7

Отслеживание действий пользователя

В предыдущей главе вы разработали букмарклет JavaScript, чтобы делиться контентом с других веб-сайтов на своей платформе. Вы также внедрили в свой проект асинхронные действия с помощью JavaScript и создали бесконечную прокрутку.

В этой главе вы научитесь разрабатывать систему подписки и создавать поток активности пользователей. Вы также узнаете, как работают сигналы Django, и интегрируете в свой проект хранилище данных Redis с быстрым вводом-выводом с целью хранения определенного количества просмотров элементов.

В данной главе будут рассмотрены следующие темы:

- разработка системы подписки;
- создание взаимосвязей многие-ко-многим с промежуточной моделью;
- создание приложения потока активности;
- добавление обобщенных отношений в модели;
- оптимизация наборов запросов для связанных объектов;
- использование сигналов для денормализации количественных данных;
- использование меню отладочных инструментов Django Debug Toolbar для получения соответствующей отладочной информации;
- ведение подсчета просмотров изображений с помощью хранилища Redis;
- создание рейтинга самых просматриваемых изображений с помощью хранилища Redis.

Исходный код к этой главе находится по адресу https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter07.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Разработка системы подписки

Давайте разработаем для вашего проекта систему подписки. Под ней подразумевается, что ваши пользователи смогут подписываться друг на друга и отслеживать то, чем другие пользователи делятся на платформе. Взаимосвязь между пользователями классифицируется как связь многие-ко-многим: пользователь может следить за несколькими пользователями, а за ним, в свою очередь, могут следить несколько других пользователей.

Формирование взаимосвязей многие-ко-многим с промежуточной моделью

В предыдущих главах вы создавали взаимосвязи многие-ко-многим, добавляя `ManyToManyField` в одну из связанных моделей и позволяя Django создавать таблицу базы данных для этой взаимосвязи. Такой подход применим для большинства случаев, но иногда для подобной взаимосвязи возникает потребность в создании промежуточной модели. Создание промежуточной модели необходимо, когда требуется хранить дополнительную информацию о взаимосвязи, например дату создания взаимосвязи или поле, описывающее природу взаимосвязи.

Давайте создадим промежуточную модель с целью формирования взаимосвязей между пользователями. Есть две причины использования промежуточной модели:

- используется встроенная в Django модель `User` и нужно избежать ее изменения;
- нужно хранить время создания взаимосвязи.

Отредактируйте файл `models.py` приложения `account`, добавив следующий ниже исходный код:

```
class Contact(models.Model):
    user_from = models.ForeignKey('auth.User',
                                  related_name='rel_from_set',
                                  on_delete=models.CASCADE)
    user_to = models.ForeignKey('auth.User',
                                related_name='rel_to_set',
                                on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True)

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

```
    def __str__(self):
        return f'{self.user_from} follows {self.user_to}'
```

В приведенном выше исходном коде показана модель `Contact`, которая будет использоваться для взаимосвязей пользователей. Она содержит следующие поля:

- `user_from`: внешний ключ (`ForeignKey`) для пользователя, который создает взаимосвязь;
- `user_to`: внешний ключ (`ForeignKey`) для пользователя, на которого есть подписка;
- `created`: поле `DateTimeField` с параметром `auto_now_add=True` для хранения времени создания взаимосвязи.

Для полей `ForeignKey` индекс базы данных создается автоматически. В `Meta`-классе модели такой индекс определен в убывающем порядке по полю `created`. Также был добавлен атрибут `ordering`, чтобы сообщать Django, что по умолчанию он должен сортировать результаты по полю `created`. Используя дефис перед именем поля, указывается убывающий порядок. Например, `-created`.

Используя ORM, взаимосвязь между пользователем, `user1`, подписанным на другого пользователя, `user2`, можно создать, например, так:

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

Родственные менеджеры, `rel_from_set` и `rel_to_set`, будут возвращать набор запросов для модели `Contact`. Для того чтобы обратиться к конечной стороне взаимосвязи из модели `User`, желательно, чтобы `User` содержал поле `ManyToManyField`, как показано ниже:

```
following = models.ManyToManyField('self',
                                   through=Contact,
                                   related_name='followers',
                                   symmetrical=False)
```

Исходный код в приведенном выше примере сообщает Django, что для взаимосвязи нужно использовать конкретно-прикладную промежуточную модель, что достигается путем добавления параметра `through=Contact` в объект `ManyToManyField`. Это взаимосвязь многие-ко-многим из модели `User` в саму себя; в поле `ManyToManyField` делается ссылка `'self'`, чтобы создать взаимосвязь с той же моделью.

Если во взаимосвязи многие-ко-многим требуются дополнительные поля, то следует создать конкретно-прикладную модель с внешним ключом (`ForeignKey`) для каждой стороны взаимосвязи. В одну из связанных моделей следует добавить поле `ManyToManyField` и сообщить Django, что нужно использовать вашу промежуточную модель, включив ее в параметр `through`.

Если бы модель `User` была частью вашего приложения, то вы могли бы добавить упомянутое выше поле в модель. Однако изменить класс `User` напрямую невозможно, потому что он принадлежит приложению `django.contrib.auth`. Давайте воспользуемся немного другим подходом и будем добавлять это поле в модель `User` динамически.

Отредактируйте файл `models.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
from django.contrib.auth import get_user_model

# ...

# Добавить следующее поле в User динамически
user_model = get_user_model()
user_model.add_to_class('following',
                        models.ManyToManyField('self',
                            through=Contact,
                            related_name='followers',
                            symmetrical=False))
```

Здесь модель `User` извлекается встроенной в Django типовой функцией `get_user_model()`. Метод `add_to_class()` моделей Django применяется для того, чтобы динамически подправлять модель `User`.

Имейте в виду, что использование метода `add_to_class()` не является рекомендуемым способом добавления полей в модели. Тем не менее в данном случае его можно использовать, чтобы избежать создания конкретно-прикладной модели `User`, сохраняя все преимущества встроенной в Django модели `User`.

Также упрощается способ извлечения связанных объектов с использованием ORM-преобразователя посредством методов `user.followers.all()` и `user.following.all()`. Применяется промежуточная модель `Contact` и избегаются сложные запросы, которые потребовали бы дополнительных соединений в базе данных, как это было бы в случае, если бы вы определили взаимосвязи в своей конкретно-прикладной модели `Profile`. Таблица для этой взаимосвязи многие-ко-многим будет создана с использованием модели `Contact`. Таким образом, из динамически добавляемого поля `ManyToManyField` не будет следовать, что модель `User` будет вызывать какие-либо изменения в базе данных.

Имейте в виду, что в большинстве случаев предпочтительнее добавлять поля в созданную ранее модель `Profile`, а не вносить динамические правки в модель `User`. В идеале существующую модель `User` изменять не следует. Django позволяет применять конкретно-прикладные модели пользователя. Если вы хотите использовать такую модель пользователя, то ознакомьтесь с документацией по адресу https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#specifying-a-custom-user-model.

Обратите внимание, что взаимосвязь содержит параметр `symmetrical=False`. При определении поля `ManyToManyField` в модели, создавая взаимосвязь с самой моделью, Django навязывает взаимосвязи симметричность. В данном же случае устанавливается параметр `symmetrical=False`, чтобы определить не-

симметричную взаимосвязь (если я на вас подписываюсь, то это не означает, что вы автоматически подписываетесь на меня).

При использовании промежуточной модели для взаимосвязей многие-ко-многим некоторые методы родственного менеджера отключаются, такие как `add()`, `create()` или `remove()`. Вместо этого нужно создавать или удалять экземпляры промежуточной модели.

Выполните следующую ниже команду, чтобы создать первоначальные миграции для приложения `account`:

```
python manage.py makemigrations account
```

Вы получите результат, подобный следующему ниже:

```
Migrations for 'account':
  account/migrations/0002_auto_20220124_1106.py
    - Create model Contact
    - Create index account_con_created_8bdae6_idx on field(s) -created of model
contact
```

Теперь выполните следующую ниже команду, чтобы синхронизировать приложение с базой данных:

```
python manage.py migrate account
```

Вы должны увидеть результат, который содержит такую строку:

```
Applying account.0002_auto_20220124_1106... OK
```

Теперь модель `Contact` синхронизирована с базой данных, и вы можете создавать взаимосвязи между пользователями. Однако ваш сайт пока не предлагает возможности просмотра пользователей или просмотра профиля того либо иного пользователя. Давайте создадим представления списка и детальной информации для модели `User`.

Создание представлений списка и детальной информации для профилей пользователей

Откройте файл `views.py` приложения `account` и добавьте в него следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User
```

```
# ...

@login_required
def user_list(request):
    users = User.objects.filter(is_active=True)
    return render(request,
                  'account/user/list.html',
                  {'section': 'people',
                   'users': users})

@login_required
def user_detail(request, username):
    user = get_object_or_404(User,
                             username=username,
                             is_active=True)
    return render(request,
                  'account/user/detail.html',
                  {'section': 'people',
                   'user': user})
```

Это простые представления списка и детальной информации для объектов `User`. Представление `user_list` получает всех активных пользователей. Модель `User` содержит флаг `is_active`, который маркирует, считается учетная запись пользователя активной или нет. Запрос фильтруется по параметру `is_active=True`, чтобы возвращать только активных пользователей. Это представление возвращает все результаты, но его можно улучшить, добавив постраничную разбивку так же, как это делалось для представления `image_list`.

В представлении `user_detail` используется функция сокращенного доступа `get_object_or_404()`, чтобы извлекать активного пользователя с переданным пользовательским именем (`username`). Данное представление возвращает HTTP-ответ 404, если активный пользователь с переданным пользовательским именем не найден.

Отредактируйте файл `urls.py` приложения `account`, добавив шаблон URL-адреса для каждого представления, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
    path('users/', views.user_list, name='user_list'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

Шаблон URL-адреса `user_detail` будет использоваться для того, чтобы генерировать канонический URL-адрес для пользователей. Метод `get_absolute_`

url() уже в модели определен и будет возвращать канонический URL-адрес для каждого объекта. Еще одним способом указания URL-адреса для модели является добавление в проект настроечного параметра ABSOLUTE_URL_OVERRIDES.

Отредактируйте файл settings.py проекта, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.urls import reverse_lazy

# ...

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])
}
```

Django добавляет метод get_absolute_url() динамически в любые модели, которые появляются в настроечном параметре ABSOLUTE_URL_OVERRIDES. Этот метод возвращает соответствующий URL-адрес для заданной модели, которая указана в настроечном параметре. URL-адрес user_detail возвращается для заданного пользователя. Теперь метод get_absolute_url() можно использовать для экземпляра модели User, чтобы получать соответствующий URL-адрес.

