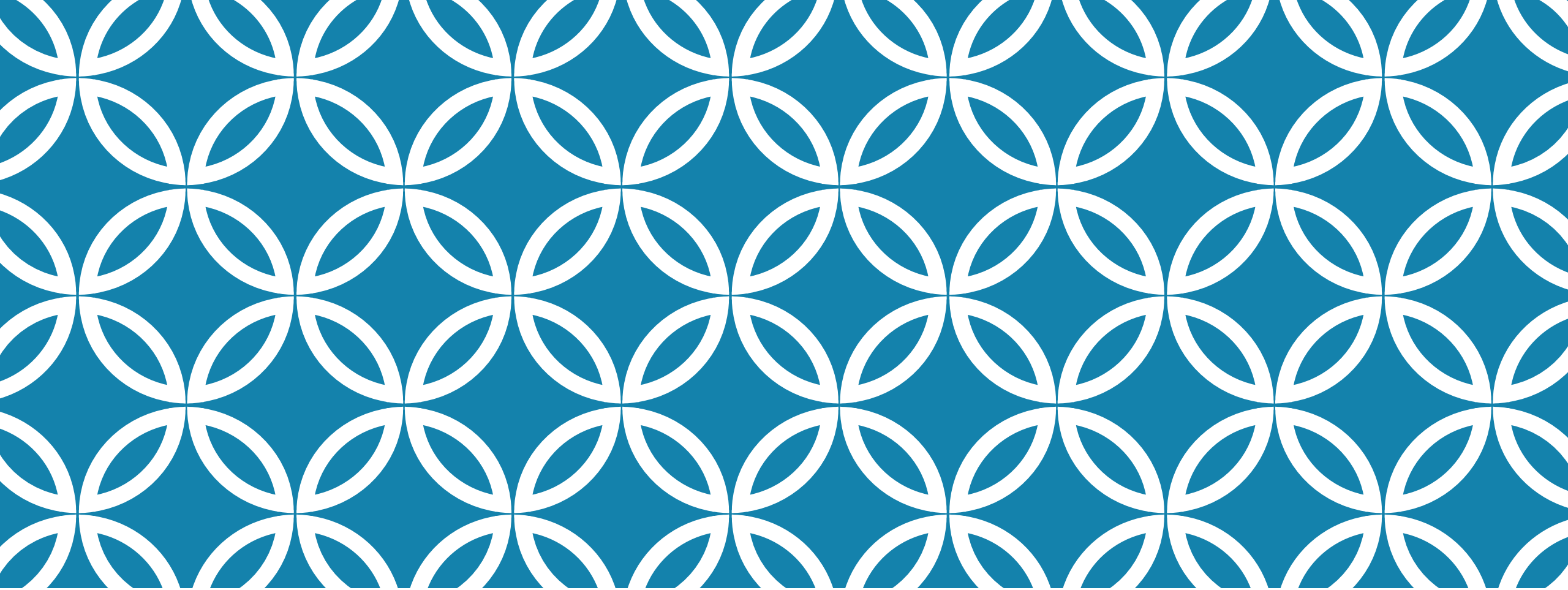


СОЗДАНИЕ ПЕРВОГО ПРОЕКТА

ДОБАВЛЕНИЕ ПОЛЕЙ ДАТЫ/ВРЕМЕНИ

- Добавить в модель Post различные поля даты/времени (каждый пост будет публиковаться в определенную дату и время, необходимо иметь поле для хранения даты и времени публикации);

- Также будут хранится дата и время создания объекта Post и его последнего изменения.

Отредактируйте файл models.py приложения blog, придав ему следующий вид. Новые строки выделены жирным шрифтом:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

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

ДОБАВЛЕНИЕ ПОЛЕЙ ДАТЫ/ВРЕМЕНИ

- В модель Post были добавлены следующие ниже поля:

• **publish**: поле с типом **DateTimeField**, которое транслируется в столбец **DATETIME** в базе данных SQL. Оно будет использоваться для хранения даты и времени публикации поста. По умолчанию значения поля задаются методом Django timezone.now. Метод timezone.now возвращает текущую дату/время в формате, зависящем от часового пояса;

