unique and has a maximum length of 4 characters. This field is required.
- **active (BooleanField)**: A boolean indicating whether the OTP is active or not. By default, this field is set to True.
- **created_at (DateTimeField)**: The date and time when the OTP was created. This field is automatically set when the OTP is created.
- **updated_at (DateTimeField)**: The date and time when the OTP was last updated. This field is automatically set when the OTP is saved.

Methods

- **create()**: Creates a new OTP for the given user and returns it.
- **get_latest()**: Gets the latest active OTP for the given user or returns None.
- **is_valid()**: Checks whether or not the OTP is valid (i.e. active and not expired) and returns a boolean value.
- **save()**: Saves the OTP to the database after validating the code is a 4-digit number or raises a ValidationError.

To create a new OTP for a user, call the OTP.create() method and pass in the user as an argument:

```
from accounts.models import CustomUSer
from accounts.models import OTP

user = CustomUser.objects.get(pk=1)
otp = OTP.create(user)
```

To get the latest active OTP for a user, call the **OTP.get_latest()** method and pass in the user as an argument:

```
from accounts.models import CustomUSer
from accounts.models import OTP

user = CustomUser.objects.get(pk=1)
otp = OTP.get_latest(user)
```

To check if an OTP is valid, call the is_valid() method on an instance of the OTP model:

```
if otp.is_valid():
    # The OTP is valid
else:
    # The OTP is invalid
```

4.2 apps.accounts.models

class apps.accounts.models.**CustomUser**(**args*, ***kwargs*)

Bases: `AbstractBaseUser`, `PermissionsMixin`

A custom user model that extends Django's built-in User model.

Fields:

email: The user's email address (unique). phone_number: The user's phone number (optional). is_active: Whether the user account is active. is_staff: Whether the user is a member of the staff. is_superuser: Whether the user has all permissions. date_joined: The date and time the user account was created.

USERNAME_FIELD

The field to use for authentication (email in this case).

REQUIRED_FIELDS

A list of required fields for creating a user.

__str__()

Returns the user's email address.

Managers:

objects: The manager for this model.

Meta:

verbose_name: A human-readable name for this model (singular). verbose_name_plural: A human-readable name for this model (plural).

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

REQUIRED_FIELDS = []

USERNAME_FIELD = 'email'

date_joined

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

email

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

extradata_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

get_next_by_date_joined(*, *field*=<*django.db.models.fields.DateTimeField: date_joined*>, *is_next*=*True*, ***kwargs*)

get_previous_by_date_joined(*, *field*=<*django.db.models.fields.DateTimeField: date_joined*>, *is_next*=*False*, ***kwargs*)

groups

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

is_staff

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

logentry_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

loginattemptshistory_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

loginhistorytrail_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

objects = <apps.accounts.models.CustomUserManager object>

otp_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

phone_number

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user_permissions

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

Pizza.toppings and Topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

uservisithistory_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class apps.accounts.models.**CustomUserManager**(**args*, ***kwargs*)

Bases: `BaseUserManager`

Custom user model manager where email is the unique identifier for authentication instead of username.

create_superuser(*email*, *password=None*, ***extra_fields*)

Create and save a SuperUser with the given email and password.

create_user(*email*, *password=None*, ***extra_fields*)

Create and save a User with the given email and password.

class apps.accounts.models.**ExtraData**(**args*, ***kwargs*)

Bases: `Model`

Model for storing extra data related to user activity, such as browser, IP address, device, operating system, and location.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

browser

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

device

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

get_next_by_timestamp(**, field=<django.db.models.fields.DateTimeField: timestamp>, is_next=True, **kwargs*)

get_previous_by_timestamp(**, field=<django.db.models.fields.DateTimeField: timestamp>, is_next=False, **kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

ip_address

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

location

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

os

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

timestamp

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_id

class apps.accounts.models.**LoginAttemptsHistory**(**args*, ***kwargs*)

Bases: `Model`

Model to store a history of login attempts made by users.

Fields:

user: A foreign key to the user who made the login attempt. timestamp: The date and time of the login attempt. successful: Whether the login attempt was successful. ip_address: The IP address used to make the login attempt. user_agent: The user agent string for the browser or

other client used to make the login attempt.

location: The location (city, country) of the IP

address used to make the login attempt, if available.

Meta:

verbose_name_plural: A human-readable name for this model (plural). ordering: The default ordering for querysets of this model,

by timestamp in descending order.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

get_next_by_timestamp(*, *field=<django.db.models.fields.DateTimeField: timestamp>*, *is_next=True*, ***kwargs*)

get_previous_by_timestamp(*, *field=<django.db.models.fields.DateTimeField: timestamp>*, *is_next=False*, ***kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

ip_address

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

location

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

successful

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

timestamp

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_agent

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user_id

class apps.accounts.models.**LoginHistoryTrail**(**args*, ***kwargs*)

Bases: `Model`

Model to store a trail of login attempts made by users.

Fields:

user: A foreign key to the user who made the login attempt. timestamp: The date and time of the login attempt. successful: Whether the login attempt was successful. ip_address: The IP address used to make the login attempt. user_agent: The user agent string for the browser or other client

used to make the login attempt.

location: The location (city, country) of the IP address used
to make the login attempt, if available.

Meta:

verbose_name_plural: A human-readable name for this model (plural). ordering: The default ordering for querysets of this model,

by timestamp in descending order.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

get_next_by_timestamp(*, *field=<django.db.models.fields.DateTimeField: timestamp>*, *is_next=True*, ***kwargs*)

get_previous_by_timestamp(*, *field=<django.db.models.fields.DateTimeField: timestamp>*, *is_next=False*, ***kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

ip_address

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

location

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

successful

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

timestamp

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_agent

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user_id

class apps.accounts.models.**OTP**(**args*, ***kwargs*)

Bases: Model

Model for storing one-time passwords (OTPs) associated with a user.

exception DoesNotExist

Bases: ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

active

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

code

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

classmethod create(*user*)

Creates a new OTP for the given user.

Parameters
user (*CustomUser*) – The user associated with the new OTP.

Returns
The newly created OTP.

created_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

classmethod get_latest(*user*)

Gets the latest active OTP for the given user.

Parameters
user (*CustomUser*) – The user associated with the OTP.

Returns
The latest active OTP, or None if there are no active OTPs.

get_next_by_created_at(***, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(***, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(***, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(***, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

is_valid()

Checks whether or not the OTP is valid (i.e. active and not expired).

Returns

True if the OTP is valid, False otherwise.

objects = <django.db.models.manager.Manager object>

save(**args*, ***kwargs*)

Saves the OTP to the database.

Parameters

- ***args** –
- ****kwargs** –

Raises

ValidationError – If the code is not a 4-digit number.

updated_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_id

class apps.accounts.models.**UserVisitHistory**(**args*, ***kwargs*)

Bases: `Model`

Model to store the history of user visits to the site.

Fields:

user: A foreign key to the user who made the visit. timestamp: The date and time of the visit. url: The URL of the page visited. referer: The URL of the referring page, if any. user_agent: The user agent string for the browser

or other client used to make the visit.

Meta:

verbose_name_plural: A human-readable name for this model (plural). ordering: The default ordering for querysets of this model,

by timestamp in descending order.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

get_next_by_timestamp(*, *field=<django.db.models.fields.DateTimeField: timestamp>*, *is_next=True*, ***kwargs*)

get_previous_by_timestamp(*, *field=<django.db.models.fields.DateTimeField: timestamp>*, *is_next=False*, ***kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

referer

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

timestamp

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

url

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_agent

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

user_id

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search