

8

Разработка интернет-магазина

В предыдущей главе вы создали систему подписки и разработали поток активности пользователей. Вы также узнали, как работают сигналы Django, и интегрировали Redis в свой проект, чтобы вести подсчет просмотров изображений.

В этой главе вы начнете новый проект Django, состоящий из полнофункционального интернет-магазина. Данная и следующие две главы покажут, как разрабатывать ключевые функциональности платформы электронной коммерции. Ваш интернет-магазин предоставит клиентам возможность просматривать товары, добавлять их в корзину, применять коды скидочных, проходить процесс оформления платежа, оплачивать кредитной картой и получать счета-фактуры. Вы также имплементируете рекомендательный механизм, чтобы рекомендовать товары своим клиентам, и будете использовать интернационализацию, чтобы предлагать свой сайт на нескольких языках.

В этой главе научитесь:

- создавать каталог товаров;
- создавать корзину покупок, используя сеансы Django;
- создавать конкретно-прикладные процессоры контекста;
- управлять заказами клиентов;
- конфигурировать в своем проекте очередь заданий Celery с помощью брокера сообщений RabbitMQ;
- отправлять асинхронные уведомления клиентам с помощью Celery;
- проводить мониторинг Celery посредством инструмента Flower.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter08>.

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`.

Создание проекта интернет-магазина

Давайте начнем с нового проекта Django по разработке интернет-магазина. Ваши пользователи смогут просматривать каталог товаров и добавлять товары в корзину. Наконец, они смогут оформлять и размещать заказ. В этой главе будут рассмотрены следующие ниже функциональности интернет-магазина:

- создание моделей каталога товаров, добавление их на административный сайт и разработка базовых представлений для отображения каталога;
- разработка системы корзины покупок с использованием сеансов Django, предоставляющей пользователям возможность сохранять выбранные товары во время просмотра сайта;
- создание формы и функциональности размещения заказов на сайте;
- отправка пользователям асинхронного электронного письма о подтверждении заказа после его размещения.

Откройте оболочку и примените следующую ниже команду, чтобы создать новую виртуальную среду для этого проекта в каталоге `env/`:

```
python -m venv env/myshop
```

Если вы используете Linux или macOS, то выполните следующую ниже команду, чтобы активировать виртуальную среду:

```
source env/myshop/bin/activate
```

Если же вы используете Windows, то вместо этого примените следующую ниже команду:

```
.\env\myshop\Scripts\activate
```

Приглашение оболочки отобразит вашу активную виртуальную среду, как показано ниже:

```
(myshop)laptop:~ zenx$
```

Следующей ниже командой установите Django в вашей виртуальной среде:

```
pip install Django~=4.1.0
```

Запустите новый проект под названием `myshop` с приложением под названием `shop`, открыв оболочку и выполнив такую команду:

```
django-admin startproject myshop
```

Создана начальная структура проекта. Примените следующие ниже команды, чтобы попасть в каталог проекта и создать новое приложение с именем `shop`:

```
cd myshop/  
django-admin startapp shop
```

Отредактируйте файл `settings.py`, добавив в список `INSTALLED_APPS` следующую ниже строку, выделенную жирным шрифтом:

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

Теперь приложение в этом проекте является активным. Давайте определим модели для каталога товаров.

Создание моделей каталога товаров

Каталог магазина будет состоять из товаров, разделенных на разные категории. У каждого товара будет имя, опциональное описание, опциональное изображение, цена и наличие.

Отредактируйте файл `models.py` только что созданного вами приложения `shop`, добавив следующий ниже исходный код:

```
from django.db import models  
  
class Category(models.Model):  
    name = models.CharField(max_length=200)  
    slug = models.SlugField(max_length=200,  
                            unique=True)  
  
    class Meta:  
        ordering = ['name']  
        indexes = [  
            models.Index(fields=['name']),  
        ]  
        verbose_name = 'category'  
        verbose_name_plural = 'categories'  
  
    def __str__(self):  
        return self.name  
  
class Product(models.Model):
```

```

category = models.ForeignKey(Category,
                              related_name='products',
                              on_delete=models.CASCADE)
name = models.CharField(max_length=200)
slug = models.SlugField(max_length=200)
image = models.ImageField(upload_to='products/%Y/%m/%d',
                           blank=True)
description = models.TextField(blank=True)
price = models.DecimalField(max_digits=10,
                             decimal_places=2)
available = models.BooleanField(default=True)
created = models.DateTimeField(auto_now_add=True)
updated = models.DateTimeField(auto_now=True)

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

def __str__(self):
    return self.name

```

Это модели Category и Product. Модель Category состоит из поля name и уникального поля slug (уникальность подразумевает создание индекса). В Meta-классе модели Category определен индекс по полю name.

Поля модели Product таковы:

- category: внешний ключ (ForeignKey) к модели Category. Это взаимосвязь один-ко-многим: товар принадлежит одной категории, а категория содержит несколько товаров;
- name: название товара;
- slug: слаг этого товара для создания красивых URL-адресов;
- image: опциональное изображение товара;
- description: опциональное описание товара;
- price: в этом поле используется тип Python decimal.Decimal, чтобы хранить десятичное число фиксированной точности. Максимальное количество цифр (включая десятичные разряды) устанавливается с помощью атрибута max_digits, а десятичных разрядов – с помощью атрибута decimal_places;
- available: булево значение, указывающее на наличие или отсутствие товара. Оно будет использоваться для активирования/деактивирования товара в каталоге;

- `created`: в этом поле хранится информация о дате/времени создания объекта;
- `updated`: в этом поле хранится информация о дате/времени обновления объекта в последний раз.

Во избежание проблем с округлением в поле `price` вместо типа `FloatField` используется тип `DecimalField`.



Для хранения значений денежных сумм всегда следует использовать `DecimalField`. Внутри `FloatField` применяется Python'овский тип `float`, тогда как внутри `DecimalField` – Python'овский тип `Decimal`. Используя тип `Decimal`, вы избежите проблем с округлением чисел с плавающей запятой.

В Meta-классе модели `Product` был определен многопольный индекс по полям `id` и `slug`. Оба поля индексируются вместе с целью повышения производительности запросов, в которых эти два поля используются.

Планируется, что товары будут запрашиваться как по идентификатору, так и по слагю. Добавлен индекс по полю `name` и индекс по полю `created`. Перед именем поля использован дефис, чтобы определить индекс в убывающем порядке.

На рис. 8.1 показаны две созданные модели данных:

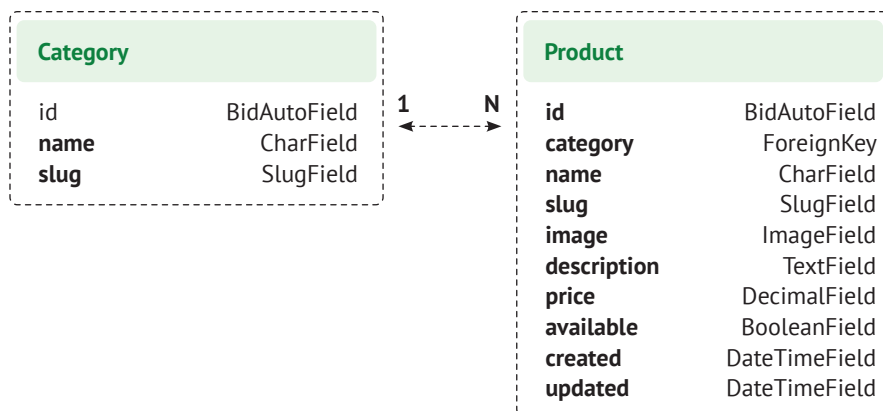


Рис. 8.1. Модели каталога товаров

На рис. 8.1 показаны различные поля моделей данных и взаимосвязи один-ко-многим между моделями `Category` и `Product`.

Результатом этих моделей будут следующие ниже таблицы базы данных, которые показаны на рис. 8.2.

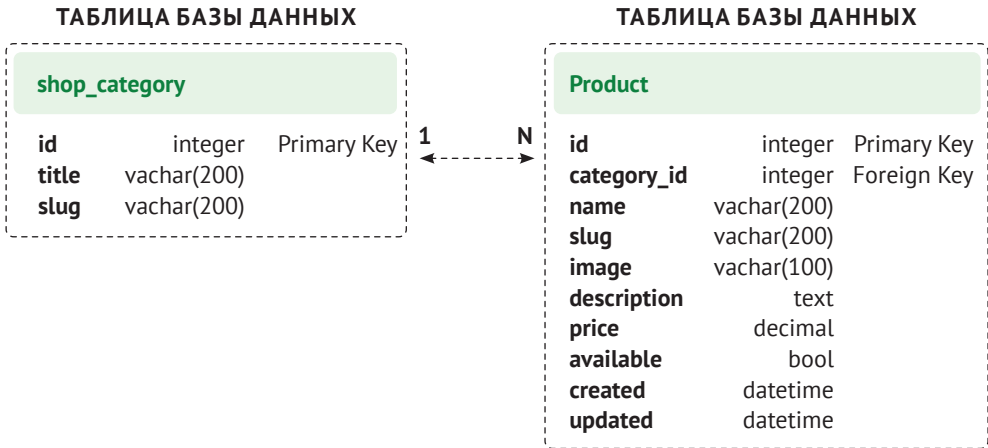


Рис. 8.2. Таблицы базы данных для моделей каталога товаров

Взаимосвязь один-ко-многим между обеими таблицами определяется полем `category_id` в таблице `shop_product`, которая используется для хранения ИД связанной категории по каждому объекту `Product`.