Следующей ниже командой откройте оболочку Python:

```
python manage.py shell
```

Затем выполните приведенный далее исходный код, чтобы его протестировать:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.latest('id')
>>> str(user.get_absolute_url())
'/account/users/ellington/'
```

Возвращаемый URL-адрес соответствует ожидаемому формату /account/users/<username>/.

Теперь необходимо создать шаблоны только что разработанных представлений. Добавьте следующий ниже каталог и файлы в каталог templates/account/ приложения account:

```
/user/
    detail.html
    list.html
```

Отредактируйте шаблон account/user/list.html, добавив такой исходный код:

```
{% extends "base.html" %}
{% load thumbnail %}

{% block title %}People{% endblock %}

{% block content %}
  <h1>People</h1>
  <div id="people-list">
    {% for user in users %}
      <div class="user">
        <a href="{{ user.get_absolute_url }}">
          <img src="{% thumbnail user.profile.photo 180x180 %}">
        </a>
        <div class="info">
          <a href="{{ user.get_absolute_url }}" class="title">
            {{ user.get_full_name }}
          </a>
        </div>
      </div>
    {% endfor %}
  </div>
{% endblock %}
```

Приведенный выше шаблон позволяет перечислять всех активных пользователей на сайте. Заданные пользователи прокручиваются в цикле, и шаблонный тег `{% thumbnail %}` из `easy-thumbnails` используется для генерирования миниатюр изображений, относящихся к данному профилю.

Обратите внимание, что у пользователей должно быть изображение профиля. Для того чтобы использовать типовое изображение для тех пользователей, у которых нет изображения профиля, можно добавить инструкцию `if/else`, чтобы проверять наличие у пользователя фотографии профиля, например `{% if user.profile.photo %} {# миниатюра фотографии #} {% else %} {# типовое изображение #} {% endif %}`.

Откройте шаблон `base.html` проекта и вставьте URL-адрес `user_list` в атрибут `href` следующего ниже пункта меню. Новый исходный код выделен жирным шрифтом:

```
<ul class="menu">
  ...
  <li {% if section == "people" %}class="selected"{% endif %}>
    <a href="{% url "user_list" %}">People</a>
  </li>
</ul>
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/users/` в своем браузере. Вы должны увидеть список пользователей, подобный следующему ниже:

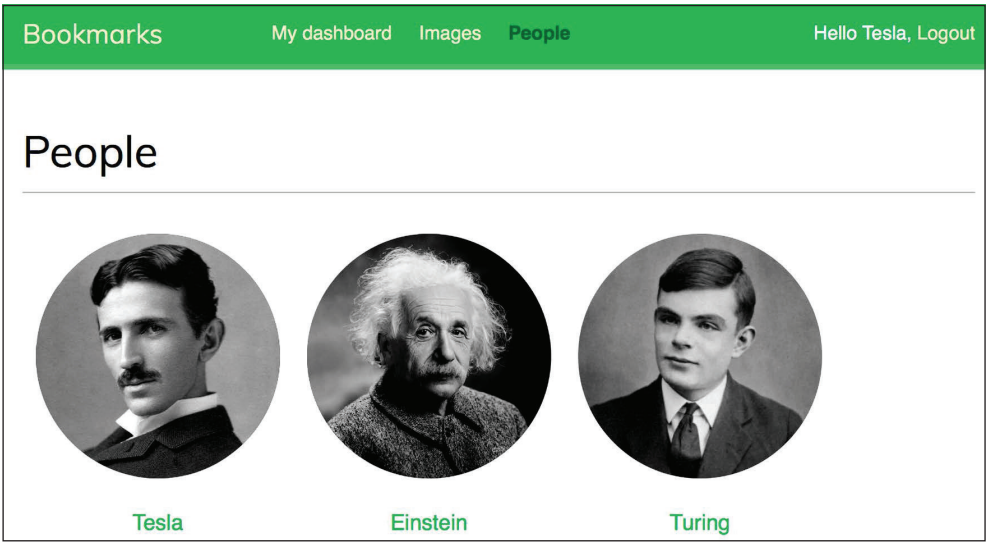


Рис. 7.1. Страница списка пользователей с миниатюрами изображений профилей

Напомним, что если у вас возникли трудности с генерированием миниатюр, то в свой файл `settings.py` можно добавить настроечный параметр `THUMBNAIL_DEBUG = True`, чтобы получать отладочную информацию в оболочке.

Отредактируйте шаблон `account/user/detail.html` приложения `account`, добавив следующий ниже исходный код:

```
{% extends "base.html" %}
{% load thumbnail %}

{% block title %}{{ user.get_full_name }}{% endblock %}

{% block content %}
  <h1>{{ user.get_full_name }}</h1>
  <div class="profile-info">
    <img src="{% thumbnail user.profile.photo 180x180 %}" class="user-detail">
  </div>
  {% with total_followers=user.followers.count %}
    <span class="count">
      <span class="total">{{ total_followers }}</span>
      follower{{ total_followers|pluralize }}
    </span>
    <a href="#" data-id="{{ user.id }}" data-action="{% if request.user in user.
followers.all %}un{% endif %}follow" class="follow button">
```

```
        {% if request.user not in user.followers.all %}
          Follow
        {% else %}
          Unfollow
        {% endif %}
      </a>
      <div id="image-list" class="image-container">
        {% include "images/image/list_images.html" with images=user.images_created.all %}
      </div>
    {% endwith %}
  {% endblock %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк; Django не поддерживает многострочные теги.

В шаблоне детальной информации отображается профиль пользователя, а шаблонный тег `{% thumbnail %}` используется для вывода изображения профиля на странице. При этом отображается общее число подписчиков и ссылка, чтобы подписаться либо отписаться от пользователя. Эта ссылка будет использоваться для подписки/отписки от конкретного пользователя. Атрибуты `data-id` и `data-action` HTML-элемента `<a>` содержат ИД пользователя и первоначальное действие, которое необходимо выполнять при нажатии на ссылочный элемент, – `follow` (подписаться) либо `unfollow` (отписаться). Первоначальное действие (*follow* либо *unfollow*) зависит от того, является запрашивающий страницу пользователь уже подписчиком пользователя или нет. Изображения, отмеченные пользователем закладкой, отображаются путем вставки шаблона `images/image/list_images.html`.

Снова откройте браузер и кликните на пользователе, который отметил несколько изображений закладкой. Страница пользователя будет выглядеть следующим образом:

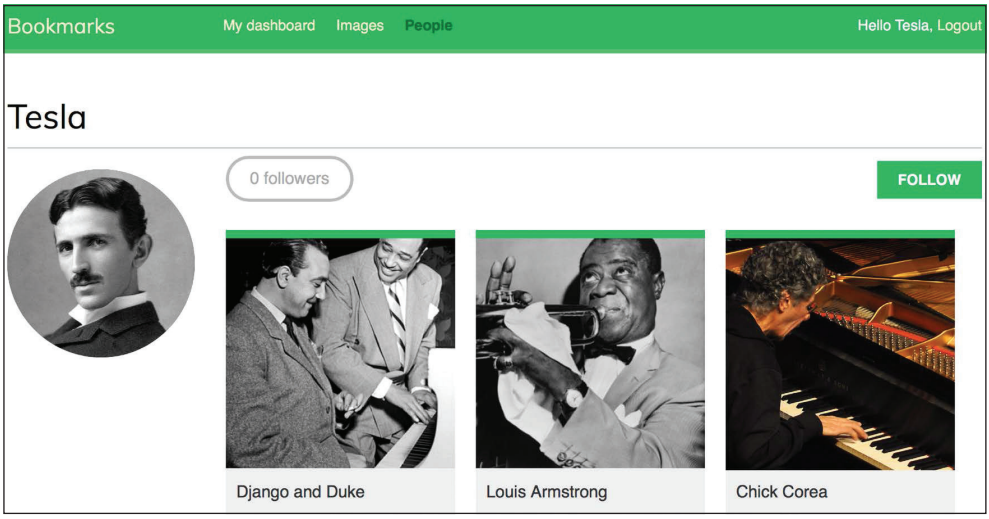


Рис. 7.2. Страница детальной информации о пользователе

Изображение Чика Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: https://creativecommons.org/licenses/by/2.0/).

Добавление действий пользователя по подписке/отписке с помощью JavaScript

Давайте добавим функциональность подписки на пользователей и отписки от пользователей. Мы создадим новое представление подписки/отписки и реализуем асинхронный HTTP-запрос с помощью JavaScript для действия по подписке/отписке.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from .models import Contact

# ...

@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                    user_from=request.user,
                    user_to=user)
            else:
                Contact.objects.filter(user_from=request.user,
                                       user_to=user).delete()
            return JsonResponse({'status':'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status':'error'})
    return JsonResponse({'status':'error'})
```

Представление `user_follow` очень похоже на представление `image_like`, которое вы создали в главе 6 «Распространение контента на веб-сайте». Поскольку вы применяете конкретно-прикладную промежуточную модель для пользовательской взаимосвязи многие-ко-многим, стандартные методы

`add()` и `remove()` автоматического менеджера полей `ManyToManyField` недоступны. Вместо этого для создания или удаления пользовательских взаимосвязей применяется промежуточная модель `Contact`.

Отредактируйте файл `urls.py` приложения `account`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
    path('users/', views.user_list, name='user_list'),
    path('users/follow/', views.user_follow, name='user_follow'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

Проверьте, чтобы приведенный выше шаблон был помещен перед шаблоном URL-адреса `user_detail`. В противном случае любые запросы к `/users/follow/` будут совпадать с регулярным выражением шаблона `user_detail`, и вместо этого будет выполняться указанное представление. Напомним, что в каждом HTTP-запросе веб-фреймворк Django проверяет запрошенный URL-адрес на соответствие каждому шаблону в порядке появления и останавливается при первом совпадении.

Отредактируйте шаблон `user/detail.html` приложения `account`, добавив следующий ниже исходный код:

```
{% block domready %}
  var const = '{% url "user_follow" %}';
  var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrftoken},
    mode: 'same-origin'
  }

  document.querySelector('a.follow')
          .addEventListener('click', function(e){
    e.preventDefault();
    var followButton = this;

    // добавить тело запроса
    var formData = new FormData();
    formData.append('id', followButton.dataset.id);
    formData.append('action', followButton.dataset.action);
    options['body'] = formData;

    // отправить HTTP-запрос
    fetch(url, options)
```

```
    .then(response => response.json())
    .then(data => {
      if (data['status'] === 'ok')
      {
        var previousAction = followButton.dataset.action;

        // переключить текст кнопки и data-action
        var action = previousAction === 'follow' ? 'unfollow' : 'follow';
        followButton.dataset.action = action;
        followButton.innerHTML = action;

        // обновить количество подписчиков
        var followerCount = document.querySelector('span.count .total');
        var totalFollowers = parseInt(followerCount.innerHTML);
        followerCount.innerHTML = previousAction === 'follow' ? totalFollowers + 1 :
totalFollowers - 1;
      }
    })
  });
{% endblock %}
```

Приведенный выше шаблонный блок содержит исходный код JavaScript, который выполняет асинхронный HTTP-запрос на подписку или отписку от того или иного пользователя, а также переключает ссылку подписки/отписки. Интерфейс Fetch API используется для выполнения запроса AJAX и установки как атрибута `data-action`, так и текста HTML-элемента `<a>` на основе его предыдущего значения. После завершения действия также обновляется отображаемое на странице общее число подписчиков.

Откройте страницу детальной информации о существующем пользователе и кликните по ссылке **FOLLOW** (Подписаться), чтобы протестировать только что созданную функциональность. Вы увидите, что количество подписчиков увеличилось:

Рис. 7.3. Количество подписчиков и кнопка подписаться/отписаться

Теперь система подписки завершена, и пользователи могут подписываться друг на друга. Далее мы создадим поток активности, создав соответствующий контент для каждого пользователя, основанный на людях, на которых они подписаны.

Разработка типового приложения для потока активности

Многие социальные веб-сайты показывают своим пользователям поток активности, предоставляя им возможность отслеживать то, что другие пользователи делают на платформе. Поток активности – это список последних действий, выполненных пользователем или группой пользователей. Например, новостная лента Facebook является потоком активности. Примеры действий могут быть такими: *пользователь X пометил изображение Y закладкой* или *теперь пользователь X подписан на пользователя Y*.

Сейчас вы разработаете приложение для потока активности, которое будет давать пользователю возможность видеть недавние взаимодействия пользователей, на которых он подписан. Для этого понадобится модель, которая будет хранить действия пользователей на сайте, и простой способ добавления действий в новостную ленту.