• **created**: поле с типом DateTimeField. Оно будет использоваться для хранения даты и времени создания поста. При применении параметра auto_now_add дата будет сохраняться автоматически во время создания объекта;

• **updated**: поле с типом DateTimeField. Оно будет использоваться для хранения последней даты и времени обновления поста. При применении параметра auto_now дата будет обновляться автоматически во время сохранения объекта

ОПРЕДЕЛЕНИЕ ПРЕДУСТАНОВЛЕННОГО ПОРЯДКА СОРТИРОВКИ

- Посты блога обычно отображаются на странице в обратном хронологическом порядке (от самых новых к самым старым). В нашей модели мы определим заранее заданный порядок. Он будет применяться при извлечении объектов из базы данных, в случае если в запросе порядок не будет указан. Отредактируйте файл models.py приложения blog, придав ему следующий вид

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-publish']

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

ОПРЕДЕЛЕНИЕ ПРЕДУСТАНОВЛЕННОГО ПОРЯДКА СОРТИРОВКИ

Внутрь модели был добавлен Meta-класс. Этот класс определяет метаданные модели. Используется атрибут **ordering**, сообщающий Django, что он должен сортировать результаты по полю **publish**.

Указанный порядок будет применяться по умолчанию для запросов к базе данных, когда в запросе не указан конкретный порядок.

Убывающий порядок задается с помощью дефиса перед именем поля: -**publish**. По умолчанию посты будут возвращаться в обратном хронологическом порядке

ДОБАВЛЕНИЕ ИНДЕКСА БАЗЫ ДАННЫХ

Необходимо определить индекс базы данных по полю **publish**. Индекс повысит производительность запросов, фильтрующих или упорядочивающих результаты по указанному полю. Ожидаемо, что многие запросы извлекут преимущества из этого индекса, поскольку для упорядочивания результатов по умолчанию используется поле **publish**.

Отредактируйте файл **models.py** приложения **blog**, придав ему следующий вид. Новые строки выделены жирным шрифтом

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

ДОБАВЛЕНИЕ ИНДЕКСА БАЗЫ ДАННЫХ

В Meta-класс модели была добавлена опция **indexes**. Указанная опция позволяет определять в модели индексы базы данных, которые могут содержать одно или несколько полей в возрастающем либо убывающем порядке, или функциональные выражения и функции базы данных.

Был добавлен индекс по полю **publish**, а перед именем поля применен дефис, чтобы определить индекс в убывающем порядке.

Создание этого индекса будет вставляться в миграции базы данных, которую мы сгенерируем позже для моделей блога.

АКТИВАЦИЯ ПРИЛОЖЕНИЯ

Теперь необходимо активировать приложение **blog** в проекте, чтобы Django мог отслеживать приложение и имел возможность создавать таблицы базы данных для его моделей.

Отредактируйте файл **settings.py**, добавив **blog.apps.BlogConfig** в настроечный параметр **INSTALLED_APPS**. Это должно выглядеть, как показано ниже.

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

Класс **BlogConfig** – это конфигурация приложения. Теперь Django знает, что для этого проекта приложение является активным, и сможет загружать модели приложения

ДОБАВЛЕНИЕ ПОЛЯ СТАТУСА

Очень часто в функциональность ведения блогов входит хранение постов в виде черновика до тех пор, пока они не будут готовы к публикации. Мы добавим в модель поле статуса, которое позволит управлять статусом постов блога. В постах будут использоваться статусы **Draft** (Черновик) и **Published** (Опубликован).