Давайте создадим первоначальные миграции базы данных для приложения `shop`. Поскольку в своих моделях вы собираетесь работать с изображениями, необходимо установить библиотеку `Pillow`. Напомним, что в главе 4 «Разработка социального веб-сайта» вы научились устанавливать библиотеку `Pillow`, для того чтобы с ее помощью управлять изображениями. Откройте оболочку и следующей ниже командой установите `Pillow`:

```
pip install Pillow==9.2.0
```

Теперь выполните следующую команду, чтобы создать начальные миграции для проекта:

```
python manage.py makemigrations
```

Вы увидите такой результат:

```
Migrations for 'shop':
  shop/migrations/0001_initial.py
    - Create model Category
    - Create model Product
    - Create index shop_catego_name_289c7e_idx on field(s) name of model
category
    - Create index shop_produc_id_f21274_idx on field(s) id, slug of model
product
    - Create index shop_produc_name_a2070e_idx on field(s) name of model
product
    - Create index shop_produc_created_ef211c_idx on field(s) -created of model
product
```

Выполните следующую ниже команду, чтобы синхронизировать базу данных:

```
python manage.py migrate
```

Вы увидите результат, который содержит такую строку:

```
Applying shop.0001_initial... OK
```

Теперь база данных синхронизирована с вашими моделями.

Регистрация моделей каталога на сайте администрирования

Давайте добавим ваши модели на сайт администрирования, чтобы легко управлять категориями и товарами. Отредактируйте файл `admin.py` приложения `shop`, добавив следующий ниже исходный код:

```
from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price',
                  'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
```

Напомним, что атрибут `prepopulated_fields` используется для того, чтобы указывать поля, значение которых устанавливается автоматически с использованием значения других полей. Как вы уже убедились, это удобно для генерирования слаггов.

Атрибут `list_editable` в классе `ProductAdmin` используется для того, чтобы задать поля, которые можно редактировать, находясь на странице отображения списка на сайте администрирования. Такой подход позволит редактировать несколько строк одновременно. Любое поле в `list_editable` также должно быть указано в атрибуте `list_display`, поскольку редактировать можно только отображаемые поля.

Теперь следующей ниже командой создайте сайт суперпользователя:

```
python manage.py createsuperuser
```

Введите желаемое пользовательское имя, адрес электронной почты и пароль. Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/shop/product/add/> в своем браузере и войдите под именем пользователя, которого вы только что создали. Добавьте новую категорию и товар, используя интерфейс администрирования. Форма для добавления товара должна выглядеть следующим образом:

The image shows a web form titled "Add product". It contains the following elements:

- Category:** A dropdown menu with "Tea" selected and a plus sign icon.
- Name:** A text input field containing "Green tea".
- Slug:** A text input field containing "green-tea".
- Image:** A "Choose File" button and the text "no file selected".
- Description:** A large text area containing placeholder text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."
- Price:** A numeric input field containing "30.00".
- Available:** A checked checkbox.
- Buttons:** Three buttons at the bottom right: "Save and add another", "Save and continue editing", and "SAVE".

Рис. 8.3. Форма для создания товара

Кликните по кнопке **Save** (Сохранить). Страница списка изменения товара на странице администрирования будет выглядеть следующим образом:

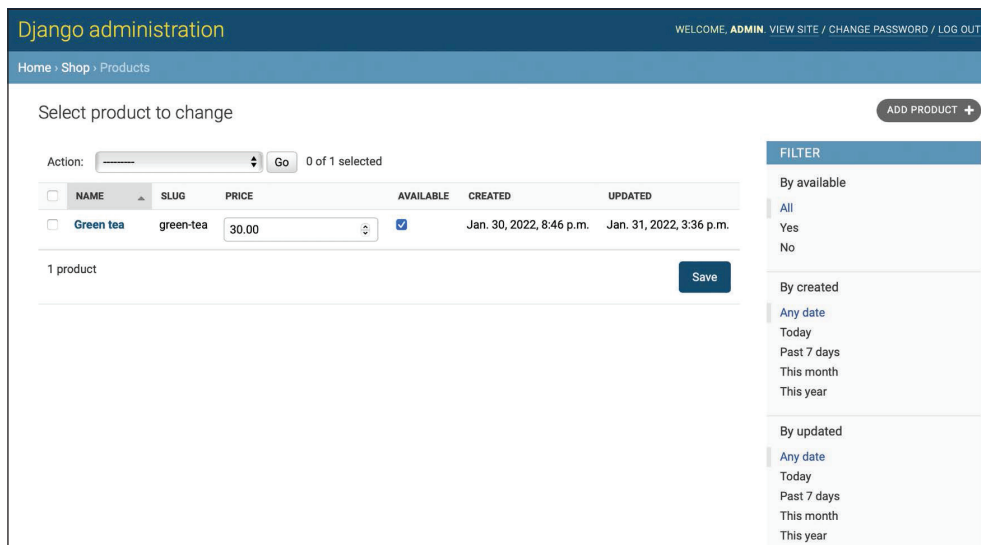


Рис. 8.4. Страница списка изменения товара

Формирование представлений каталога

Для того чтобы отобразить каталог товаров на странице, необходимо создать представление перечисления списка всех товаров или фильтрации товаров по заданной категории. Отредактируйте файл `views.py` приложения `shop`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        category = get_object_or_404(Category,
                                     slug=category_slug)
        products = products.filter(category=category)
    return render(request,
                  'shop/product/list.html',
                  {'category': category,
                  'categories': categories,
                  'products': products})
```

В приведенном выше исходном коде набор запросов `QuerySet` фильтруется с параметром `available=True`, чтобы получать только те товары, которые имеются в наличии. Опциональный параметр `category_slug` используется для дополнительной фильтрации товаров по заданной категории.

Еще требуется представление извлечения и отображения одного товара. Добавьте следующее ниже представление в файл `views.py`:

```
def product_detail(request, id, slug):
    product = get_object_or_404(Product,
                                id=id,
                                slug=slug,
                                available=True)

    return render(request,
                  'shop/product/detail.html',
                  {'product': product})
```

На вход в представление `product_detail` должны передаваться параметры `id` и `slug`, чтобы извлекать экземпляр класса `Product`. Указанный экземпляр можно получить только по ИД, так как это уникальный атрибут. Однако в URL-адрес еще включается слаг, чтобы формировать дружественные для поисковой оптимизации URL-адреса товаров.

После разработки представлений списка товаров и детальной информации о товаре для них необходимо определить шаблоны URL-адресов. Создайте новый файл в каталоге приложения `shop` и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

```
from django.urls import path
from . import views

app_name = 'shop'

urlpatterns = [
    path('', views.product_list, name='product_list'),
    path('<slug:category_slug>/', views.product_list,
         name='product_list_by_category'),
    path('<int:id>/<slug:slug>/', views.product_detail,
         name='product_detail'),
]
```

Это шаблоны URL-адресов для каталога товаров. Для представления `product_list` здесь определено два разных шаблона URL-адреса: шаблон с именем `product_list`, который вызывает представление `product_list` без каких-либо параметров, и шаблон с именем `product_list_by_category`, который передает представлению параметр `category_slug`, чтобы фильтровать товары в соответствии с заданной категорией. Для представления `product_detail` был до-

бавлен шаблон, который передает в него параметры `id` и `slug`, чтобы извлекать конкретный товар.

Отредактируйте файл `urls.py` проекта `myshop`, придав ему следующий вид:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('shop.urls', namespace='shop')),
]
```

В главные шаблоны URL-адресов проекта вставляются URL-адреса приложения `shop` в конкретно-прикладном именном пространстве под тем же именем `shop`.

Далее отредактируйте файл `models.py` приложения `shop`, импортировав функцию `reverse()` и добавив метод `get_absolute_url()` в модели `Category` и `Product`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.db import models
from django.urls import reverse

class Category(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_list_by_category',
                       args=[self.slug])

class Product(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_detail',
                       args=[self.id, self.slug])
```

Как вы уже знаете, `get_absolute_url()` – это общепринятый способ получения URL-адреса заданного объекта.

Здесь используются шаблоны URL-адресов, которые были только что определены в файле `urls.py`.

Создание шаблонов каталога

Теперь нужно создать шаблоны для представлений списка товаров и детальной информации о товаре. Внутри каталога приложения `shop` создайте следующую ниже структуру каталогов и файлов:

```
templates/  
  shop/  
    base.html  
  product/  
    list.html  
    detail.html
```

Необходимо определить базовый шаблон, а затем расширить его в шаблонах списка товаров и детальной информации о товаре. Отредактируйте шаблон `shop/base.html`, добавив следующий ниже исходный код:

```
{% load static %}  
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="utf-8" />  
    <title>{% block title %}My shop{% endblock %}</title>  
    <link href="{% static 'css/base.css' %}" rel="stylesheet">  
  </head>  
  <body>  
    <div id="header">  
      <a href="/" class="logo">My shop</a>  
    </div>  
    <div id="subheader">  
      <div class="cart">  
        Your cart is empty.  
      </div>  
    </div>  
    <div id="content">  
      {% block content %}  
      {% endblock %}  
    </div>  
  </body>  
</html>
```

Это базовый шаблон, который будет использоваться для магазина. Для того чтобы включить используемые в шаблонах стили CSS и изображения, необходимо скопировать сопровождающие эту главу статические файлы, которые находятся в каталоге `static/` приложения `shop`. Скопируйте их в то же место в своем проекте. Содержимое каталога находится на странице <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter08/myshop/shop/static>.

Отредактируйте шаблон `shop/product/list.html`, добавив следующий ниже исходный код:

```
{% extends "shop/base.html" %}  
{% load static %}
```

```

{% block title %}
  {% if category %}{ category.name }{% else %}Products{% endif %}
{% endblock %}

{% block content %}
  <div id="sidebar">
    <h3>Categories</h3>
    <ul>
      <li {% if not category %}class="selected"{% endif %}>
        <a href="{% url "shop:product_list" %}">All</a>
      </li>
      {% for c in categories %}
        <li {% if category.slug == c.slug %}class="selected"
          {% endif %}>
          <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
  <div id="main" class="product-list">
    <h1>{% if category %}{ category.name }{% else %}Products
    {% endif %}</h1>
    {% for product in products %}
      <div class="item">
        <a href="{{ product.get_absolute_url }}">
          
        </a>
        <a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
        <br>
        ${{ product.price }}
      </div>
    {% endfor %}
  </div>
{% endblock %}

```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон списка товаров. Он расширяет шаблон `shop/base.html` и использует контекстную переменную `categories`, чтобы отображать все категории на боковой панели, а также `products` для отображения товаров на текущей странице. Один и тот же шаблон используется как для списка всех имеющихся в наличии товаров, так и для списка товаров, отфильтрованных по категории. Поскольку поле `image` модели `Product` может быть пустым, необходимо предоставить типовое изображение для тех товаров, у которых нет изображения.

Это изображение находится в каталоге статических файлов с относительным путем `img/no_image.png`.

Поскольку для хранения изображений товаров используется тип `ImageField`, для раздачи скачанных файлов изображений нужен сервер разработки.

Отредактируйте файл `settings.py` проекта `myshop`, добавив следующие ниже настроечные параметры:

```
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Параметр `MEDIA_URL` – это базовый URL-адрес, по которому раздаются закачанные пользователями медиафайлы. Параметр `MEDIA_ROOT` – это локальный путь, по которому эти файлы находятся и который следует формировать динамически, добавляя значение переменной `BASE_DIR` в качестве префикса.

Для того чтобы Django раздавал закачанные медиафайлы с помощью сервера разработки, отредактируйте файл `mainurls.py` проекта `myshop`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('shop.urls', namespace='shop')),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Напомним, что во время разработки статические файлы раздаются только таким образом. В производственной среде никогда не следует раздавать статические файлы с помощью Django; сервер разработки Django не способен эффективно раздавать статические файлы. В главе 17 «Выход в прямой эфир» будет объяснено, как раздавать статические файлы в производственной среде.

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Добавьте пару товаров в свой магазин с помощью сайта администрирования и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Вы увидите страницу списка товаров, которая будет выглядеть примерно так:

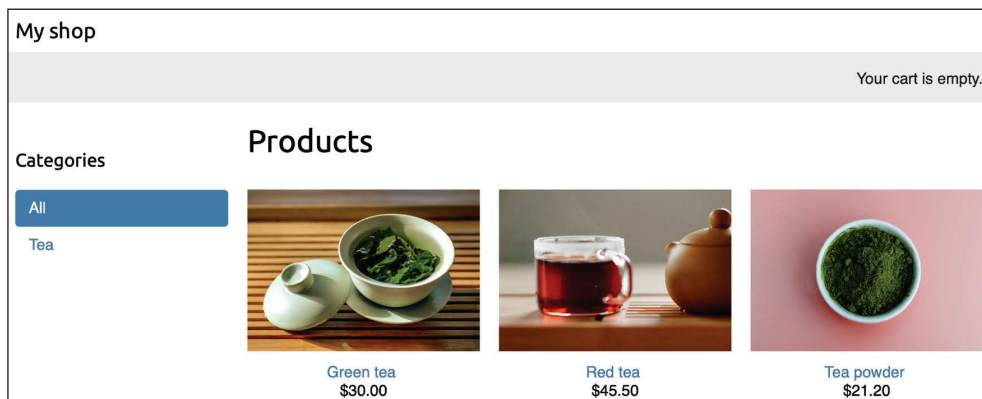
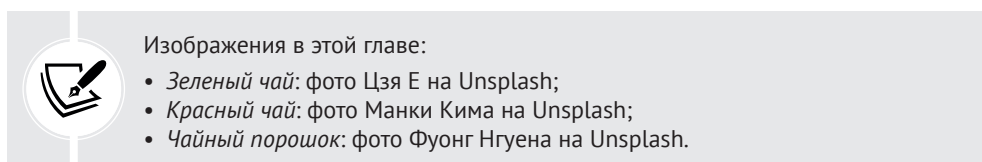


Рис. 8.5. Страница списка товаров



Если создать товар с помощью сайта администрирования и не загрузить его изображение, то вместо него по умолчанию будет отображаться изображение `no_image.png`:

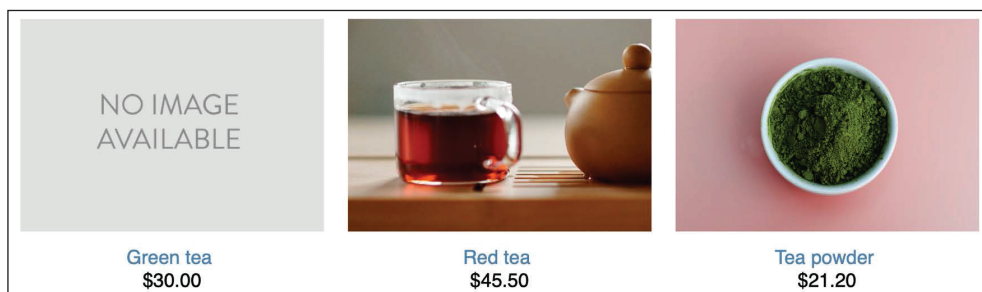


Рис. 8.6. Список товаров, отображающий типовое изображение, используемое для товаров, у которых нет изображения

Отредактируйте шаблон `shop/product/detail.html`, добавив следующий ниже исходный код:

```
{% extends "shop/base.html" %}
{% load static %}
```

```

{% block title %}
  {{ product.name }}
{% endblock %}

{% block content %}
<div class="product-detail">
  
  <h1>{{ product.name }}</h1>
  <h2>
    <a href="{{ product.category.get_absolute_url }}">
      {{ product.category }}
    </a>
  </h2>
  <p class="price">${{ product.price }}</p>
  {{ product.description|linebreaks }}
</div>
{% endblock %}

```

В приведенном выше исходном коде метод `get_absolute_url()` вызывается с объектом связанной категории, чтобы отобразить имеющиеся товары, принадлежащие к той же категории.

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и кликните по любому товару, чтобы увидеть страницу детальной информации о товаре. Она будет выглядеть следующим образом:

My shop Your cart is empty.



Red tea

Tea

\$45.50

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Рис. 8.7. Страница детальной информации о товаре

Вы создали базовый каталог товаров. Далее вы реализуете корзину покупок, которая позволит пользователям добавлять в нее любые товары во время просмотра интернет-магазина.

Разработка корзины покупок

После разработки каталога товаров следующим шагом будет создание корзины покупок, чтобы пользователи могли выбирать товары, которые они хотят приобрести. Корзина покупок позволяет пользователям выбирать товары и устанавливать сумму, на которую они желают сделать заказ, а затем временно сохранять эту информацию, пока они просматривают сайт, до тех пор, пока они в конечном итоге не разместят заказ. Состояние корзины покупок должно храниться в сеансе, чтобы товарные позиции корзины оставались в ней во время посещения магазина пользователем.

Для хранения состояния корзины будет использоваться встроенный в Django фреймворк сеансов. Корзина будет оставаться в сеансе до тех пор, пока сеанс не завершится либо пользователь не оформит заказ. Также нужно будет создать дополнительные модели Django для корзины и ее товарных позиций.

Использование сеансов Django

Django предоставляет фреймворк сеансов, который поддерживает анонимные и пользовательские сеансы. Фреймворк сеансов позволяет хранить произвольные данные каждого посетителя. Сеансовые данные хранятся на стороне сервера, а cookie-файлы содержат ИД сеанса, если только вы не используете cookie-ориентированный сеансовый механизм. Сеансовый промежуточный компонент управляет отправкой и получением cookie-файлов. Стандартный сеансовый механизм хранит сеансовые данные в базе данных, но существует возможность выбирать и другие сеансовые механизмы.

В целях использования сеансов необходимо, чтобы настроечный параметр MIDDLEWARE проекта содержал 'django.contrib.sessions.middleware.SessionMiddleware'. Данный промежуточный компонент управляет сеансами. Он добавляется в настроечный параметр MIDDLEWARE по умолчанию при создании нового проекта с помощью команды `startproject`.

Сеансовый промежуточный компонент делает текущий сеанс доступным в объекте `request`. К текущему сеансу можно обращаться, используя `request.session`, трактуя его как словарь Python для хранения и извлечения сеансовых данных. Словарь `session` по умолчанию принимает любой объект Python, который можно сериализовать в JSON. Значение переменной в сеансе устанавливается следующим образом:

```
request.session['foo'] = 'bar'
```

Извлечение сеансового ключа выполняется так:

```
request.session.get('foo')
```

Удаление ключа, который ранее был сохранен в сеансе, выполняется следующим образом:

```
del request.session['foo']
```



При входе пользователей на сайт их анонимный сеанс теряется, и создается новый сеанс для аутентифицированных пользователей. Если в анонимном сеансе хранятся элементы, которые необходимо сохранить после входа пользователя в систему, то придется копировать данные старого сеанса в новый сеанс. Это делается путем извлечения сеансовых данных перед входом пользователя в систему с помощью функции `login()`, встроенной в Django системы аутентификации, и сохранения их в сеансе после этого.