Следующей ниже командой создайте новое приложение с именем `actions` внутри проекта:

```
python manage.py startapp actions
```

Добавьте новое приложение в настроечный параметр `INSTALLED_APPS` в файле `settings.py` проекта, чтобы активировать приложение в проекте. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [
    # ...
    'actions.apps.ActionsConfig',
]
```

Отредактируйте файл `models.py` приложения `actions`, добавив следующий ниже исходный код:

```
from django.db import models

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                             related_name='actions',
                             on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True)
```

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

В приведенном выше исходном коде показана модель `Action`, которая будет использоваться для хранения действий пользователя. Поля этой модели таковы:

- `user`: пользователь, выполнивший действие; это внешний ключ (`ForeignKey`) для встроенной в Django модели `User`;
- `verb`: глагол, описывающий действие, которое выполнил пользователь;
- `created`: дата и время создания этого действия. Параметр `auto_now_add=True` используется для того, чтобы автоматически устанавливать текущую дату и время при первом сохранении объекта в базе данных.

В `Meta`-классе модели был определен индекс базы данных в убывающем порядке по полю `created`. Кроме того, был добавлен атрибут `ordering`, чтобы сообщать Django, что по умолчанию результаты следует сортировать по полю `created` в убывающем порядке.

В этой базовой модели можно хранить только такие действия, как: *пользователь X что-то сделал*. Необходимо иметь дополнительное поле `ForeignKey`, чтобы хранить действия, связанные с целевым объектом `target`, например *пользователь X пометил изображение Y закладкой* или *теперь пользователь X подписан на пользователя Y*. Как вы уже знаете, обычный внешний ключ (`ForeignKey`) может указывать только на одну модель. Вместо этого понадобится способ, которым целевой объект действия будет экземпляром существующей модели. В этом поможет встроенный в Django фреймворк типов контента `contenttypes`.

Применение фреймворка contenttypes

Django содержит фреймворк `contenttypes`, расположенный в приложении `django.contrib.contenttypes`. Указанное приложение может отслеживать все установленные в проекте модели и предоставляет типовой интерфейс взаимодействия с этими моделями.

Приложение `django.contrib.contenttypes` включается в настроечный параметр `INSTALLED_APPS` по умолчанию при создании нового проекта с помощью команды `startproject`. Он используется другими пакетами `contrib`, такими как фреймворк аутентификации и приложение администрирования.

Приложение `contenttypes` содержит модель `ContentType`. Экземпляры этой модели представляют фактические модели вашего приложения, а новые экземпляры `ContentType` создаются автоматически при установке в проект новых моделей. Модель `ContentType` имеет следующие поля:

- `app_label`: указывает имя приложения, которому модель принадлежит. Оно берется автоматически из атрибута `app_label` модельных `Meta`-опций. Например, ваша модель `Image` принадлежит приложению `images`;
- `model`: имя модельного класса;
- `name`: указывает удобочитаемое имя модели. Оно берется автоматически из атрибута `verbose_name` модельных `Meta`-опций.

Давайте посмотрим, как взаимодействовать с объектами `ContentType`. Следующей ниже командой откройте оболочку:

```
python manage.py shell
```

Объект `ContentType`, соответствующий конкретной модели, можно получать путем выполнения запроса с атрибутами `app_label` и `model`, как показано ниже:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> image_type = ContentType.objects.get(app_label='images', model='image')
>>> image_type
<ContentType: images | image>
```

Из объекта `ContentType` можно получать модельный класс, вызвав его метод `model_class()`:

```
>>> image_type.model_class()
<class 'images.models.Image'>
```

Объект `ContentType` также принято получать для определенного модельного класса, как показано ниже:

```
>>> from images.models import Image
>>> ContentType.objects.get_for_model(Image)
<ContentType: images | image>
```

Это всего лишь несколько примеров использования объектов приложения `contenttypes`, и Django предлагает много других способов работы с ними. Официальная документация по фреймворку `contenttypes` находится на странице https://docs.djangoproject.com/en/4.1/ref/contrib/contenttypes/.

Добавление обобщенных отношений в модели

В обобщенных отношениях объекты `ContentType` играют роль указателей на модель, используемую для взаимосвязи. При этом для того чтобы установить обобщенное отношение, в модели понадобятся три поля:

- поле `ForeignKey` для `ContentType`: оно будет сообщать о модели, используемой для взаимосвязи;

- поле для хранения первичного ключа связанного объекта: обычно это `PositiveIntegerField`, чтобы сочетаться со встроенными в Django автоматическими полями первичных ключей;
- поле для определения обобщенного отношения и управления им с использованием двух предыдущих полей: для этой цели фреймворк `contenttypes` предлагает поле `GenericForeignKey`.

Отредактируйте файл `models.py` приложения `actions`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                             related_name='actions',
                             on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True)
    target_ct = models.ForeignKey(ContentType,
                                  blank=True,
                                  null=True,
                                  related_name='target_obj',
                                  on_delete=models.CASCADE)
    target_id = models.PositiveIntegerField(null=True,
                                            blank=True)
    target = GenericForeignKey('target_ct', 'target_id')

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

В модель `Action` были добавлены следующие поля:

- `target_ct`: поле `ForeignKey`, указывающее на модель `ContentType`;
- `target_id`: `PositiveIntegerField` для хранения первичного ключа связанного объекта;
- `target`: поле `GenericForeignKey` для связанного объекта на основе комбинации двух предыдущих полей.

Кроме того, был добавлен многопольный индекс, включающий поля `target_ct` и `target_id`.

Django не создает поля `GenericForeignKey` в базе данных. Единственными полями, которые соотносятся с полями базы данных, являются `target_ct`

и `target_id`. Оба поля имеют атрибуты `empty=True` и `null=True`, поэтому при сохранении объектов `Action` целевой объект `target` не требуется.

Применение обобщенных отношений вместо внешних ключей будет придавать приложениям бóльшую гибкость.

Выполните следующую ниже команду, чтобы создать первоначальные миграции приложения:

```
python manage.py makemigrations actions
```

Вы должны увидеть следующий ниже результат:

```
Migrations for 'actions':
  actions/migrations/0001_initial.py
    - Create model Action
    - Create index actions_act_created_64f10d_idx on field(s) -created of model
action
    - Create index actions_act_target__f20513_idx on field(s) target_ct,
target_id of model action
```

Затем выполните приведенную ниже команду, чтобы синхронизировать приложение с базой данных:

```
python manage.py migrate
```

Результат команды должен указывать на то, что новые миграции были применены, как показано далее:

```
Applying actions.0001_initial... OK
```

Давайте добавим модель `Action` на сайт администрирования. Отредактируйте файл `admin.py` приложения `actions`, добавив следующий ниже исходный код:

```
from django.contrib import admin
from .models import Action

@admin.register(Action)
class ActionAdmin(admin.ModelAdmin)
    list_display = ['user', 'verb', 'target', 'created']
    list_filter = ['created']
    search_fields = ['verb']
```

Вы только что зарегистрировали модель `Action` на сайте администрирования.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/actions/action/add/` в своем браузере. Вы должны увидеть страницу создания нового объекта `Action`, как показано далее:

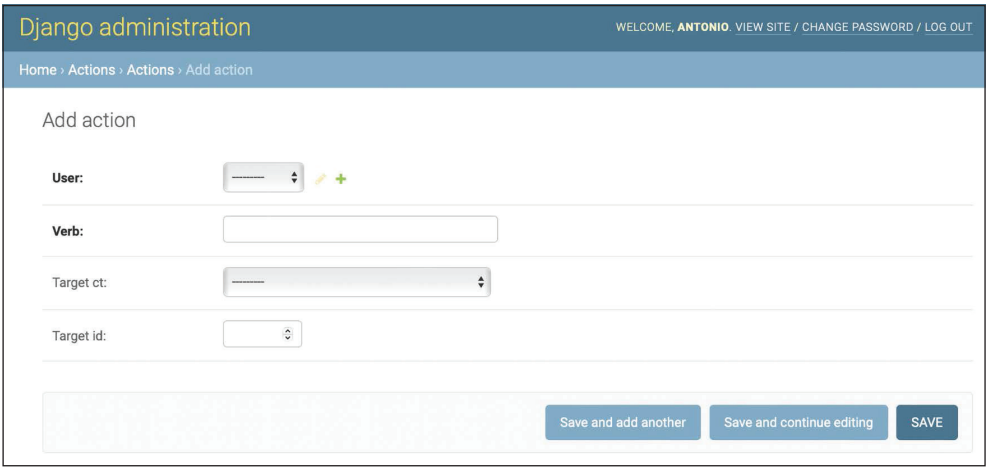


Рис. 7.4. Страница добавления действия на сайте администрирования

Как вы заметили, на приведенном выше снимке экрана показаны только поля `target_ct` и `target_id`, которые соотносятся с фактическими полями базы данных. Поле `GenericForeignKey` в форме не отображается. Поле `target_ct` позволяет выбирать любую зарегистрированную модель своего проекта Django. Выбор типов контента можно ужимать путем выбора из ограниченного набора моделей, используя атрибут `limit_choices_to` в поле `target_ct`; атрибут `limit_choices_to` позволяет ограничивать содержимое полей `ForeignKey` определенным набором значений.

Внутри каталога приложения `action` создайте новый файл и назовите его `utils.py`. Сейчас необходимо определить функцию быстрого доступа, которая позволит создавать новые объекты `Action` простым способом. Отредактируйте новый файл `utils.py`, добавив следующий ниже исходный код:

```
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    action = Action(user=user, verb=verb, target=target)
    action.save()
```

Функция `create_action()` позволяет создавать действия, которые опционально включают целевой объект `target`. Эту функцию можно применять в любом месте своего исходного кода в качестве функции сокращенного доступа, чтобы добавлять новые действия в поток активности.

Игнорирование повторных действий в потоке активности

Иногда могут возникать ситуации, когда пользователи будут кликать по кнопке **Like** или **Unlike** несколько раз либо неоднократно выполнять одно и то же действие за короткий промежуток времени. Они легко будут приводить к сохранению и отображению повторяющихся действий. Во избежание этого давайте усовершенствуем функцию `create_action()`, чтобы игнорировать очевидные повторяющиеся действия.

Отредактируйте файл `utils.py` приложения `actions`, как показано ниже:

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    # проверить, не было ли каких-либо аналогичных
    # действий, совершенных за последнюю минуту
    now = timezone.now()
    last_minute = now - datetime.timedelta(seconds=60)
    similar_actions = Action.objects.filter(user_id=user.id,
                                            verb= verb,
                                            created__gte=last_minute)
    if target:
        target_ct = ContentType.objects.get_for_model(target)
        similar_actions = similar_actions.filter(
                                                target_ct=target_ct,
                                                target_id=target.id)
    if not similar_actions:
        # никаких существующих действий не найдено
        action = Action(user=user, verb=verb, target=target)
        action.save()
        return True
    return False
```

Функция `create_action()` была изменена, чтобы избегать сохранения повторяющихся действий и возвращать булево значение, сообщающее о том, не было ли действие сохранено. Вот как игнорируются повторы:

1) сначала берется текущее время. Это делается с помощью встроенного в Django метода `timezone.now()`. Указанный метод делает то же самое,

что и `datetime.datetime.now()`, но возвращает объект с учетом часового пояса. Django предоставляет настроечный параметр `USE_TZ`, которым активируется либо деактивируется поддержка часового пояса. Стандартный файл `settings.py`, созданный с помощью команды `startproject`, содержит `USE_TZ=True`;

2) переменная `last_minute` используется для хранения даты/времени давностью одна минута назад и для получения любых идентичных действий, выполненных пользователем с тех пор;
3) если за последнюю минуту не было идентичного действия, то создается объект `Action`. При этом возвращается `True`, если объект `Action` был создан, либо `False` в противном случае.

Добавление действий пользователя в поток активности

Теперь самое время добавить несколько действий в представления, чтобы сформировать поток активности для ваших пользователей. Действие будет сохраняться для каждого следующего ниже взаимодействия:

- пользователь отмечает изображение закладкой;
- пользователю нравится изображение;
- пользователь создает учетную запись;
- пользователь только что подписался на другого пользователя.

Отредактируйте файл `views.py` приложения `images`, добавив следующую ниже инструкцию импорта:

```
from actions.utils import create_action
```

В представлении `image_create` сразу после сохранения изображения добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
@login_required
def image_create(request):
    if request.method == 'POST':
        # форма отправлена
        form = ImageCreateForm(data=request.POST)
        if form.is_valid():
            # данные в форме валидны
            cd = form.cleaned_data
            new_image = form.save(commit=False)
            # назначить текущего пользователя элементу
            new_image.user = request.user
            new_image.save()
            create_action(request.user, 'bookmarked image', new_image)
            messages.success(request, 'Image added successfully')
```

```
            # перенаправить к представлению детальной
            # информации о только что созданном элементе
            return redirect(new_image.get_absolute_url())
    else:
        # скомпоновать форму с данными,
        # предоставленными букмарклетом методом GET
        form = ImageCreateForm(data=request.GET)
    return render(request,
                  'images/image/create.html',
                  {'section': 'images',
                   'form': form})
```

В представлении `image_like` сразу после добавления пользователя во взаимосвязь `users_like` добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
@login_required
@require_POST
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
            if action == 'like':
                image.users_like.add(request.user)
                create_action(request.user, 'likes', image)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status':'ok'})
        except Image.DoesNotExist:
            pass
    return JsonResponse({'status':'error'})
```

Теперь отредактируйте файл `views.py` приложения `account`, добавив следующую ниже инструкцию импорта:

```
from actions.utils import create_action
```

В представлении `register` сразу после создания объекта `Profile` добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
```

```
            # Создать новый объект пользователя,
            # но пока не сохранять его
            new_user = user_form.save(commit=False)
            # Установить выбранный пароль
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Сохранить объект User
            new_user.save()
            # Создать профиль пользователя
            Profile.objects.create(user=new_user)
            create_action(new_user, 'has created an account')
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

В представление `user_follow` добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                    user_from=request.user,
                    user_to=user)
                create_action(request.user, 'is following', user)
            else:
                Contact.objects.filter(user_from=request.user,
                                       user_to=user).delete()
            return JsonResponse({'status':'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status':'error'})
    return JsonResponse({'status':'error'})
```

Как видно из приведенного выше исходного кода, благодаря модели `Action` и вспомогательной функции сохранять новые действия в потоке активности довольно легко.