Отредактируйте файл **models.py** приложения **blog**, придав ему следующий вид.

```
from django.db import models
from django.utils import timezone

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

ДОБАВЛЕНИЕ ПОЛЯ СТАТУСА

Мы определили перечисляемый класс **Status** путем подклассирования класса **models.TextChoices**. Доступными вариантами статуса поста являются **DRAFT** и **PUBLISHED**. Их соответствующими значениями выступают **DF** и **PB**, а их метками или читаемыми именами являются **Draft** и **Published**.

Django предоставляет перечисляемые типы, которые можно подклассировать, чтобы легко и просто определять варианты выбора. Они основаны на объекте enum стандартной библиотеки Python.

Для того чтобы получать имеющиеся варианты, можно обращаться к вариантам статуса (**Post.Status.choices**), для того чтобы получать удобочитаемые имена – к меткам статуса (**Post.Status.labels**), и для того чтобы получать фактические значения вариантов – к значениям статуса (**Post.Status.values**).

В модель также было добавлено новое поле **status**, являющееся экземпляром типа **CharField**. Оно содержит параметр **choices**, чтобы ограничивать значение поля вариантами из **Status.choices**. Кроме того, применяя параметр **default**, задано значение поля, которое будет использоваться по умолчанию. В этом поле статус **DRAFT** используется в качестве предустановленного варианта, если не указан иной.

ДОБАВЛЕНИЕ ПОЛЯ СТАТУСА

Как взаимодействовать с вариантами статуса. Выполните следующую ниже команду в командной оболочке, чтобы открыть оболочку Python:

```
python manage.py shell
```

Затем наберите такие строки:

```
>>> from blog.models import Post
>>> Post.Status.choices
```

Вы получите варианты перечисления в формате пар значение–метка, подобные показанным ниже:

```
[('DF', 'Draft'), ('PB', 'Published')]
```

Наберите следующую ниже строку:

```
>>> Post.Status.labels
```

Получите удобочитаемые имена членов перечисления **enum**, как показано ниже:

```
['Draft', 'Published']
```

Наберите такие строки:

```
>>> Post.Status.values
```

Вы получите значения элементов перечисления **enum**, как показано ниже. Эти значения можно сохранить в базе данных в поле **status**:

```
['DF', 'PB']
```

Наберите строку:

```
>>> Post.Status.names
```

Получите имена вариантов, как показано ниже:

```
['DRAFT', 'PUBLISHED']
```

К конкретному искомому перечисляемому элементу можно обращаться посредством **Post.Status.PUBLISHED**, а также обращаться к его свойствам .**name** и .**value**.

ДОБАВЛЕНИЕ ВЗАИМОСВЯЗИ МНОГИЕ-К-ОДНОМУ

Посты всегда пишутся автором. В данном разделе будет создана взаимосвязь между пользователями и постами, которая будет указывать на конкретных пользователей и написанные ими посты. Django идет в комплекте с фреймворком аутентификации, который ведет учетные записи пользователей. Встроенный в Django фреймворк аутентификации располагается в пакете **django.contrib.auth** и содержит модель **User** (Пользователь). Модель **User** будет применяться из указанного фреймворка аутентификации, чтобы создавать взаимосвязи между пользователями и постами. Отредактируйте файл **models.py** приложения **blog**, придав ему следующий вид:

ДОБАВЛЕНИЕ ВЗАИМОСВЯЗИ МНОГИЕ-К-ОДНОМУ

```python
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]
```

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

ДОБАВЛЕНИЕ ВЗАИМОСВЯЗИ МНОГИЕ-К-ОДНОМУ

Мы импортировали модель **User** из модуля **django.contrib.auth.models** и добавили в модель **Post** поле **author**. Это поле определяет взаимосвязь многие-к-одному, означающую, что каждый пост написан пользователем и пользователь может написать любое число постов. Для этого поля Django создаст внешний ключ в базе данных, используя первичный ключ соответствующей модели.

Параметр **on_delete** определяет поведение, которое следует применять при удалении объекта, на который есть ссылка. Это поведение не относится конкретно к Django; оно является стандартным для SQL. Использование ключевого слова CASCADE указывает на то, что при удалении пользователя, на которого есть ссылка, база данных также удалит все связанные с ним посты в блоге.

Мы используем **related_name**, чтобы указывать имя обратной связи, от **User** к **Post**. Такой подход позволит легко обращаться к связанным объектам из объекта **User**, используя обозначение **user.blog_posts**.

Теперь модель **Post** завершена, и сейчас можно синхронизировать ее с базой данных. Но перед этим нужно активировать приложение **blog** в проекте Django

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

Теперь, когда есть модель постов блога, необходимо создать соответствующую таблицу базы данных. Django идет в комплекте с системой миграции, которая отслеживает внесенные в модели изменения и позволяет их распространять по базе данных.

Команда **migrate** применяет миграции ко всем приложениям, перечисленным в **INSTALLED_APPS**. Она синхронизирует базу данных с текущими моделями и существующими миграциями. Прежде всего необходимо создать первоначальную миграцию модели **Post**. Выполните следующую ниже команду в командной оболочке из корневого каталога своего проекта:

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