Настроечные параметры сеанса

С целью конфигурирования сеансов для своего проекта можно использовать несколько настроечных параметров. Наиболее важным является `SESSION_ENGINE`. Этот параметр позволяет указывать место, где будут храниться сеансы. По умолчанию Django хранит сеансы в базе данных, используя модель `Session` приложения `django.contrib.sessions`.

Django предлагает следующие варианты хранения сеансовых данных:

- **сеансы на основе базы данных:** сеансовые данные хранятся в базе данных. Этот сеансовый механизм применяется по умолчанию;
- **сеансы на основе файлов:** сеансовые данные хранятся в файловой системе;
- **сеансы на основе кеша:** сеансовые данные хранятся в кеш-памяти. Кеширующие бэкенды указываются в настроечном параметре `CACHES`. Хранение сеансовых данных в системе кеширования обеспечивает наилучшую производительность;
- **кешированные сеансы на основе базы данных:** сеансовые данные хранятся в кеше со сквозной записью¹ и в базе данных. Операции чтения используют базу данных только в том случае, если данные еще не находятся в кеше;
- **сеансы на основе cookie-файлов:** сеансовые данные хранятся в cookie-файлах, которые отправляются в браузер.

¹ Сквозная запись (write-through) – это метод хранения, при котором данные одновременно записываются в кеш и в соответствующую ячейку основной памяти. Кешированные данные позволяют быстро извлекать их по требованию, тогда как те же данные в основной памяти гарантируют, что ничего не будет потеряно в случае сбоя. – *Прим. перев.*



В целях повышения производительности следует использовать сеансовый механизм на основе кеша. Django поддерживает кеш-бэкенд Memcached прямо «из коробки», при этом еще имеются сторонние механизмы кеширования для Redis и других систем кеширования.

Сеансы можно адаптировать под конкретно-прикладную задачу с использованием конкретных настроечных параметров. Вот несколько важных настроечных параметров, связанных с сеансом:

- `SESSION_COOKIE_AGE`: продолжительность сеансовых cookie-файлов в секундах. По умолчанию равна 1 209 600 (две недели);
- `SESSION_COOKIE_DOMAIN`: домен, используемый для сеансовых cookie-файлов. Установите его значение равным `mydomain.com`, чтобы активировать междоменные cookie-файлы, или задайте `None` для стандартного доменного cookie-файла;
- `SESSION_COOKIE_HTTPONLY`: булево значение, указывающее на использование/неиспользование флага `HttpOnly` для сеансового cookie-файла. Если для этого параметра установлено значение `True`, то клиентский JavaScript не сможет получать доступ к сеансовому cookie-файлу. По умолчанию равно `True` с целью повышения защиты от перехвата пользовательского сеанса;
- `SESSION_COOKIE_SECURE`: булево значение, указывающее, что cookie-файл следует отправлять только в том случае, если соединение работает по протоколу HTTPS. По умолчанию равно `False`;
- `SESSION_EXPIRE_AT_BROWSER_CLOSE`: булево значение, указывающее, что сеанс должен истечь при закрытии браузера. По умолчанию равно `False`;
- `SESSION_SAVE_EVERY_REQUEST`: булево значение, которое, если оно равно `True`, будет сохранять сеанс в базе данных при каждом запросе. Срок истечения сеанса также обновляется при каждом его сохранении. По умолчанию равно `False`.

Список всех сеансовых настроечных параметров и их заданные по умолчанию значения находятся по адресу <https://docs.djangoproject.com/en/4.1/ref/settings/#sessions>.

Срок истечения сеанса

Используя настроечный параметр `SESSION_EXPIRE_AT_BROWSER_CLOSE`, можно применять сеансы браузерной продолжительности либо постоянные (персистентные) сеансы. По умолчанию значение этого параметра установлено равным `False`, что принудительно устанавливает продолжительность сеанса равной значению, хранящемуся в настроечном параметре `SESSION_COOKIE_AGE`. Если значение параметра `SESSION_EXPIRE_AT_BROWSER_CLOSE` установлено равным `True`, то сеанс будет истекать, когда пользователь закроет браузер, и параметр `SESSION_COOKIE_AGE` не будет иметь никакого эффекта.

Для перезаписи продолжительности текущего сеанса можно воспользоваться методом `set_expiry()` объекта `request.session`.

Хранение корзин покупок в сеансах

Теперь нужно создать простую структуру, которую можно сериализовывать в JSON, чтобы хранить товарные позиции корзины в сеансе.

Корзина должна содержать следующие данные по каждому содержащемуся в ней товару:

- ИД экземпляра класса `Product`;
- выбранное количество товара;
- цена за единицу товара.

Поскольку цены на товары могут варьироваться, давайте воспользуемся подходом, при котором цена товара будет сохраняться вместе с самим товаром при его добавлении в корзину. Поступая таким образом, в момент, когда пользователи добавляют товар в свою корзину, будет использоваться текущая цена товара, независимо от того, изменится цена товара впоследствии или нет. Это означает, что цена, которую товар имеет, когда клиент добавляет его в корзину, остается для этого клиента в сеансе до завершения оформления заказа либо завершения сеанса.

Далее необходимо создать функциональность создания корзин покупок и связывания их с сеансами. Она должна работать следующим образом:

- в случае когда нужна корзина, проверить, что конкретно-прикладной сеансовый ключ установлен. Если в сеансе корзина не установлена, то создается новая корзина и сохраняется в сеансовом ключе корзины;
- в случае поочередных запросов выполнить ту же проверку и извлечь товарные позиции корзины из сеансового ключа корзины. При этом товарные позиции корзины извлекаются из сеанса, а связанные с ними объекты `Product` – из базы данных.

Отредактируйте файл `settings.py` проекта, добавив следующий ниже настроечный параметр:

```
CART_SESSION_ID = 'cart'
```

Это ключ, который будет использоваться для хранения корзины в пользовательском сеансе. Поскольку сеансы Django управляются по каждому посетителю, для всех сеансов можно использовать один и тот же сеансовый ключ корзины.

Давайте создадим приложение для управления корзинами покупок. Откройте терминал и создайте новое приложение, выполнив следующую ниже команду из каталога проекта:

```
python manage.py startapp cart
```

Затем отредактируйте файл `settings.py` проекта, добавив новое приложение в настроечный параметр `INSTALLED_APPS`, используя следующую ниже строку, выделенную жирным шрифтом:

```
INSTALLED_APPS = [  
    # ...  
    'shop.apps.ShopConfig',  
    'cart.apps.CartConfig',  
]
```

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

```
from decimal import Decimal  
from django.conf import settings  
from shop.models import Product  
  
class Cart:  
    def __init__(self, request):  
        """  
        Инициализировать корзину.  
        """  
        self.session = request.session  
        cart = self.session.get(settings.CART_SESSION_ID)  
        if not cart:  
            # сохранить пустую корзину в сеансе  
            cart = self.session[settings.CART_SESSION_ID] = {}  
        self.cart = cart
```

Это класс `Cart`, который позволит управлять корзиной покупок. Переменная `cart` должна быть инициализирована объектом `request`. Текущий сеанс сохраняется посредством инструкции `self.session = request.session`, чтобы сделать его доступным для других методов класса `Cart`.

Сначала с помощью метода `self.session.get(settings.CART_SESSION_ID)` делается попытка получить корзину из текущего сеанса. Если в сеансе корзины нет, то путем задания пустого словаря в сеансе создается пустая корзина.

Далее надо сформировать словарь `cart` с идентификаторами товаров в качестве ключей и словарем, который будет содержать количество и цену, по каждому ключу товара. Поступая таким образом, можно гарантировать, что товар не будет добавляться в корзину более одного раза. Благодаря этому также упрощается извлечение товаров из корзины.

Давайте создадим метод добавления товаров в корзину или обновления их количеств. Добавьте следующие ниже методы `add()` и `save()` в класс `Cart`:

```
class Cart:  
    # ...  
    def add(self, product, quantity=1, override_quantity=False):
```

```

"""
Добавить товар в корзину либо обновить его количество.
"""
product_id = str(product.id)
if product_id not in self.cart:
    self.cart[product_id] = {'quantity': 0,
                             'price': str(product.price)}

if override_quantity:
    self.cart[product_id]['quantity'] = quantity
else:
    self.cart[product_id]['quantity'] += quantity
self.save()

def save(self):
    # пометить сеанс как "измененный",
    # чтобы обеспечить его сохранение
    self.session.modified = True

```

Метод `add()` принимает на входе следующие ниже параметры:

- `product`: экземпляр `product` для его добавления в корзину либо его обновления;
- `quantity`: опциональное целое число с количеством товара. По умолчанию равен 1;
- `override_quantity`: это булево значение, указывающее, нужно ли заменить количество переданным количеством (`True`) либо прибавить новое количество к существующему количеству (`False`).