Отображение потока активности

Наконец, требуется способ отображения потока активности по каждому пользователю. В связи с этим необходимо добавить поток активности на информационную панель пользователя. Отредактируйте файл `views.py` приложения `account`, импортировав модель `Action` и видоизменив представление информационной панели, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from actions.models import Action

# ...

@login_required
def dashboard(request):
    # По умолчанию показать все действия
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # Если пользователь подписан на других,
        # то извлечь только их действия
        actions = actions.filter(user_id__in=following_ids)
    actions = actions[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

В приведенном выше представлении из базы данных извлекаются все действия, за исключением тех, которые выполняются текущим пользователем. По умолчанию извлекаются последние действия, выполненные всеми пользователями на платформе. Если пользователь подписан на других пользователей, то запрос ограничивается, чтобы получать только те действия, которые выполняются пользователями, на которых он подписан. Наконец, результат ограничивается первыми 10 возвращаемыми действиями. Метод `order_by()` в наборе запросов QuerySet не используется, потому что вы опираетесь на заранее заданный порядок сортировки, указанный в `Meta`-опциях модели `Action`. Недавние действия будут первыми, поскольку в модели `Action` было задано `ordering = ['-created']`.

Оптимизация наборов запросов, предусматривающих связанные объекты

Всякий раз, когда извлекается объект `Action`, обычно обращаются к связанному с ним объекту `User` и к связанному с пользователем объекту `Profile`.

Встроенный в Django ORM-преобразователь предлагает простой способ одновременного извлечения связанных объектов, что позволяет избегать дополнительных запросов к базе данных.

Применение метода select_related()

Django предлагает QuerySet-метод под названием `select_related()`, который позволяет извлекать связанные объекты для взаимосвязей один-ко-многим. Это транслируется в один более сложный набор запросов, но зато позволяет избегать дополнительных запросов при доступе к связанным объектам. Метод `select_related` предназначен для полей `ForeignKey` и `OneToOne`. Он работает, выполняя SQL-инструкцию `JOIN` и включая поля связанного объекта в инструкцию `SELECT`.

Для того чтобы воспользоваться преимуществами метода `select_related()`, отредактируйте следующую ниже строку приведенного выше исходного кода в файле `views.py` приложения `account`, добавив `select_related`, включая поля, которые будут использоваться, например, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
@login_required
def dashboard(request):
    # По умолчанию показать все действия
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # Если пользователь подписан на других,
        # то извлечь только их действия
        actions = actions.filter(user_id__in=following_ids)
    actions = actions.select_related('user', 'user__profile')[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                  'actions': actions})
```

Аргумент `user__profile` используется для того, чтобы выполнять операцию соединения на таблице `Profile` в одном SQL-запросе. Если вызвать `select_related()` без передачи каких-либо аргументов, то он будет извлекать объекты из всех взаимосвязей с внешними ключами `ForeignKey`. Следует всегда ограничивать метод `select_related()` взаимосвязями, которые будут доступны позже.

Осторожное применение метода `select_related()` может значительно сократить время исполнения запросов.

Применение метода prefetch_related()

Метод `select_related()` поможет повышать производительность при извлечении связанных объектов во взаимосвязях один-ко-многим. Однако он не работает для взаимосвязей многие-ко-многим или многие-к-одному (поля `ManyToMany` или обратного внешнего ключа `ForeignKey`). Django предлагает другой QuerySet-метод под названием `prefetch_related`, который в дополнение к взаимосвязям, поддерживаемым методом `select_related()`, успешно работает для взаимосвязей многие-ко-многим и многие-к-одному. Метод `prefetch_related()` выполняет отдельный поиск по каждой взаимосвязи и соединяет результаты с помощью Python. Этот метод также поддерживает упреждающую выборку полей `GenericRelation` и `GenericForeignKey`.

Отредактируйте файл `views.py` приложения `account`, чтобы завершить свой запрос, добавив в него функцию `prefetch_related()` для целевого поля `GenericForeignKey`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # Если пользователь подписан на других,
        # то извлечь только их действия
        actions = actions.filter(user_id__in=following_ids)
    actions = actions.select_related('user', 'user__profile')[:10]\
                     .prefetch_related('target')[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                  'actions': actions})
```

Теперь этот запрос оптимизирован под получение действий пользователя, включая связанные объекты.

Создание шаблонов действий

Теперь давайте создадим шаблон для отображения того или иного объекта `Action`. Создайте новый каталог внутри каталога приложения `actions` и назовите его `templates`. Добавьте в него следующую ниже файловую структуру:

```
actions/
    action/
        detail.html
```

Отредактируйте файл шаблона `action/action/detail.html`, добавив следующие ниже строки:

```
{% load thumbnail %}

{% with user=action.user profile=action.user.profile %}
<div class="action">
  <div class="images">
    {% if profile.photo %}
      {% thumbnail user.profile.photo "80x80" crop="100%" as im %}
      <a href="{{ user.get_absolute_url }}">
        <img src="{{ im.url }}" alt="{{ user.get_full_name }}"
          class="item-img">
      </a>
    {% endif %}
    {% if action.target %}
      {% with target=action.target %}
        {% if target.image %}
          {% thumbnail target.image "80x80" crop="100%" as im %}
          <a href="{{ target.get_absolute_url }}">
            <img src="{{ im.url }}" class="item-img">
          </a>
        {% endif %}
      {% endwith %}
    {% endif %}
  </div>
  <div class="info">
    <p>
      <span class="date">{{ action.created|timesince }} ago</span>
      <br />
      <a href="{{ user.get_absolute_url }}">
        {{ user.first_name }}
      </a>
      {{ action.verb }}
      {% if action.target %}
        {% with target=action.target %}
          <a href="{{ target.get_absolute_url }}">{{ target }}</a>
        {% endwith %}
      {% endif %}
    </p>
  </div>
</div>
{% endwith %}
```

Это шаблон, который будет использоваться для отображения объекта `Action`. Вначале используется шаблонный тег `{% with %}`, чтобы извлекать выполняющего действие пользователя и связанный с ним объект `Profile`. Затем, если объект `Action` имеет связанный объект `target`, отображается изображение объекта `target`. Наконец, отображается ссылка на выполняющего действие пользователя, глагол и объект `target`, если он есть.

Отредактируйте шаблон `account/dashboard.html` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в нижнюю часть блока `content`:

```
{% extends "base.html" %}

{% block title %}Dashboard{% endblock %}

{% block content %}

  ...

  <h2>What's happening</h2>
  <div id="action-list">
    {% for action in actions %}
      {% include "actions/action/detail.html" %}
    {% endfor %}
  </div>
{% endblock %}
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/` в своем браузере. Войдите в систему как существующий пользователь и выполните несколько действий, чтобы они были сохранены в базе данных. Затем войдите в систему, используя другого пользователя, подпишитесь на предыдущего пользователя и посмотрите на сгенерированный поток действий на странице информационной панели.

Это должно выглядеть следующим образом:

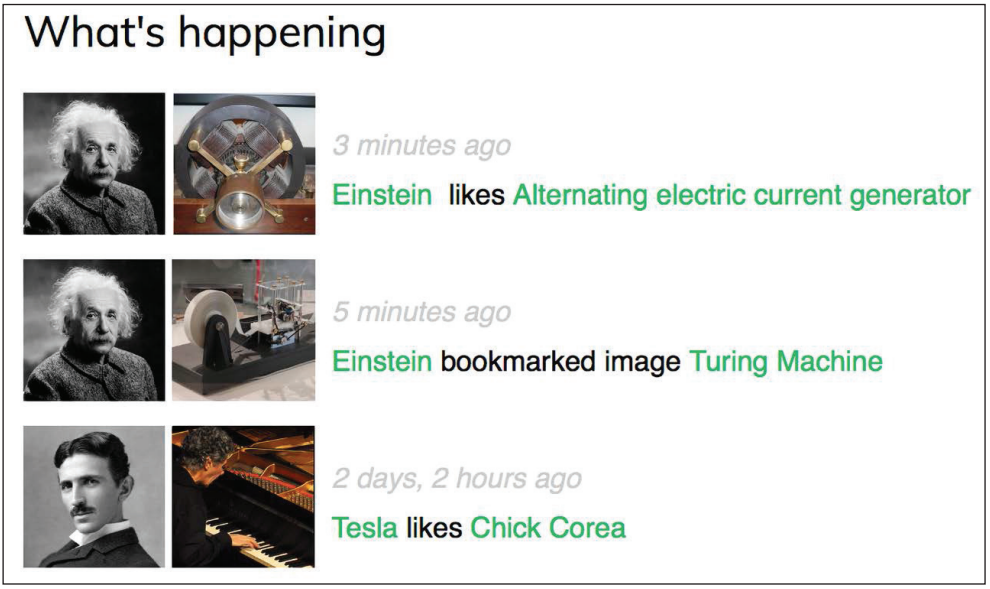


Рис. 7.5. Поток активности текущего пользователя

Авторство изображений на рис. 7.5:

- Асинхронный электродвигатель Теслы, автор: Ctac (лицензия: Creative Commons Attribution Share-Alike 3.0). Непортированная форма: https://creativecommons.org/licenses/by-sa/3.0/);
- Модель машины Тьюринга Davey 2012, автор: Рокки Акоста (лицензия: Creative Commons Attribution 3.0 Непортированная форма: https://creativecommons.org/licenses/by/3.0/);
- Чик Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: https://creativecommons.org/licenses/by/2.0/).

Вы только что создали полный поток активности своих пользователей и можете легко добавлять в него новые действия пользователей. В поток активности также можно добавлять функциональность бесконечной прокрутки, реализовав того же AJAX-ориентированного постраничного разбивщика, которого вы использовали для представления `image_list`. Далее вы научитесь использовать сигналы Django для денормализации количеств действий.

Использование сигналов для денормализации количественных данных

В некоторых случаях вы, возможно, захотите проводить денормализацию своих данных. Денормализация делает данные избыточными, но при этом она оптимизирует производительность операций чтения. Например, можно скопировать связанные данные в объект, чтобы избежать дорогостоящих запросов к базе данных, предусматривающих операцию чтения при извлечении связанных данных. В работе с денормализацией нужно быть осмотрительным и начинать ее применять только тогда, когда она действительно нужна. Самая большая проблема, с которой вы столкнетесь при денормализации, заключается в том, что денормализованные данные сложно обновлять.

Давайте взглянем на пример того, как улучшать свои запросы путем денормализации количественных данных. Вы будете выполнять денормализацию данных из своей модели `Image` и использовать сигналы Django, чтобы поддерживать данные в обновленном состоянии.

Работа с сигналами

Django идет в комплекте с диспетчером сигналов, который позволяет функциям-получателям получать уведомления о том, когда происходят те или иные действия. Сигналы очень полезны, когда требуется, чтобы исходный код делал что-то всякий раз, когда происходит что-то еще. Сигналы позволяют отцеплять логику: можно улавливать определенное действие, независимо от приложения или исходного кода, вызвавшего это действие, и реализовывать логику, которая выполняется всякий раз, когда это действие происходит. Например, можно разработать функцию-получатель сигналов, которая будет выполняться всякий раз при сохранении объекта `User`. Также можно создавать свои собственные сигналы, чтобы другие могли получать уведомления, когда происходит событие.

Django предоставляет моделям ряд сигналов, расположенных в `django.db.models.signals`. Вот несколько таких сигналов:

- `pre_save` и `post_save` отправляются до или после вызова метода `save()` модели;
- `pre_delete` и `post_delete` отправляются до или после вызова метода `delete()` модели или объекта QuerySet;
- `m2m_changed` отправляется при изменении поля `ManyToManyField` в модели.

Это всего лишь часть встроенных в Django сигналов. Список всех встроенных сигналов находится на странице https://docs.djangoproject.com/en/4.1/ref/signals/.

Допустим, вы хотите извлекать изображения по популярности. Для того чтобы получать изображения, упорядоченные по числу пользователей, которым они нравятся, можно использовать встроенные в Django функции агрегирования. Напомним, что вы использовали функции агрегирования Django в главе 3 «Расширение приложения для ведения блога». В следующем ниже примере исходного кода изображения будут извлекаться в соответствии с числом их лайков:

```
from django.db.models import Count
from images.models import Image

images_by_popularity = Image.objects.annotate(
    total_likes=Count('users_like')).order_by('-total_likes')
```

Однако с точки зрения производительности упорядочивание изображений путем подсчета их общих количеств лайков (`likes`) будет затратнее, чем упорядочивание их по полю, в котором эти общие количества хранятся. В модель `Image` можно добавить поле, чтобы денормализовать общее количество лайков с целью значительного повышения производительности запросов, в которых это поле используется. Вопрос в том, как поддерживать это поле в обновленном состоянии.

Отредактируйте файл `models.py` приложения `images`, добавив следующее ниже поле `total_likes` в модель `Image`. Новый исходный код выделен жирным шрифтом:

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(default=0)

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

Поле `total_likes` позволит хранить общее количество пользователей, которым понравилось каждое изображение. Денормализация количеств широко применяется тогда, когда нужно по ним фильтровать или упорядочить набор запросов QuerySet. Индекс базы данных по полю `total_likes` был добавлен в убывающем порядке, так как планируется, что изображения будут извлекаться упорядоченными в убывающем порядке по их общему числу лайков.

Перед денормализацией полей следует учитывать, что существует несколько способов улучшения производительности. Прежде чем начинать денормализацию данных, рассмотрите возможность применения индексов базы данных, оптимизации запросов и кеширования.

Выполните следующую ниже команду, чтобы создать миграции для добавления нового поля в таблицу базы данных:

```
python manage.py makemigrations images
```

Вы должны увидеть такой результат:

```
Migrations for 'images':
  images/migrations/0002_auto_20220124_1757.py
    - Add field total_likes to image
    - Create index images_imag_total_l_0bcd7e_idx on field(s) -total_likes of
model image
```

Затем выполните ниже следующую команду, чтобы применить миграцию:

```
python manage.py migrate images
```

Результат должен содержать такую строку:

```
Applying images.0002_auto_20220124_1757... OK
```

Теперь нужно прикрепить функцию-получатель к сигналу `m2m_changed`.

Внутри каталога приложения `images` создайте новый файл и назовите его `signal.py`. Добавьте в него следующий ниже исходный код:

```
from django.db.models.signals import m2m_changed
from django.dispatch import receiver
from .models import Image

@receiver(m2m_changed, sender=Image.users_like.through)
def users_like_changed(sender, instance, **kwargs):
    instance.total_likes = instance.users_like.count()
    instance.save()
```

Сперва, используя декоратор `receiver()`, в качестве функции-получателя регистрируется функция `users_like_changed`. Она привязывается к сигналу `m2m_changed`. Затем эта функция соединяется с `Image.users_like.through`, чтобы функция вызывалась только в том случае, если сигнал `m2m_changed` был запущен этим отправителем. Есть и альтернативный метод регистрации функции-получателя; он состоит в использовании метода `connect()` объекта `Signal`.

Сигналы Django бывают синхронными и блокирующими. Не путайте сигналы с асинхронными заданиями. Однако когда ваш исходный код получает уведомление с помощью сигнала, то можно комбинировать и то, и другое для запуска асинхронных заданий. Вы научитесь создавать асинхронные задания с помощью очереди заданий Celery в главе 8 «Разработка интернет-магазина».

Необходимо соединить функцию-получатель с сигналом, чтобы она вызывалась всякий раз, когда сигнал отправляется. Рекомендуемый метод регистрации своих собственных сигналов состоит в импортировании их в метод `ready()` класса конфигурации вашего приложения. Django предоставляет реестр приложений, который позволяет конфигурировать и контролировать ваши приложения.

Конфигурационные классы приложений

Django позволяет задавать приложениям конфигурационные классы. При создании приложения с помощью команды `startapp` Django добавляет файл `apps.py` в каталог приложения, включая базовую конфигурацию приложения, которая наследует от класса `AppConfig`.

Конфигурационный класс приложения позволяет хранить метаданные и конфигурацию приложения, а также обеспечивает интроспекцию приложения. Дополнительная информация о конфигурациях приложений находится на странице https://docs.djangoproject.com/en/4.1/ref/applications/.

Для того чтобы зарегистрировать свои функции-получатели (`receiver`) сигналов при использовании декоратора `receiver()`, нужно просто импортировать модуль `signal` своего приложения в метод `ready()` конфигурационного класса приложения. Этот метод вызывается сразу после того, как реестр приложений будет полностью заполнен. В указанный метод также должны быть включены любые другие инициализации приложения.

Отредактируйте файл `apps.py` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.apps import AppConfig

class ImagesConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'images'

    def ready(self):
        # импортировать обработчики сигналов
        import images.signals
```

Сигналы для этого приложения импортируются в методе `ready()`, для того чтобы они импортировались при загрузке приложения `images`.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Откройте браузер, чтобы просмотреть страницу детальной информации об изображении, и кликните по кнопке **Like** (Нравится).

Перейдите на сайт администрирования, откройте URL-адрес редактирования изображения, например `http://127.0.0.1:8000/admin/images/image/1/change/`, и посмотрите на атрибут `total_likes`. Вы должны увидеть, что атрибут `total_likes` обновляется общим числом пользователей, которым нравится изображение, как показано ниже:

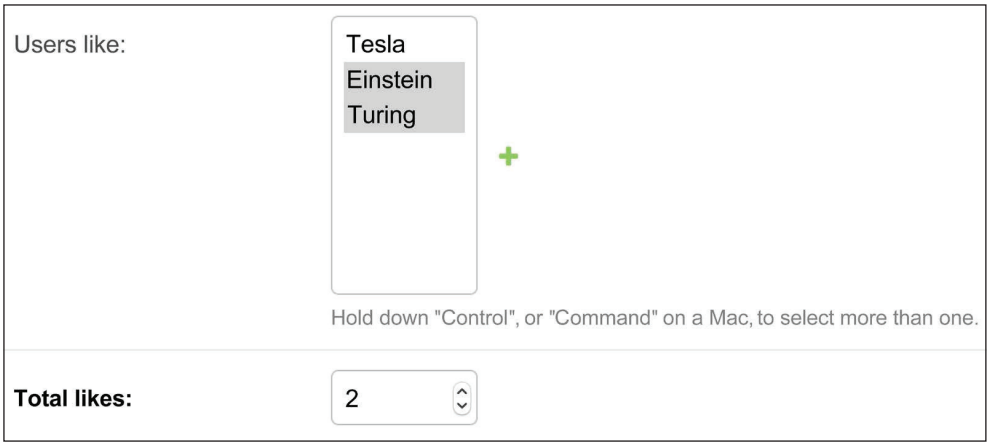


Рис. 7.6. Страница редактирования изображения на сайте администрирования, включая денормализацию общего числа лайков

Теперь можно использовать атрибут `total_likes`, чтобы упорядочивать изображения по популярности или отображать значение в любом месте, избегая сложных запросов для его вычисления.

Рассмотрим следующий ниже запрос, чтобы получать изображения, упорядоченные в убывающем порядке по количеству лайков:

```
from django.db.models import Count

images_by_popularity = Image.objects.annotate(
    likes=Count('users_like')).order_by('-likes')
```

Теперь приведенный выше запрос можно записать следующим образом:

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

Такой подход приводит к менее дорогостоящему SQL-запросу благодаря денормализации общего числа лайков к изображениям. Кроме того, вы научились использовать сигналы Django.

Сигналы следует использовать с осторожностью, поскольку они затрудняют понимание процедуры управления. Во многих случаях можно обойтись без использования сигналов, если известны получатели, которые должны уведомляться.

Теперь необходимо установить начальные значения количеств для остальных объектов `Image`, чтобы они соответствовали текущему состоянию базы данных.

Следующей ниже командой откройте оболочку:

```
python manage.py shell
```

Выполните такой исходный код в оболочке:

```
>>> from images.models import Image
>>> for image in Image.objects.all():
...     image.total_likes = image.users_like.count()
...     image.save()
```

Вы вручную обновили количество лайков для существующих изображений в базе данных. С этого момента функция-получатель сигналов `users_like_changed` будет обрабатывать обновление поля `total_likes` всякий раз, когда изменяются объекты, соединенные взаимосвязью многие-ко-многим.

Далее вы научитесь использовать меню отладочных инструментов Django Debug Toolbar, чтобы получать соответствующую отладочную информацию о запросах, включая время исполнения, исполненные запросы SQL, прорисованные шаблоны, зарегистрированные сигналы и многое другое.

Использование меню отладочных инструментов Django

К этому моменту вы уже знакомы с отладочной страницей Django. В предыдущих главах вы несколько раз видели отличительную желто-серую отладочную страницу. Например, в главе 2 «Совершенствование блога за счет продвинутых функциональностей» в разделе «Обработка ошибок постраничной разбивки» на отладочной странице показывалась информация, связанная с необработанными исключениями при реализации постраничной разбивки объектов.

Отладочная страница Django предоставляет полезную информацию об отладке. Вместе с тем есть приложение Django, которое содержит более детальную отладочную информацию и бывает очень удобным при разработке.

Меню отладочных инструментов Django Debug Toolbar – это внешнее приложение Django, которое позволяет просматривать соответствующую отладочную информацию о текущем цикле запроса/ответа. Информация разделена на несколько панелей, которые отображают различную информацию, включая данные запроса/ответа, используемые версии пакетов Python, время исполнения, настроечные параметры, заголовки, SQL-запросы, применяемые шаблоны, кеш, сигналы и журнальные данные.

Документация по меню отладочных инструментов Django Debug Toolbar находится на странице https://django-debug-toolbar.readthedocs.io/.

Установка меню отладочных инструментов Django

Установите `django-debug-toolbar` через `pip`, используя следующую ниже команду:

```
pip install django-debug-toolbar==3.6.0
```

Отредактируйте файл `settings.py` проекта, добавив `debug_toolbar` в настроечный параметр `INSTALLED_APPS`, как показано ниже. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [
    # ...
    'debug_toolbar',
]
```

В том же файле добавьте следующую ниже строку, выделенную жирным шрифтом, в настроечный параметр `MIDDLEWARE`:

```
MIDDLEWARE = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Меню отладочных инструментов Django Debug Toolbar в основном реализовано как промежуточный программный компонент. Порядок следования

таких компонентов в настроечном параметре MIDDLEWARE имеет значение. Компонент DebugToolbarMiddleware должен располагаться перед любым другим промежуточным компонентом, за исключением промежуточного компонента, кодирующего содержимое ответа, например GZipMiddleware, который, если он присутствует, должен стоять первым.

Добавьте следующие ниже строки в конец файла settings.py:

```
INTERNAL_IPS = [
    '127.0.0.1',
]
```

Меню отладочных инструментов Django будет отображаться только в том случае, если ваш IP-адрес соответствует записи в настроечном параметре INTERNAL_IPS. Для того чтобы предотвратить отображение отладочной информации в производственной среде, ПО меню отладочных инструментов Django Debug Toolbar проверяет, что в настроечном параметре DEBUG установлено значение True.