```
python manage.py makemigrations blog
```

Вы должны получить результат, аналогичный приведенному ниже:

```
Migrations for 'blog':
    blog/migrations/0001_initial.py
        - Create model Post
        - Create index blog_post_publish_bb7600_idx on field(s)
          -publish of model post
```

Внутри каталога миграций приложения **blog Django** только что создал файл **0001_initial.py**. Эта миграция содержит инструкции **SQL** по созданию таблицы базы данных для модели **Post** и определения индекса базы данных для поля **publish**.

Можно взглянуть на содержимое файла, чтобы увидеть, как определяется миграция. Миграция задает зависимости от других миграций и операций, которые необходимо выполнить в базе данных, чтобы синхронизировать ее с изменениями модели.

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

Давайте взглянем на исходный код **SQL**, который Django исполнит в базе данных, чтобы создать таблицы вашей модели. Команда **sqlmigrate** принимает имена миграций и возвращает их **SQL** без его исполнения. Выполните следующую ниже команду из командной оболочки, чтобы проинспектировать результирующий исходный код **SQL** вашей первой миграции:

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

```
python manage.py sqlmigrate blog 0001
Результат должен выглядеть вот так:
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" (
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "title" varchar(250) NOT NULL,
  "slug" varchar(250) NOT NULL,
  "body" text NOT NULL,
  "publish" datetime NOT NULL,
  "created" datetime NOT NULL,
  "updated" datetime NOT NULL,
  "status" varchar(10) NOT NULL,
  "author_id" integer NOT NULL REFERENCES "auth_user" ("id") DEFERRABLE
INITIALLY DEFERRED);
--
-- Create blog_post_publish_bb7600_idx on field(s) -publish of model post
--
CREATE INDEX "blog_post_publish_bb7600_idx" ON "blog_post" ("publish" DESC);
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

Точный результат зависит от используемой вами базы данных. Приведенный выше результат сгенерирован для **SQLite**. Из полученного результата видно, что Django генерирует имена таблиц, комбинируя имя приложения с именем модели, обозначенной в нижнем регистре (**blog_post**). Кроме того, существует возможность указывать своей модели имя конкретно-прикладной базы данных. Это делается в Meta-классе модели при помощи атрибута **db_table**

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

Django создает автоинкрементный столбец **id**, используемый в каждой модели в качестве первичного ключа, указав **primary_key=True** в одном из полей модели, но это поведение можно тоже переопределять.

Столбец **id** состоит из автоматически увеличивающегося целого числа. Этот столбец соответствует полю **id**, которое добавляется в модель автоматически. Создаются следующие три индекса базы данных:

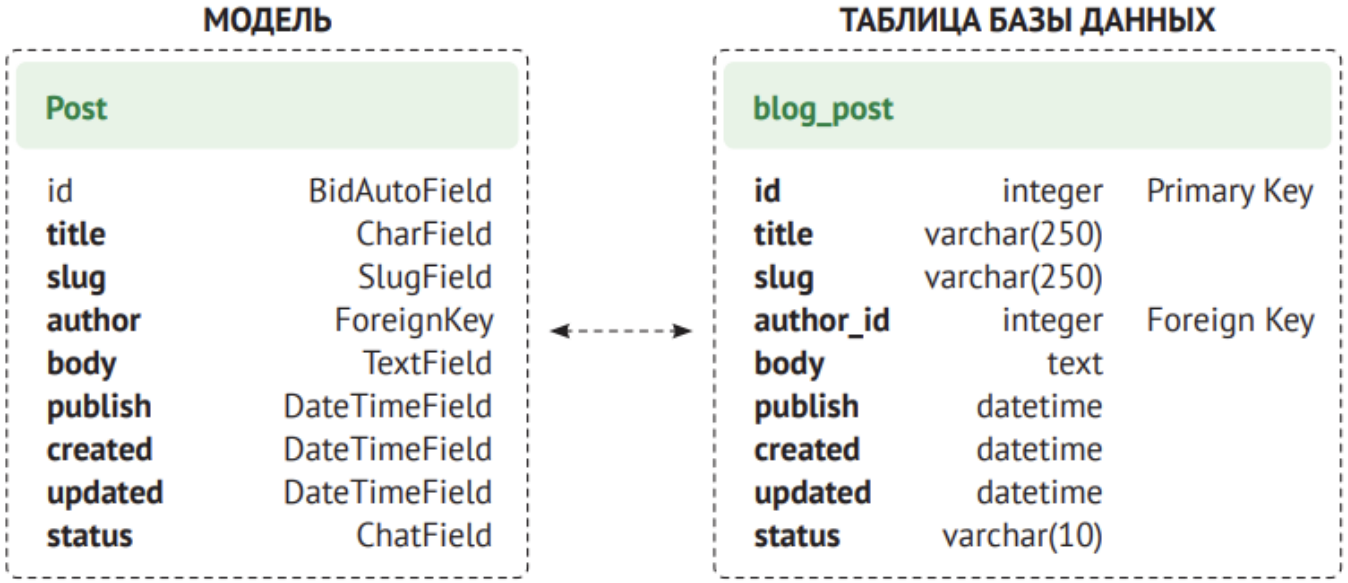
• индекс в убывающем порядке по столбцу **publish**. Мы определили этот индекс явным образом с помощью опции **indexes** Meta-класса модели;

• индекс по столбцу **slug**, поскольку поля типа **SlugField** по умолчанию подразумевают индекс;

• индекс по столбцу **author_id**, поскольку поля типа **ForeignKey** по умолчанию подразумевают индекс.

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

Давайте сравним модель Post с соответствующей ей таблицей blog_post базы данных



Полное соответствие модели Post и таблицы базы данных

Показано, как поля модели соответствуют столбцам таблицы базы данных

СОЗДАНИЕ И ПРИМЕНЕНИЕ МИГРАЦИЙ

Давайте синхронизируем базу данных с новой моделью. Примените следующую ниже команду в командной оболочке, чтобы воспользоваться существующими миграциями:

```
python manage.py migrate
```

Получите результат, который заканчивается следующей ниже строкой:

```
Applying blog.0001_initial... OK
```

Мы только что применили миграции приложений, перечисленных в **INSTALLED_APPS**, включая приложение **blog**.

После применения миграций база данных отражает текущее состояние моделей.

Если вы внесете в файл **models.py** любые правки, чтобы добавить, удалить либо изменить поля существующих моделей, либо добавите новые модели, то вам придется создать новые миграции, снова применив команду **makemigrations**.

Каждая миграция дает Django возможность отслеживать изменения модели. Затем нужно применить миграцию командой **migrate**, чтобы синхронизировать базу данных с моделями

СОЗДАНИЕ САЙТА АДМИНИСТРИРОВАНИЯ ДЛЯ МОДЕЛЕЙ

Теперь, когда модель **Post** синхронизирована с базой данных, можно создать простой сайт администрирования, чтобы управлять постами блога.

Django идет в комплекте со встроенным интерфейсом администрирования, который широко используется для редактирования контента.

Сайт Django формируется динамически путем чтения метаданных моделей и предоставления готового к работе интерфейса для редактирования контента.

Его можно использовать прямо «из коробки», сконфигурировав его так, чтобы ваши модели отображались в нем в том виде, в котором вы хотите.

Приложение **django.contrib.admin** уже вставлено в настроечный параметр **INSTALLED_APPS**, поэтому добавлять его нет необходимости.

СОЗДАНИЕ СУПЕРПОЛЬЗОВАТЕЛЯ

Сперва необходимо создать пользователя, который будет иметь право управлять сайтом администрирования. Выполните приведенную ниже команд:

```
python manage.py createsuperuser
```

Результат представлен ниже. Введите желаемое пользовательское имя (username), адрес электронной почты и пароль, как показано:

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
```

И вы увидите такое сообщение об успехе:

```
Superuser created successfully.
```

Мы только что создали пользователя-администратора с самым высоким уровнем разрешений