ИД товара используется в качестве ключа в словаре содержимого корзины. ИД товара конвертируется в строковое значение, потому что Django использует JSON для сериализации сеансовых данных, а JSON допускает только строковые имена ключей. ИД товара является ключом, а сохраняемое значение – словарем с числами количества и цены товара. Цена товара конвертируется из десятичного числа фиксированной точности в строковое значение для его сериализации. Наконец, вызывается метод `save()`, чтобы сохранить корзину в сеансе.

Метод `save()` помечает сеанс как измененный, используя `session.modified = True`. Это сообщает Django о том, что сеанс изменился и его необходимо сохранить.

Также потребуется метод удаления товаров из корзины. Добавьте следующий ниже метод в класс `Cart`:

```

class Cart:
    # ...
    def remove(self, product):
        """

```

```
    """  
    Удалить товар из корзины.  
    """  
    product_id = str(product.id)  
    if product_id in self.cart:  
        del self.cart[product_id]  
    self.save()
```

Метод `remove()` удаляет данный товар из словаря `cart` и вызывает метод `save()`, чтобы обновить корзину в сеансе.

Кроме того, нужно будет прокручивать в цикле содержащиеся в корзине товарные позиции и обращаться к соответствующим экземплярам класса `Product`. Для этого в своем классе можно определить метод `__iter__()`. Добавьте следующий ниже метод в класс `Cart`:

```
class Cart:  
    # ...  
    def __iter__(self):  
        """  
        Прокрутить товарные позиции корзины в цикле и  
        получить товары из базы данных.  
        """  
        product_ids = self.cart.keys()  
        # получить объекты product и добавить их в корзину  
        products = Product.objects.filter(id__in=product_ids)  
        cart = self.cart.copy()  
        for product in products:  
            cart[str(product.id)]['product'] = product  
        for item in cart.values():  
            item['price'] = Decimal(item['price'])  
            item['total_price'] = item['price'] * item['quantity']  
        yield item
```

В методе `__iter__()` извлекаются присутствующие в корзине экземпляры класса `Product`, чтобы включить их в товарные позиции корзины. Текущая корзина копируется в переменную `cart`, и в нее добавляются экземпляры класса `Product`. Наконец, товары корзины прокручиваются в цикле, конвертируя цену каждого товара обратно в десятичное число фиксированной точности и добавляя в каждый товар атрибут `total_price`. Метод `__iter__()` позволит легко прокручивать товарные позиции корзины в представлениях и шаблонах.

Кроме того, понадобится способ возвращать общее число товаров в корзине. Когда функция `len()` выполняется с объектом в качестве аргумента, Python вызывает свой метод `__len__()` для получения его длины. Далее будет определен конкретно-прикладной метод `__len__()`, чтобы возвращать общее число товаров, хранящихся в корзине.

Добавьте следующий ниже метод `__len__()` в класс `Cart`:

```
class Cart:
    # ...
    def __len__(self):
        """
        Подсчитать все товарные позиции в корзине.
        """
        return sum(item['quantity'] for item in self.cart.values())
```

В результате будет возвращаться сумма количеств всех товаров в корзине. Добавьте следующий ниже метод расчета общей стоимости товаров в корзине:

```
class Cart:
    # ...
    def get_total_price(self):
        return sum(Decimal(item['price']) * item['quantity']
                   for item in self.cart.values())
```

Наконец, добавьте метод очистки сеанса корзины:

```
class Cart:
    # ...
    def clear(self):
        # удалить корзину из сеанса
        del self.session[settings.CART_SESSION_ID]
        self.save()
```

Теперь класс `Cart` готов к управлению корзинами покупок.

Создание представлений корзины покупок

Теперь, когда у нас имеется класс `Cart` для управления корзиной, необходимо создать представления, чтобы добавлять, обновлять или удалять товары из корзины. Нужно создать следующие ниже представления:

- представление добавления или обновления товаров в корзине, которое может обрабатывать текущие и новые количества;
- представление удаления товаров из корзины;
- представление отображения товарных позиций корзины и итоговых величин.

Добавление товаров в корзину

Для того чтобы добавить товары в корзину, нужна форма, позволяющая пользователю выбирать количество. Внутри каталога приложения `cart` создайте файл `forms.py`, добавив следующий ниже исходный код:

```
from django import forms

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
        choices=PRODUCT_QUANTITY_CHOICES,
        coerce=int)
    override = forms.BooleanField(required=False,
        initial=False,
        widget=forms.HiddenInput)
```

Эта форма будет использоваться для добавления товаров в корзину. Ваш класс `CartAddProductForm` содержит следующие два поля:

- `quantity`: позволяет пользователю выбирать количество от 1 до 20. Для конвертирования входных данных в целое число используется поле `TypedChoiceField` вместе с `coerce=int`;
- `override`: позволяет указывать, должно ли количество быть прибавлено к любому существующему количеству в корзине для этого товара (`False`) или же существующее количество должно быть переопределено данным количеством (`True`). Для этого поля используется виджет `HiddenInput`, так как это поле не будет показываться пользователю.

Давайте создадим представление добавления товаров в корзину. Отредактируйте файл `views.py` приложения `cart`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from shop.models import Product
from .cart import Cart
from .forms import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product,
```

```

        quantity=cd['quantity'],
        override_quantity=cd['override'])
    return redirect('cart:cart_detail')

```

Это представление добавления товаров в корзину или обновления количества существующих товаров. В нем используется декоратор `require_POST`, чтобы разрешать запросы только методом `POST`. Указанное представление получает ИД товара в качестве параметра. Затем извлекается экземпляр класса `Product` с заданным ИД и выполняется валидация формы посредством `CartAddProductForm`. Если форма валидна, то товар в корзине либо добавляется, либо обновляется. Представление перенаправляет на URL-адрес `cart_detail`, который будет отображать содержимое корзины. Вы создадите представление `cart_detail` чуть позже.

Также потребуется представление удаления товаров из корзины. Добавьте следующий ниже исходный код в файл `views.py` приложения `cart`:

```

@require_POST
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')

```

Представление `cart_remove` получает ИД товара в качестве параметра. В нем используется декоратор `require_POST`, чтобы разрешать запросы только методом `POST`. Экземпляр товара извлекается с заданным ИД, и товар удаляется из корзины. Затем пользователь перенаправляется на URL-адрес `cart_detail`.

Наконец, потребуется представление отображения корзины и ее товаров. Добавьте следующее ниже представление в файл `views.py` приложения `cart`:

```

def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})

```

Представление `cart_detail` получает текущую корзину, чтобы ее отобразить.

Вы создали представления добавления товаров в корзину, обновления количества, удаления товаров из корзины и отображения содержимого корзины. Давайте добавим шаблоны URL-адресов этих представлений. Внутри каталога приложения `cart` создайте новый файл и назовите его `urls.py`. Добавьте в него следующие ниже URL-адреса:

```

from django.urls import path
from . import views

app_name = 'cart'

```

```
urlpatterns = [  
    path('', views.cart_detail, name='cart_detail'),  
    path('add/<int:product_id>/', views.cart_add, name='cart_add'),  
    path('remove/<int:product_id>/', views.cart_remove,  
        name='cart_remove'),  
]
```

Отредактируйте главный файл `urls.py` проекта `myshop`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом, чтобы включить URL-адреса корзины:

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('cart/', include('cart.urls', namespace='cart')),  
    path('', include('shop.urls', namespace='shop')),  
]
```

Проверьте, чтобы этот шаблон URL-адреса был вставлен перед шаблоном `shop.urls`, так как он более строгий, чем последний.

Разработка шаблона отображения корзины

Представления `cart_add` и `cart_remove` не будут прорисовывать никаких шаблонов, но вам нужно создать шаблон для представления `cart_detail`, чтобы отображались товарные позиции корзины и итоговые величины.

Внутри каталога приложения `cart` создайте следующую ниже файловую структуру:

```
templates/  
  cart/  
    detail.html
```

Отредактируйте шаблон `cart/detail.html`, добавив приведенный ниже исходный код:

```
{% extends "shop/base.html" %}  
{% load static %}  
  
{% block title %}  
  Your shopping cart  
{% endblock %}  
  
{% block content %}  
  <h1>Your shopping cart</h1>  
  <table class="cart">  
    <thead>
```

```

<tr>
  <th>Image</th>
  <th>Product</th>
  <th>Quantity</th>
  <th>Remove</th>
  <th>Unit price</th>
  <th>Price</th>
</tr>
</thead>
<tbody>
  {% for item in cart %}
    {% with product=item.product %}
      <tr>
        <td>
          <a href="{{ product.get_absolute_url }}">
            
          </a>
        </td>
        <td>{{ product.name }}</td>
        <td>{{ item.quantity }}</td>
        <td>
          <form action="{% url "cart:cart_remove" product.id %}"
method="post">
            <input type="submit" value="Remove">
              {% csrf_token %}
            </form>
        </td>
        <td class="num">${{ item.price }}</td>
        <td class="num">${{ item.total_price }}</td>
      </tr>
    {% endwhile %}
  {% endfor %}
  <tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price }}</td>
  </tr>
</tbody>
</table>
<p class="text-right">
  <a href="{% url "shop:product_list" %}" class="button
light">Continue shopping</a>
  <a href="#" class="button">Checkout</a>
</p>
{% endblock %}

```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон, который используется для отображения содержимого корзины. Он содержит таблицу с товарами, хранящимися в текущей корзине. Пользователи имеют возможность изменять количество выбранных товаров, используя форму, которая отправляется методом POST в представление cart_add. Кроме того, пользователи могут удалять товары из корзины с помощью кнопки **Delete** (Удалить), предоставляемой для каждого из них. Наконец, используется HTML-форма с атрибутом action, которая указывает на URL-адрес cart_remove, включая ИД товара.