Отредактируйте главный файл urls.py проекта, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом, в список urlpatterns:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
         include('social_django.urls', namespace='social')),
    path('images/', include('images.urls', namespace='images')),
    path('__debug__/', include('debug_toolbar.urls')),
]
```

Теперь меню отладочных инструментов Django Debug Toolbar установлено в вашем проекте. Давайте попробуем!

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу http://127.0.0.1:8000/images/ в своем браузере. Теперь вы должны увидеть сворачиваемое боковое меню с правой стороны. Оно должна выглядеть следующим образом:

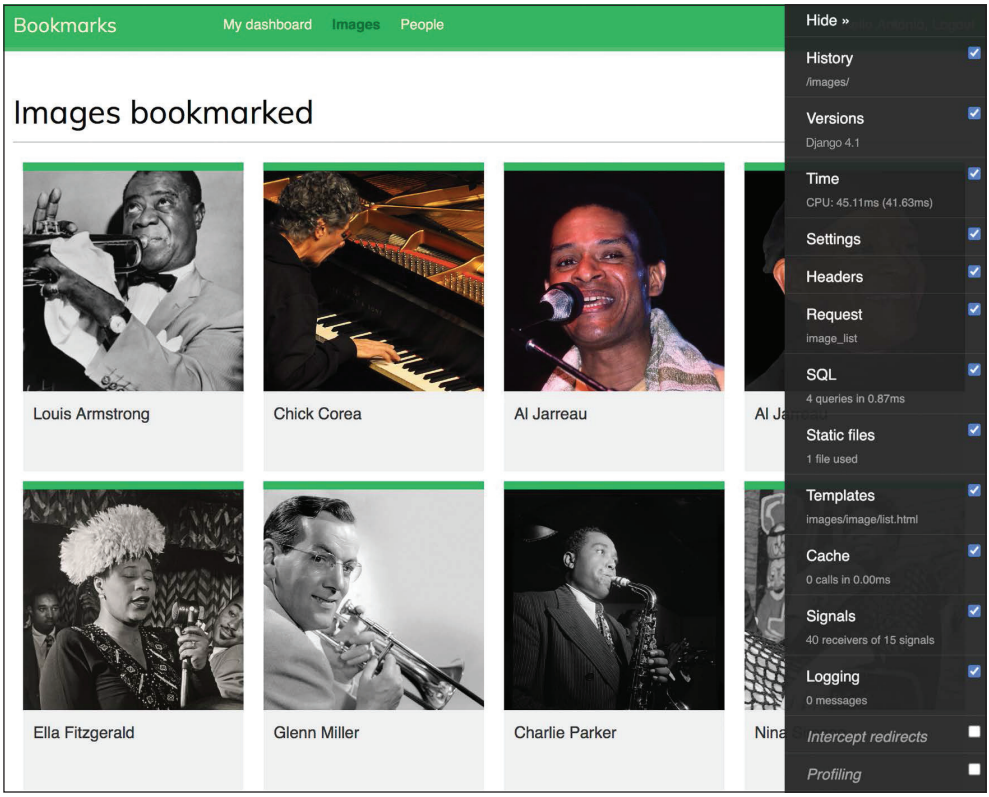


Рис. 7.7. Боковое меню отладочных инструментов Django

Авторство изображений на рис. 7.7:

- Чик Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: https://creativecommons.org/licenses/by/2.0/);
- Эл Джарро – Дюссельдорф, 1981, автор: Эдди Лауманнс, он же RX-Guru (лицензия: Creative Commons Attribution 3.0 Unported: https://creativecommons.org/licenses/by/3.0/);
- Эл Джарро, авторы: Kingkongphoto и www.celebrity-photos.com (лицензия: Creative Commons Attribution-ShareAlike 2.0 Типовая форма: https://creativecommons.org/licenses/by-sa/2.0/).

Если боковое меню отладочных инструментов не появляется, то проверьте журнал консоли оболочки `RunServer`. Если вы видите ошибку MIME-типа, то, скорее всего, ваши файлы соотнесения MIME неверны либо нуждаются в обновлении.

Правильное соотнесение для файлов JavaScript и CSS достигается путем добавления следующих ниже строк в файл `settings.py`:

```
if DEBUG:
    import mimetypes
        mimetypes.add_type('application/javascript', '.js', True)
        mimetypes.add_type('text/css', '.css', True)
```

Панели меню отладочных инструментов Django

Меню отладочных инструментов Django Debug Toolbar содержит несколько панелей, которые упорядочивают отладочную информацию цикла запрос/ответ. Боковое вертикальное меню содержит ссылки на каждую панель, и в любой панели можно использовать флажок, чтобы ее активировать либо деактивировать. Изменение будет применено к следующему запросу. Это бывает удобно, когда та или иная панель не интересует, а вычисление добавляет к запросу слишком много накладных расходов.

Кликните по **Time** (Время) в боковом меню. Вы увидите следующую ниже панель:

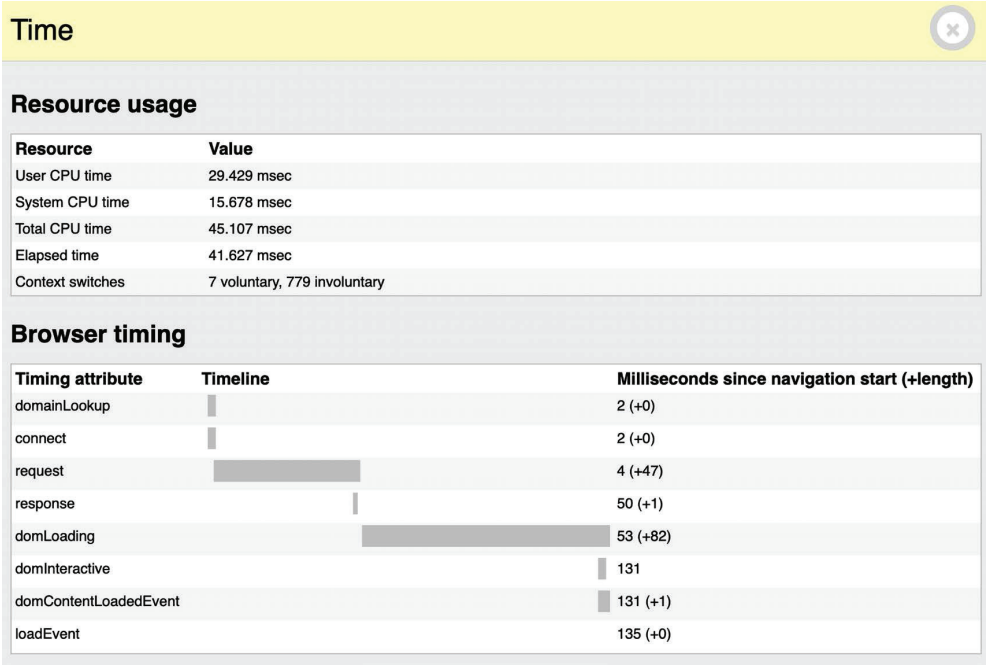


*Рис. 7.8. Панель **Time** – меню отладочных инструментов Django*

Панель **Time** вставляет таймер для разных фаз цикла запроса/ответа. Она также показывает CPU, прошедшее время и количество переключений контекста. Если вы используете Windows, то панель **Time** вы не увидите. В Windows на инструментальной панели доступно и отображается только общее время.

Кликните по **SQL** в боковом меню. Вы увидите следующую ниже панель:

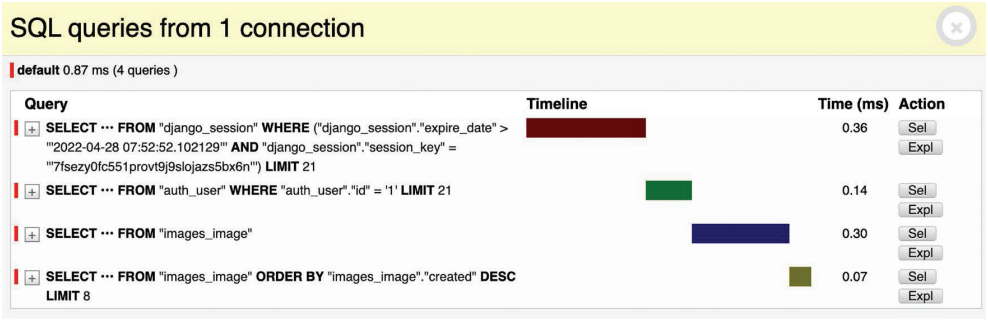


*Рис. 7.9. Панель **SQL** – меню отладочных инструментов Django*

Здесь можно наблюдать различные исполненные SQL-запросы. Эта информация помогает определять ненужные запросы, повторяющиеся запросы, которые можно реиспользовать, или длительные запросы, которые можно оптимизировать. Основываясь на своих находках, вы можете улучшать наборы запросов QuerySets в своих представлениях, при необходимости создавать новые индексы по полям модели или кешировать информацию, когда это необходимо. В этой главе вы научились оптимизировать запросы, которые предусматривают взаимосвязи, используя методы `select_related()` и `prefetch_related()`. Вы научитесь кешировать данные в главе 14 «Прорисовка и кеширование контента».

Кликните по **Templates** (Шаблоны) в боковом меню. Вы увидите следующую ниже панель:

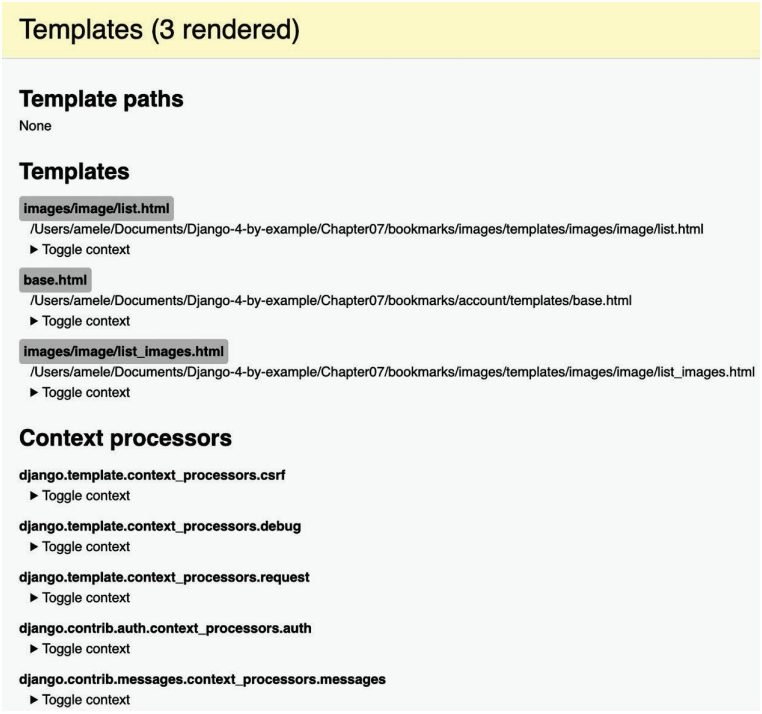
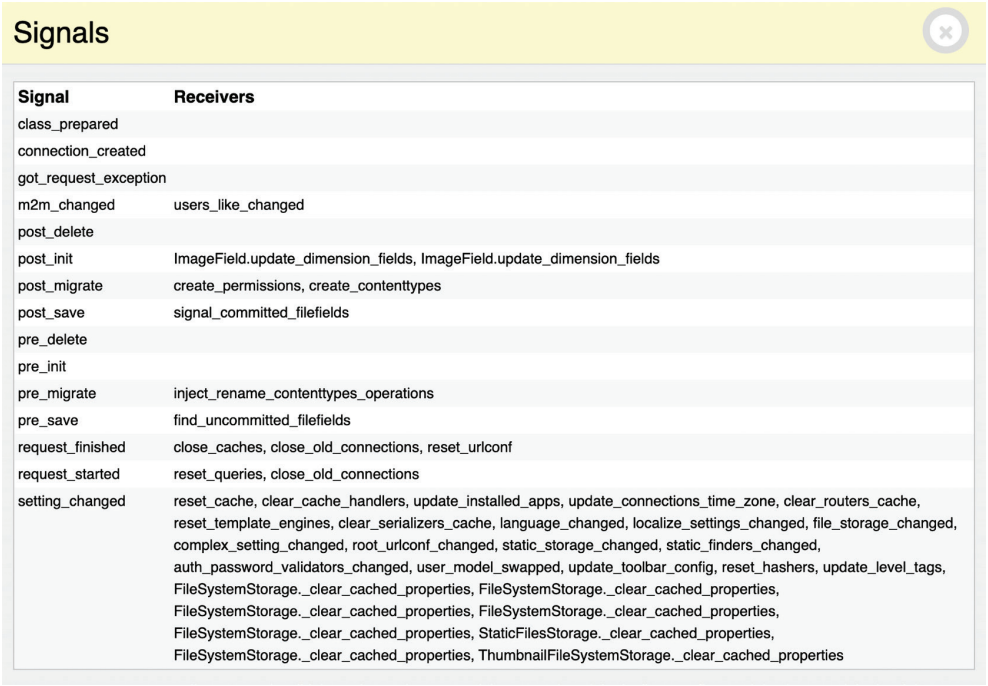


*Рис. 7.10. Панель **Templates** – меню отладочных инструментов Django*

На этой панели показаны различные шаблоны, используемые при прорисовке контента, пути к шаблонам и используемый контекст. Также можно увидеть различные используемые процессоры контекста. Вы узнаете о процессорах контекста в главе 8 «Разработка интернет-магазина».

Кликните по **Signals** (Сигналы) в боковом меню. Вы увидите следующую панель:



Signals

Signal	Receivers
class_prepared	
connection_created	
got_request_exception	
m2m_changed	users_like_changed
post_delete	
post_init	ImageField.update_dimension_fields, ImageField.update_dimension_fields
post_migrate	create_permissions, create_contenttypes
post_save	signal_committed_filefields
pre_delete	
pre_init	
pre_migrate	inject_rename_contenttypes_operations
pre_save	find_uncommitted_filefields
request_finished	close_caches, close_old_connections, reset_urlconf
request_started	reset_queries, close_old_connections
setting_changed	reset_cache, clear_cache_handlers, update_installed_apps, update_connections_time_zone, clear_routers_cache, reset_template_engines, clear_serializers_cache, language_changed, localize_settings_changed, file_storage_changed, complex_setting_changed, root_urlconf_changed, static_storage_changed, static_finders_changed, auth_password_validators_changed, user_model_swapped, update_toolbar_config, reset_hashers, update_level_tags, FileSystemStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, StaticFilesStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, ThumbnailFileSystemStorage._clear_cached_properties

*Рис. 7.11. Панель **Signals** – меню отладочных инструментов Django*

На этой панели можно увидеть все сигналы, которые зарегистрированы в вашем проекте, и функции-получатели, прикрепленные к каждому сигналу. Например, тут можно найти созданную ранее функцию-получатель `users_like_changed`, прикрепленную к сигналу `m2m_changed`. Другие сигналы и получатели являются частью различных приложений Django.

Мы рассмотрели некоторые панели, поставляемые вместе с меню отладочных инструментов Django Debug Toolbar. Помимо встроенных панелей, на странице https://django-debug-toolbar.readthedocs.io/en/latest/panels.html#first-party-panels находятся дополнительные сторонние панели, которые можно загружать и использовать.

Команды меню отладочных инструментов Django

Помимо панелей отладки запросов/ответов, меню отладочных инструментов Django Debug Toolbar предоставляет команду управления по отладке SQL-запросов, исполняемых с помощью Django ORM. Команда управления `debugsqlshell` является репликой команды оболочки Django, но показывает инструкции SQL-запросов.

Следующей ниже командой откройте оболочку:

```
python manage.py debugsqlshell
```

Исполните приведенный далее исходный код:

```
>>> from images.models import Image
>>> Image.objects.get(id=1)
```

Вы увидите такой результат:

```
SELECT "images_image"."id",
       "images_image"."user_id",
       "images_image"."title",
       "images_image"."slug",
       "images_image"."url",
       "images_image"."image",
       "images_image"."description",
       "images_image"."created",
       "images_image"."total_likes"
FROM "images_image"
WHERE "images_image"."id" = 1
LIMIT 21 [0.44ms]
<Image: Django and Duke>
```

Эту команду можно использовать для тестирования ORM-запросов, перед тем как добавлять их в представления. Результирующую инструкцию SQL и время ее исполнения можно проверить для каждого ORM-вызова.

В следующем далее разделе вы научитесь подсчитывать просмотры изображений с помощью резидентной базы данных Redis, которая обеспечивает доступ к данным с низкой задержкой и высокой пропускной способностью.

Подсчет просмотров изображений с помощью хранилища Redis

Redis (Remote Dictionary Server) – это продвинутая база данных в формате ключ-значение, позволяющая хранить различные типы данных. Она также имеет чрезвычайно быстрые операции ввода-вывода. Redis все хранит в памяти, но состояние данных можно сохранять, периодически сбрасывая набор данных на диск или добавляя каждую команду в журнал. Redis очень универсальна по сравнению с другими хранилищами в формате ключ-значение: она предоставляет набор мощных команд и поддерживает различные структуры

данных, такие как строки, хеши, списки, множества, упорядоченные множества и даже битовые карты или структуры данных Hyper-LogLog[1].

Хотя SQL лучше всего подходит для определяемого схемой постоянного хранения данных, Redis предлагает многочисленные преимущества при работе с быстро меняющимися данными, энергозависимым хранилищем или когда требуется быстрое кеширование. Давайте посмотрим, как можно использовать Redis для формирования новых функциональностей в проекте.

Дополнительная информация о базе данных Redis находится на ее домашней странице по адресу https://redis.io/.

База данных Redis предоставляет образ Docker, который упрощает развертывание Redis-сервера со стандартной конфигурацией.

Установка платформы Docker

Docker – это популярная платформа контейнеризации с открытым исходным кодом. Она обеспечивает разработчикам возможность упаковки приложений в контейнеры, упрощая процесс разработки, запуска, управления и распространения приложений.

Сначала скачайте и установите Docker для вашей ОС. Инструкции по скачиванию и установке Docker в Linux, macOS и Windows находятся на странице https://docs.docker.com/get-docker/.

Установка хранилища Redis

После установки платформы Docker на своем компьютере с Linux, macOS или Windows можно легко получить образ Redis платформы Docker. Запустите следующую ниже команду из оболочки:

```
docker pull redis
```

Она скачает образ Redis платформы Docker на локальный компьютер. Информация об официальном образе Redis платформы Docker находится на странице https://hub.docker.com/_/redis. Другие альтернативные способы установки базы данных Redis расположены на странице https://redis.io/download/.

Исполните следующую ниже команду в оболочке, чтобы запустить контейнер Docker для Redis:

---

[1] Битовые карты хранилища Redis – это расширение строкового типа данных, позволяющее обращаться со строковым литералом как с битовым вектором. Hyper-LogLog – это структура данных, которая используется для подсчета количества уникальных элементов в множестве с использованием малого, постоянного объема памяти. – *Прим. перев.*

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Эта команда запускает Redis в контейнере Docker. Опция `-it` сообщает Docker, что можно переходить прямо внутрь контейнера для интерактивного ввода. Параметр `--rm` сообщает Docker, что при выходе из контейнера нужно очищать контейнер и удалять файловую систему автоматически. Опция `--name` используется для назначения контейнеру имени. Опция `-p` применяется для публикации порта 6379, на котором работает Redis, на тот же порт интерфейса хоста. В Redis по умолчанию используется порт 6379.

Вы должны увидеть результат, который заканчивается следующими ниже строками:

```
# Server initialized
* Ready to accept connections
```

Оставьте сервер Redis работать на порту 6379 и откройте еще одну оболочку. Следующей ниже командой запустите клиента Redis:

```
docker exec -it redis sh
```

Вы увидите строку с символом решетки:

```
#
```

Следующей ниже командой запустите клиента Redis:

```
# redis-cli
```

Вы увидите приглашение оболочки клиента Redis. Например:

```
127.0.0.1:6379>
```

Клиент хранилища Redis позволяет исполнять команды Redis непосредственно из оболочки. Давайте попробуем несколько команд. Введите команду `SET` в оболочке Redis, чтобы сохранить значение в ключе:

```
127.0.0.1:6379> SET name "Peter"
OK
```

Приведенная выше команда создает ключ `name` со строковым значением `"Peter"` в хранилище Redis. Результат `OK` указывает на то, что ключ был успешно сохранен.

Затем извлеките значение с помощью команды `GET`, как показано ниже:

```
127.0.0.1:6379> GET name
"Peter"
```

Далее можно проверить существование ключа, используя команду `EXISTS`. Эта команда возвращает `1`, если данный ключ существует, и `0` в противном случае:

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

С помощью команды `EXPIRE` можно установить срок истечения ключа, причем она позволяет устанавливать время жизни ключа в секундах. Еще одним вариантом является использование команды `EXPIREAT`, которая на входе ожидает отметку времени в формате Unix. Срок истечения ключа общепринято использовать, когда Redis применяется в качестве кеша или для хранения волатильных данных:

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379> EXPIRE name 2
(integer) 1
```

Подождите более двух секунд и попробуйте снова получить тот же ключ:

```
127.0.0.1:6379> GET name
(nil)
```

Ответ (`nil`), то есть нулевой ответ, означает, что ключ не найден. Кроме того, любой ключ можно удалить с помощью команды `DEL`, как показано ниже:

```
127.0.0.1:6379> SET total 1
OK
127.0.0.1:6379> DEL total
(integer) 1
127.0.0.1:6379> GET total
(nil)
```

Это всего лишь базовые команды для операций на ключах. Список всех команд Redis находится на странице https://redis.io/commands/, список всех типов данных Redis – на странице https://redis.io/docs/manual/data-types/.

Использование хранилища Redis вместе с Python

Для использования Redis вместе с Python понадобятся привязки. Следующей ниже командой установите `redis-py` посредством `pip`:

```
pip install redis==4.3.4
```

Документация по `redis-py` находится на странице https://redis-py.readthedocs.io/.

Пакет `redis-py` взаимодействует с Redis, предоставляя Python'овский интерфейс, соответствующий синтаксису команд Redis. Следующей ниже командой откройте оболочку Python:

```
python manage.py shell
```

Выполните такой исходный код:

```
>>> import redis
>>> r = redis.Redis(host='localhost', port=6379, db=0)
```

Приведенный выше исходный код создает соединение с базой данных Redis. В Redis базы данных идентифицируются по целочисленному индексу, а не по имени базы данных. По умолчанию клиент подключается к базе данных 0. Число доступных баз данных, на которое Redis рассчитан изначально, равно 16, но это значение можно изменить в конфигурационном файле `redis.conf`.

Далее с помощью оболочки Python установите значение ключа:

```
>>> r.set('foo', 'bar')
True
```

Команда возвращает `True`, указывая на то, что ключ успешно создан. Теперь с помощью команды `get()` можно получить значение ключа повторно:

```
>>> r.get('foo')
b'bar'
```

Как вы заметили из приведенного выше примера, методы Redis подчиняются синтаксису команд Redis.