Добавление товаров в корзину

Теперь необходимо добавить кнопку **Add to cart** (Добавить в корзину) на страницу детальной информации о товаре. Отредактируйте файл views.py приложения shop, добавив CartAddProductForm в представление product_detail, как показано ниже:

```
from cart.forms import CartAddProductForm

# ...

def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id,
                                   slug=slug,
                                   available=True)

    cart_product_form = CartAddProductForm()
    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                  'cart_product_form': cart_product_form})
```

Отредактируйте шаблон shop/product/detail.html приложения shop, добавив следующую ниже форму к цене товара, как показано ниже. Новые строки выделены жирным шрифтом:

```
...
<p class="price">${{ product.price }}</p>
<form action="{% url 'cart:cart_add' product.id %}" method="post">
    {{ cart_product_form }}
    {% csrf_token %}
    <input type="submit" value="Add to cart">
</form>
{{ product.description|linebreaks }}
...
```

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Теперь откройте адрес `http://127.0.0.1:8000/` в своем браузере и пройдите на страницу детальной информации о товаре. Она будет содержать форму для выбора количества перед добавлением товара в корзину. Страница будет выглядеть так:

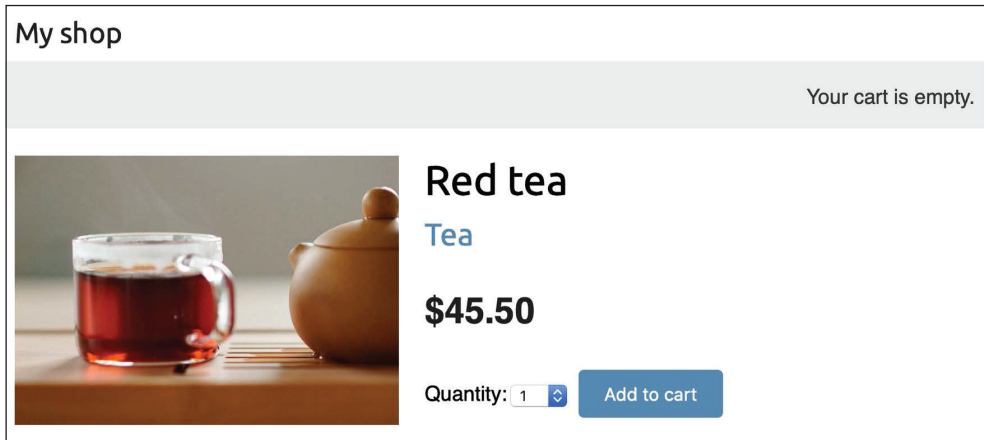


Рис. 8.8. Страница детальной информации о товаре, включающая форму для добавления товара в корзину

Выберите количество и кликните по кнопке **Add to cart** (Добавить в корзину). Форма передается на обработку в представление `cart_add` методом `POST`. Представление добавляет товар в корзину в сеансе, включая его текущую цену и выбранное количество. Затем оно перенаправляет пользователя на страницу детальной информации о корзине, которая будет выглядеть, как на рис. 8.9.

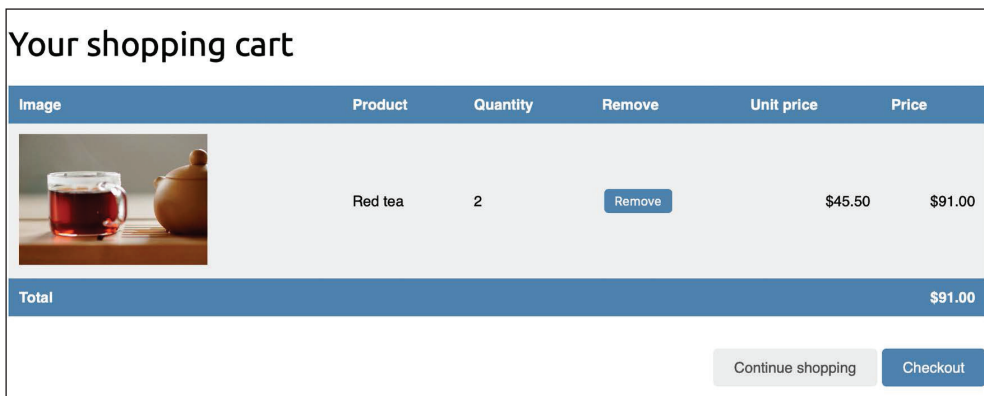


Рис. 8.9. Страница детальной информации о корзине

Обновление количества товаров в корзине

Когда пользователи увидят корзину, они, возможно, захотят изменить количество товаров перед размещением заказа. В целях обеспечения этой функциональности необходимо разрешить пользователям изменять количество на странице детальной информации о корзине.

Отредактируйте файл `views.py` приложения `cart`, добавив следующие ниже строки, выделенные жирным шрифтом, в представление `cart_detail`:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(initial={
            'quantity': item['quantity'],
            'override': True})
    return render(request, 'cart/detail.html', {'cart': cart})
```

По каждому товару в корзине создается экземпляр `CartAddProductForm`, чтобы разрешить изменение количества товара. Форма инициализируется текущим количеством товара, и поле `override` получает значение `True`, чтобы при передаче формы на обработку в представление `cart_add` текущее количество заменялось новым.

Теперь отредактируйте шаблон `cart/detail.html` приложения `cart` и найдите следующую ниже строку:

```
<td>{{ item.quantity }}</td>
```

Замените указанную строку таким исходным кодом:

```
<td>
  <form action="{% url 'cart:cart_add' product.id %}" method="post">
    {{ item.update_quantity_form.quantity }}
    {{ item.update_quantity_form.override }}
    <input type="submit" value="Update">
    {% csrf_token %}
  </form>
</td>
```

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/cart/` в своем браузере.

Вы увидите форму для редактирования количества по каждой товарной позиции корзины, как показано ниже:

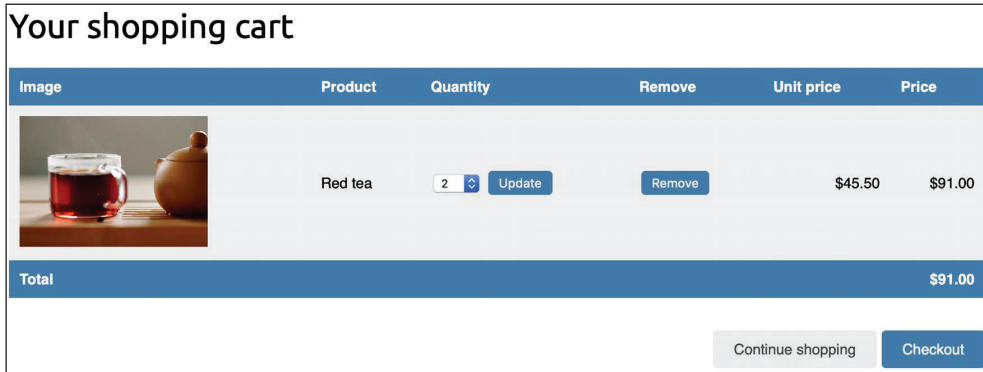


Рис. 8.10. Страница детальной информации о корзине, включающая форму для обновления количества товаров

Измените количество товара и кликните по кнопке **Update** (Обновить), чтобы протестировать новую функциональность. Вы также можете удалить товар из корзины, кликнув по кнопке **Remove** (Удалить).

Создание процессора контекста для текущей корзины

Возможно, вы заметили, что сообщение **Your cart is empty** (Ваша корзина пуста) отображается в шапке сайта, даже если в корзине есть товары. Вместо этого вы должны отображать общее количество товаров в корзине и общую стоимость. Поскольку эта информация должна отображаться на всех страницах, необходимо создать процессор контекста, чтобы включать текущую корзину в контекст запроса, независимо от представления, которым обрабатывается запрос.

Процессоры контекста

Процессор контекста – это функция Python, которая принимает объект `request` в качестве аргумента и возвращает словарь, который добавляется в контекст запроса. Процессоры контекста оказываются как нельзя кстати, когда нужно сделать что-то глобально доступным для всех шаблонов.

По умолчанию при создании нового проекта с помощью команды `start-project` проект содержит следующие ниже процессоры контекста шаблона в параметре `context_processors` внутри настроечного параметра `TEMPLATES`:

- `django.template.context_processors.debug`: устанавливает булевы переменные `debug` и `sql_queries` в контексте, представляющие список SQL-запросов, исполняемых в запросе;
- `django.template.context_processors.request`: устанавливает переменную `request` в контексте;

- `django.contrib.auth.context_processors.auth`: устанавливает переменную `user` в запросе;
- `django.contrib.messages.context_processors.messages`: устанавливает переменную `messages` в контексте, содержащую все сообщения, сгенерированные с использованием фреймворка сообщений.

Django также активирует `django.template.context_processors.csrf`, чтобы избежать атак с подделкой межсайтовых запросов (**CSRF**). Этого процессора контекста нет в настройечном параметре, но он всегда активирован, и его невозможно деактивировать из соображений безопасности.

Список всех встроенных процессоров контекста находится по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/api/#built-in-template-context-processors>.

Установка корзины в контекст запроса

Давайте создадим процессор контекста, чтобы установить текущую корзину в контекст запроса. С помощью него можно получать доступ к корзине в любом шаблоне.

Внутри каталога приложения `cart` создайте новый файл и назовите его `context_processors.py`. Процессоры контекста могут располагаться в вашем исходном коде где угодно, но размещение их здесь будет поддерживать ваш код в хорошо организованном состоянии. Добавьте в файл следующий ниже исходный код:

```
from .cart import Cart

def cart(request):
    return {'cart': Cart(request)}
```

Здесь в процессоре контекста создается экземпляр класса `Cart` с объектом `request` в качестве параметра и обеспечивается его доступность для шаблонов в виде переменной `cart`.

Отредактируйте файл `settings.py` проекта, добавив `cart.context_processors.cart` в опцию `context_processors` внутри настройечного параметра `TEMPLATES`, как показано ниже. Новая строка выделена жирным шрифтом:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
```

```

        'django.contrib.messages.context_processors.messages',
        'cart.context_processors.cart',
    ],
},
]

```

Процессор контекста переменной `cart` будет исполняться всякий раз, когда шаблон прорисовывается с использованием встроенного в Django контекстного класса `RequestContext`. Переменная `cart` будет устанавливаться в контекст ваших шаблонов. Подробнее о классе `RequestContext` можно почитать по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/api/#django.template.RequestContext>.



Процессоры контекста исполняются во всех запросах, в которых используется контекстный класс `RequestContext`. В случае если ваша функциональность не требуется во всех шаблонах, в особенности если она предусматривает запросы к базе данных, возможно, вместо процессора контекста вы захотите создать конкретно-прикладной шаблонный тег.

Затем откройте шаблон `shop/base.html` приложения `shop` и найдите следующие ниже строки:

```

<div class="cart">
    Your cart is empty.
</div>

```

Замените эти строки таким исходным кодом:

```

<div class="cart">
    {% with total_items=cart|length %}
    {% if total_items > 0 %}
        Your cart:
        <a href="{% url 'cart:cart_detail' %}">
            {{ total_items }} item{{ total_items|pluralize }},
            ${{ cart.get_total_price }}
        </a>
    {% else %}
        Your cart is empty.
    {% endif %}
    {% endwith %}
</div>

```

С помощью следующей команды перезапустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и добавьте несколько товаров в корзину.

Теперь в шапке сайта можно увидеть общее количество товаров в корзине и их общую стоимость, как показано ниже:

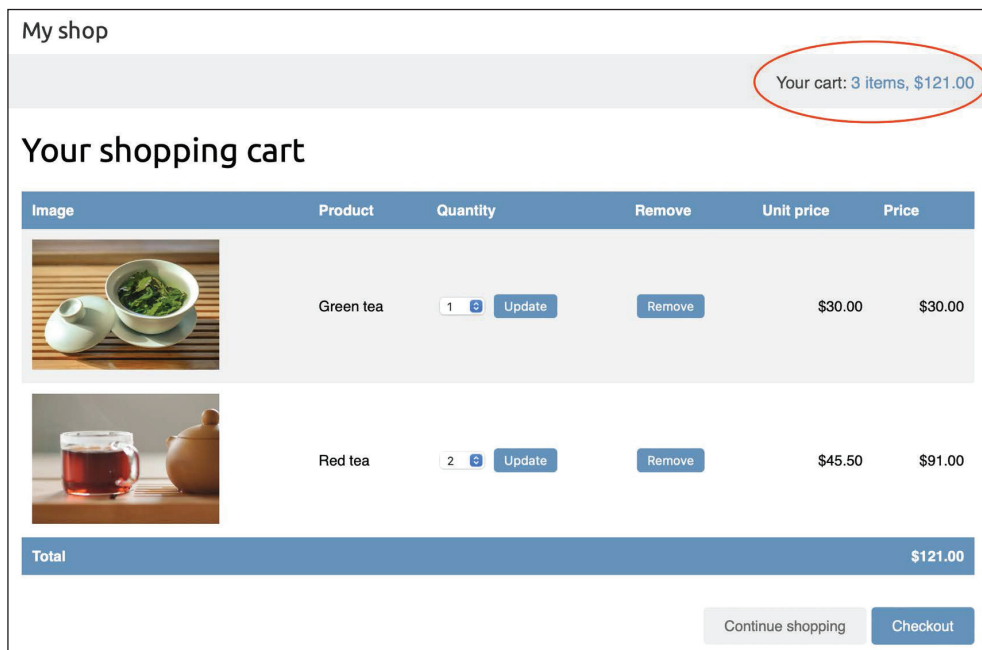




Image	Product	Quantity	Remove	Unit price	Price
	Green tea	1 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$30.00	\$30.00
	Red tea	2 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$45.50	\$91.00
Total					\$121.00

Рис. 8.11. Шапка сайта, отображающая текущие товары в корзине

Вы завершили работу над корзиной. Далее вы создадите функциональность по регистрации заказов клиентов.

Регистрация заказов клиентов

После оформления заказа необходимо сохранить заказ в базе данных. Заказы будут содержать информацию о клиентах и товарах, которые они покупают.

Следующей ниже командой создайте новое приложение для управления заказами клиентов:

```
python manage.py startapp orders
```

Отредактируйте файл `settings.py` проекта, добавив новое приложение в параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [  
    # ...  
    'shop.apps.ShopConfig',  
    'cart.apps.CartConfig',  
    'orders.apps.OrdersConfig',  
]
```

Вы активировали приложение `orders`.

Создание моделей заказа

Одна модель понадобится для хранения детальной информации о заказе и вторая – для хранения приобретенных товаров, включая их цену и количество. Отредактируйте файл `models.py` приложения `orders`, добавив следующий ниже исходный код:

```
from django.db import models  
from shop.models import Product  
  
class Order(models.Model):  
    first_name = models.CharField(max_length=50)  
    last_name = models.CharField(max_length=50)  
    email = models.EmailField()  
    address = models.CharField(max_length=250)  
    postal_code = models.CharField(max_length=20)  
    city = models.CharField(max_length=100)  
    created = models.DateTimeField(auto_now_add=True)  
    updated = models.DateTimeField(auto_now=True)  
    paid = models.BooleanField(default=False)  
  
    class Meta:  
        ordering = ['-created']  
        indexes = [  
            models.Index(fields=['-created']),  
        ]  
  
    def __str__(self):  
        return f'Order {self.id}'  
  
    def get_total_cost(self):  
        return sum(item.get_cost() for item in self.items.all())
```

```
class OrderItem(models.Model):
    order = models.ForeignKey(Order,
                              related_name='items',
                              on_delete=models.CASCADE)
    product = models.ForeignKey(Product,
                                 related_name='order_items',
                                 on_delete=models.CASCADE)
    price = models.DecimalField(max_digits=10,
                                decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return str(self.id)

    def get_cost(self):
        return self.price * self.quantity
```

Модель `Order` содержит несколько полей для хранения информации о клиенте и булево поле `paid`, которое по умолчанию имеет значение `False`. Позже это поле будет использоваться для того, чтобы различать оплаченные и неоплаченные заказы. Здесь также определен метод `get_total_cost()`, который получает общую стоимость товаров, приобретенных в этом заказе.

Модель `OrderItem` позволяет хранить товар, количество и цену, уплаченную за каждый товар. Здесь определен метод `get_cost()`, который возвращает стоимость товара путем умножения цены товара на количество.

Выполните следующую ниже команду, чтобы создать начальные миграции для приложения `orders`:

```
python manage.py makemigrations
```

Вы увидите примерно такой результат:

```
Migrations for 'orders':
  orders/migrations/0001_initial.py
    - Create model Order
    - Create model OrderItem
    - Create index orders_orde_created_743fca_idx on field(s) -created of model
  order
```

Выполните следующую ниже команду, чтобы применить новую миграцию:

```
python manage.py migrate
```

Вы увидите следующий ниже результат:

```
Applying orders.0001_initial... OK
```

Теперь модели заказов синхронизированы с базой данных.

Включение моделей заказа на сайт администрирования

Давайте добавим модели заказа на сайт администрирования. Отредактируйте файл `admin.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                  'address', 'postal_code', 'city', 'paid',
                  'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
```

Класс `ModelInline` используется с моделью `OrderItem`, чтобы включать ее *внутристрочно*¹ в класс `OrderAdmin`. Атрибут `inlines` позволяет вставлять модель в ту же страницу редактирования, что и связанная с ней модель.

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/add/` в своем браузере. Вы увидите такую страницу:

¹ Англ. `inline`; син. локально. – Прим. перев.

Add order

First name:

Last name:

Email:

Address:

Postal code:

City:

Paid

ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
<input type="text"/> Q	<input type="text"/>	1	✖
<input type="text"/> Q	<input type="text"/>	1	✖
<input type="text"/> Q	<input type="text"/>	1	✖

[+ Add another Order item](#)

Рис. 8.12. Форма для добавления заказа, включающая `OrderItemInline`

Создание заказов клиентов

Созданные ранее модели заказа будут использоваться для вывода товарных позиций корзины в постоянное хранилище, когда пользователь наконец разместит заказ. Новый заказ будет создан после следующих ниже шагов:

- 1) предоставить пользователю форму заказа, чтобы тот заполнил ее своими данными;
- 2) создать новый экземпляр `Order` с введенными данными и создать связанный экземпляр `OrderItem` для каждого товара в корзине;
- 3) очистить все содержимое корзины и перенаправить пользователя на страницу успеха.