Давайте интегрируем Redis в проект. Отредактируйте файл `settings.py` проекта `bookmarks`, добавив в него следующие ниже настроечные параметры:

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0
```

Это настроечные параметры Redis-сервера и базы данных, которую вы будете использовать для своего проекта.

Хранение просмотров изображений в хранилище Redis

Давайте найдем способ сохранить общее число просмотров изображения. Если реализовать это с помощью Django ORM, то такая реализация будет предусматривать SQL-запрос `UPDATE` всякий раз, когда изображение отображается на странице.

Если вместо этого использовать Redis, то просто нужно увеличивать хранящийся в памяти счетчик, что будет приводить к гораздо большей производительности и меньшим накладным расходам.

Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код после существующих инструкций `import`:

```
import redis
from django.conf import settings

# соединить с redis
r = redis.Redis(host=settings.REDIS_HOST,
                port=settings.REDIS_PORT,
                db=settings.REDIS_DB)
```

В приведенном выше исходном коде устанавливается соединение с базой данных Redis, чтобы использовать ее в своих представлениях. Отредактируйте файл `views.py` приложения `images`, видоизменив представление `image_detail`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # увеличить общее число просмотров изображения на 1
    total_views = r.incr(f'image:{image.id}:views')
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

В этом представлении используется команда `incr`, которая увеличивает значение данного ключа на 1. Если ключ не существует, то команда `incr` его создает. Метод `incr()` возвращает окончательное значение ключа после выполнения операции. Его значение сохраняется в переменной `total_views`, которая затем передается в контекст шаблона. Ключ Redis создается, используя формат `object-type:id:field` (например, `image:33:id`).

По традиции при именовании ключей Redis знак двоеточия используется в качестве разделителя для создания ключей в именном пространстве (или пространстве имен). Благодаря этому имена ключей становятся весьма подробными, а родственные ключи имеют в своих именах общие части одной и той же схемы.

Отредактируйте шаблон `images/image/detail.html` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
...
<div class="image-info">
  <div>
    <span class="count">
      <span class="total">{{ total_likes }}</span>
      like{{ total_likes|pluralize }}
    </span>
    <span class="count">
      {{ total_views }} view{{ total_views|pluralize }}
    </span>
    <a href="#" data-id="{{ image.id }}" data-action="{% if request.user in
users_like %}un{% endif %}like"
    class="like button">
      {% if request.user not in users_like %}
        Like
      {% else %}
        Unlike
      {% endif %}
    </a>
  </div>
  {{ image.description|linebreaks }}
</div>
...
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Откройте страницу детальной информации об изображении в своем браузере и перезагрузите ее несколько раз. Вы увидите, что всякий раз, когда представление обрабатывается, общее количество отображаемых просмотров увеличивается на 1. Взгляните на следующий ниже пример:

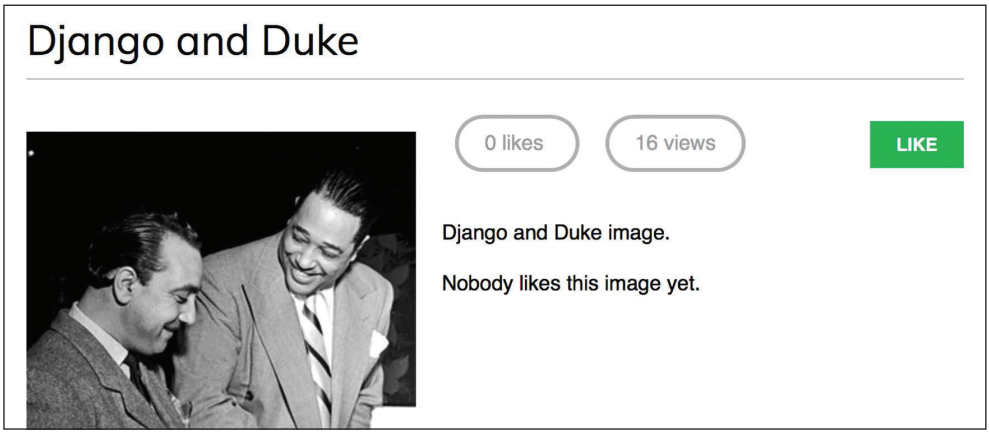


Рис. 7.12. Страница детальной информации об изображении, включающая количество лайков и просмотров

Отлично! Вы успешно интегрировали Redis в проект с целью подсчета просмотров изображений. В следующем разделе вы научитель формировать рейтинг самых просматриваемых изображений с помощью Redis.

Хранение рейтинга в хранилище Redis

Теперь с помощью хранилища Redis мы создадим что-то посложнее. Мы будем использовать Redis для хранения рейтинга самых просматриваемых изображений на платформе. Для этого мы будем использовать сортированные множества Redis. Сортированное множество – это неповторяющаяся коллекция строковых значений, в которой каждый элемент ассоциирован с баллом. Элементы сортируются по их баллу.

Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в представление `image_detail`:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # увеличить общее число просмотров изображения на 1
    total_views = r.incr(f'image:{image.id}:views')
    # увеличить рейтинг изобажения на 1
    r.zincrby('image_ranking', 1, image.id)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

Команда `zincrby()` используется для сохранения просмотров изображений в сортированном множестве с ключом `image:ranking`. В нем будут храниться `id` изображения и соответствующий балл, равный 1, который будет добавлен к общему баллу этого элемента сортированного множества. Такой подход позволит отслеживать все просмотры изображений в глобальном масштабе и иметь сортированное множество, упорядоченное по общему числу просмотров.

Теперь создайте новое представление, чтобы отображать рейтинг наиболее просматриваемых изображений. Добавьте следующий ниже исходный код в файл `views.py` приложения `images`:

```
@login_required
def image_ranking(request):
    # получить словарь рейтинга изображений
    image_ranking = r.zrange('image_ranking', 0, -1,
                             desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # получить наиболее просматриваемые изображения
    most_viewed = list(Image.objects.filter(
                           id__in=image_ranking_ids))
    most_viewed.sort(key=lambda x: image_ranking_ids.index(x.id))
    return render(request,
                  'images/image/ranking.html',
                  {'section': 'images',
                   'most_viewed': most_viewed})
```

Представление `image_ranking` работает следующим образом.

1. Для получения элементов сортированного множества используется команда `zrange()`. Эта команда ожидает конкретно-прикладной диапазон в соответствии с самым низким и самым высоким баллами. Используя 0 в качестве наименьшего значения и –1 в качестве наибольшего, базе данных Redis сообщается, что нужно вернуть все элементы сортированного множества. Для извлечения элементов, упорядоченных по убыванию балла, также указывается параметр `desc=True`. Наконец, результаты нарезаются, используя `[:10]`, чтобы получить первые 10 элементов с наивысшим баллом.
2. Создается список возвращаемых ИД изображений и сохраняется в переменной `image_ranking_ids` в виде списка целых чисел. По этим идентификаторам извлекаются объекты `Image`, и с помощью функции `list()` запрос принудительно исполняется. Важно вызвать принудительное исполнение набора запросов QuerySet, потому что для него будет использоваться списковый метод `sort()` (на этом этапе вместо набора запросов QuerySet нужен список объектов).

3. Объекты `Image` сортируются по индексу их появления в рейтинге изображений. Теперь в шаблоне можно использовать список `most_viewed`, чтобы отобразить 10 самых просматриваемых изображений.

Внутри каталога `images/image/template` приложения `images` создайте новый шаблон `rating.html` и добавьте в него следующий ниже исходный код:

```
{% extends "base.html" %}

{% block title %}Images ranking{% endblock %}

{% block content %}
  <h1>Images ranking</h1>
  <ol>
    {% for image in most_viewed %}
      <li>
        <a href="{{ image.get_absolute_url }}">
          {{ image.title }}
        </a>
      </li>
    {% endfor %}
  </ol>
{% endblock %}
```

Этот шаблон довольно простой. Содержащиеся в списке `most_viewed` объекты `Image` прокручиваются в цикле, и их имена отображаются, включая ссылку на страницу детальной информации об изображении.

Наконец, необходимо создать шаблон URL-адреса для нового представления. Отредактируйте файл `urls.py` приложения `images`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>/',
         views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
    path('', views.image_list, name='list'),
    path('ranking/', views.image_ranking, name='ranking'),
]
```

Запустите сервер разработки, обратитесь к своему сайту из своего веб-браузера и загрузите страницу детальной информации об изображении несколько раз с разными изображениями. Затем обратитесь к URL-адресу `http://127.0.0.1:8000/images/ranking/` из своего браузера. Вы должны увидеть рейтинг изображений, как показано ниже:

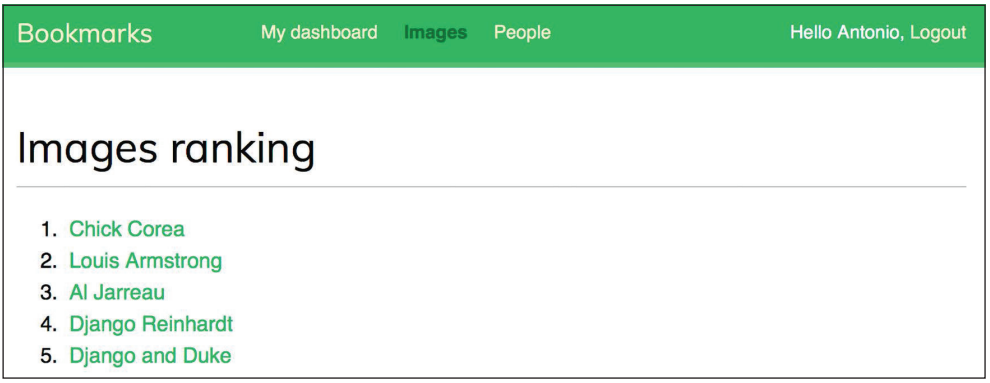


Рис. 7.13. Страница рейтинга, сформированная на основе данных, извлеченных из Redis

Отлично! Вы только что создали рейтинг с помощью Redis.

Следующие шаги с Redis

Redis не является заменой базы данных SQL, но предлагает быстрое резидентное хранилище, которое больше подходит для определенных задач. Добавьте его в свой стек и используйте тогда, когда действительно чувствуете, что это необходимо. Ниже приведено несколько сценариев, в которых Redis бывает полезен.

- **Подсчет количеств**: как вы уже убедились, с помощью Redis очень легко управлять количествами. Команды `incr()` и `incrby()` можно использовать для подсчета элементов.
- **Хранение последних элементов**: с помощью команд `lpush()` и `rpush()` можно добавлять элементы в начало/конец списка. Удаление и возврат первого/последнего элемента осуществляются посредством команд `lpop()`/`rpop()`. Длину списка можно обрезать с использованием `ltrim()`, чтобы поддерживать заданную длину.
- **Очереди**: в дополнение к командам `push` и `pop` Redis предлагает блокирование команд очереди.
- **Кеширование:** применение команд `expire()` и `expireat()` позволяет использовать Redis в качестве кеша. Кроме того, существуют сторонние для Django механизмы кеширования Redis.
- **Публикация/подписка**: Redis предоставляет команды для подписки/отписки и отправки сообщений в каналы.
- **Рейтинги и списки лидеров**: сортированные множества Redis с баллами позволяют очень легко создавать списки лидеров.
- **Отслеживание в режиме реального времени**: благодаря быстрому вводу-выводу Redis идеально подходит для реально-временных сценариев.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter07.
- Конкретно-прикладные модели пользователей: https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#specifying-a-custom-user-model.
- Фреймворк `contenttypes`: https://docs.djangoproject.com/en/4.1/ref/contrib/contenttypes/.
- Встроенные в Django сигналы: https://docs.djangoproject.com/en/4.1/ref/signals/.
- Конфигурационные классы приложений: https://docs.djangoproject.com/en/4.1/ref/applications/.
- Документация по меню отладочных инструментов Django Debug Toolbar: https://django-debug-toolbar.readthedocs.io/.
- Сторонние панели для меню отладочных инструментов Django Debug Toolbar: https://django-debug-toolbar.readthedocs.io/en/latest/panels.html#third-party-panels.
- Резидентное хранилище данных Redis: https://redis.io/.
- Инструкции по скачиванию и установке платформы Docker: https://docs.docker.com/get-docker/.
- Официальный образ Redis платформы Docker: https://hub.docker.com/_/redis.
- Опции скачивания Redis: https://redis.io/download/.
- Команды Redis: https://redis.io/commands/.
- Типы данных Redis: https://redis.io/docs/manual/data-types/.
- Документация по пакету `redis-py`: https://redis-py.readthedocs.io/.

Резюме

В этой главе вы разработали систему подписки, используя взаимосвязи многие-ко-многим с промежуточной моделью. Вы также создали поток активности, применяя обобщенные отношения, и оптимизировали наборы запросов с целью извлечения связанных объектов. Затем вы ознакомились с сигналами Django и создали функцию-получатель сигналов с целью денормализации количеств связанных объектов. Рассмотрели классы конфигурации приложений, которые использовались для загрузки обработчиков сигналов. Вы добавили меню отладочных инструментов Django Debug Toolbar в проект. Вы также научились устанавливать в свой проект и конфигурировать резидентное хранилище Redis. Наконец, вы применили Redis в проекте, чтобы хранить просмотры элементов, и с помощью нее сформировали рейтинг изображений.