Сперва потребуется форма для ввода детальной информации о заказе. Внутри каталога приложения `orders` создайте новый файл и назовите его `forms.py`. Добавьте в него следующий ниже исходный код:

```
from django import forms
from .models import Order
```

```
class OrderCreateForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                 'postal_code', 'city']
```

Это форма, которая будет использоваться для создания новых объектов `Order`. Теперь потребуется представление обработки формы и создания нового заказа. Отредактируйте файл `views.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.shortcuts import render
from .models import OrderItem
from .forms import OrderCreateForm
from cart.cart import Cart

def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                                         product=item['product'],
                                         price=item['price'],
                                         quantity=item['quantity'])

            # очистить корзину
            cart.clear()
            return render(request,
                         'orders/order/created.html',
                         {'order': order})
    else:
        form = OrderCreateForm()
        return render(request,
                     'orders/order/create.html',
                     {'cart': cart, 'form': form})
```

В представлении `order_create` текущая корзина извлекается из сеанса посредством инструкции `cart = Cart(request)`.

В зависимости от метода запроса выполняется следующая работа:

- **запрос методом GET:** создает экземпляр формы `OrderCreateForm` и присовывает шаблон `orders/order/create.html`;
- **запрос методом POST:** выполняет валидацию отправленных в запросе данных. Если данные валидны, то в базе данных создается новый заказ, используя инструкцию `order = form.save()`. Товарные позиции корзины

прокручиваются в цикле, и для каждой из них создается `OrderItem`. Наконец, содержимое корзины очищается, и шаблон `orders/order/created.html` прорисовывается.

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

```
from django.urls import path
from . import views

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
]
```

Это шаблон URL-адреса представления `order_create`.

Отредактируйте файл `urls.py` проекта `myshop` и вставьте следующий ниже шаблон. Не забудьте поместить его перед шаблоном `shop.urls`, как показано далее. Новая строка выделена жирным шрифтом:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('', include('shop.urls', namespace='shop')),
]
```

Откройте шаблон `cart/detail.html` приложения `cart` и найдите вот эту строку:

```
<a href="#" class="button">Checkout</a>
```

Добавьте URL-адрес `order_create` в HTML-атрибут `href`, как показано ниже:

```
<a href="{% url 'orders:order_create' %}" class="button">
    Checkout
</a>
```

Теперь пользователи могут переходить со страницы детальной информации о корзине в форму для заказа.

Остается определить шаблоны создания заказов. Внутри каталога приложения `orders` создайте следующую ниже файловую структуру:

```
templates/
  orders/
```

```
order/
  create.html
  created.html
```

Отредактируйте шаблон `orders/order/create.html`, добавив следующий ниже исходный код:

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

{% block title %}
  Checkout
{% endblock %}

{% block content %}
  <h1>Checkout</h1>
  <div class="order-info">
    <h3>Your order</h3>
    <ul>
      {% for item in cart %}
        <li>
          {{ item.quantity }}x {{ item.product.name }}
          <span>${{{ item.total_price }}}</span>
        </li>
      {% endfor %}
    </ul>
    <p>Total: ${{ cart.get_total_price }}</p>
  </div>
  <form method="post" class="order-form">
    {{ form.as_p }}
    <p><input type="submit" value="Place order"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

Этот шаблон отображает товарные позиции корзины, включая итоговые величины и форму для размещения заказа.

Отредактируйте шаблон `orders/order/created.html`, добавив следующий ниже исходный код:

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

{% block title %}
  Thank you
{% endblock %}
```

```
{% block content %}
  <h1>Thank you</h1>
  <p>Your order has been successfully completed. Your order number is
  <strong>{{ order.id }}</strong>.</p>
{% endblock %}
```

Это шаблон, который прорисовывается при успешном создании заказа.

Запустите сервер веб-разработки, чтобы загрузить новые файлы. Откройте адрес `http://127.0.0.1:8000/` в своем браузере, добавьте пару товаров в корзину и пройдите на страницу оформления заказа. Вы увидите следующую ниже форму:

My shop

Your cart: 4 items, \$166.50

Checkout

<p>First name:</p> <input type="text" value="Antonio"/>	<h3>Your order</h3> <ul style="list-style-type: none">• 3x Red tea \$136.50• 1x Green tea \$30.00 <p>Total: \$166.50</p>
Last name:	
<input type="text" value="Melé"/>	
Email:	
<input type="text" value="antonio.mele@zenxit.com"/>	
Address:	
<input type="text" value="1 Bank Street"/>	
Postal code:	
<input type="text" value="E14 4AD"/>	
City:	
<input type="text" value="London"/>	
<input type="button" value="Place order"/>	

Рис. 8.13. Страница создания заказа, включающая форму для оформления заказа и детальную информацию

Заполните форму валидными данными и кликните по кнопке **Place order** (Разместить заказ). Заказ будет создан, и вы увидите страницу успеха, как показано ниже:

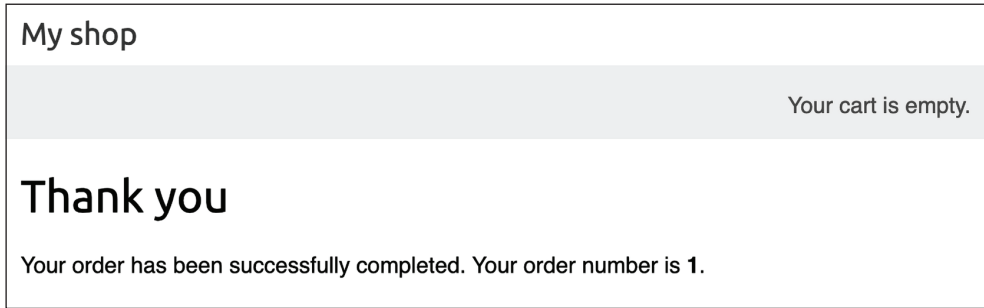


Рис. 8.14. Шаблон созданного заказа, отображающий номер заказа

Заказ зарегистрирован, корзина очищена.

Возможно, вы заметили, что сообщение **Your cart is empty** (Ваша корзина пуста) отображается в шапке после завершения заказа. Это вызвано тем, что корзина была очищена. Данное сообщение можно легко убрать в представлениях, в контексте шаблона которых есть объект `order`.

Отредактируйте шаблон `shop/base.html` приложения `shop`, заменив следующую ниже строку, выделенную жирным шрифтом:

```
...
<div class="cart">
  {% with total_items=cart|length %}
  {% if total_items > 0 %}
    Your cart:
    <a href="{% url 'cart:cart_detail' %}">
      {{ total_items }} item{{ total_items|pluralize }},
      ${{ cart.get_total_price }}
    </a>
    {% elif not order %}
    Your cart is empty.
  {% endif %}
  {% endwith %}
</div>
...
```

При создании заказа сообщение **Your cart is empty** больше отображаться не будет.

Теперь откройте сайт администрирования по адресу `http://127.0.0.1:8000/admin/orders/order/`. Вы увидите, что заказ был успешно создан, например:

Select order to change

Action: 0 of 1 selected

<input type="checkbox"/>	ID	FIRST NAME	LAST NAME	EMAIL	ADDRESS	POSTAL CODE	CITY	PAID	CREATED	UPDATED
<input type="checkbox"/>	1	Antonio	Melé	antonio.mele@zenixit.com	1 Bank Street	E14 4AD	London		Jan. 31, 2022, 5:46 p.m.	Jan. 31, 2022, 5:46 p.m.

1 order

Рис. 8.15. Раздел списка изменений заказов на сайте администрирования, включающий созданный заказ

Вы реализовали систему заказов. Далее вы научитесь создавать асинхронные задания, чтобы отправлять пользователям электронные письма о подтверждении размещенного ими заказа.

Асинхронные задания

При получении HTTP-запроса необходимо как можно быстрее вернуть ответ пользователю. Напомним, что в главе 7 «Отслеживание действий пользователя» вы использовали меню отладочных инструментов Django Debug Toolbar, чтобы проверять время различных фаз цикла запроса/ответа и время исполнения выполненных SQL-запросов. Каждое исполняемое в ходе цикла запроса/ответа задание прибавляет к общему времени ответа. Длительные задания могут серьезно замедлять ответ сервера. Как возвращать быстрый ответ пользователю, при этом выполняя времязатратные задания? Это можно делать с помощью асинхронного исполнения заданий.

Работа с асинхронными заданиями

Исполняя определенные задания в фоновом режиме, можно снимать нагрузку с цикла запрос/ответ. Например, платформа для видеообмена позволяет пользователям закачивать видео на платформу, но требует много времени для транскодирования закачанных видео. Когда пользователь закачивает видео, сайт может вернуть ответ, информирующий о том, что транскодирование скоро начнется, и начать транскодирование видео асинхронно. Еще один пример – отправка электронных писем пользователям. Если ваш сайт отправляет уведомления по электронной почте из представления, то соединение по простому протоколу передачи почты (SMTP) может завершиться ошибкой либо замедлить ответ. Отправляя электронное письмо асинхронно, избегается блокировка исполнения исходного кода.

Асинхронное исполнение доказало свою актуальность особенно в отношении процессов, интенсивных по объему обработки данных, используемых ресурсов и времени, или процессов, подверженных сбоям, для которых может потребоваться политика повторных попыток соединения.

Работники, очереди сообщений и брокеры сообщений

В то время как ваш веб-сервер обрабатывает запросы и возвращает ответы, вам нужен второй, основанный на заданиях сервер под названием **работник**¹, чтобы обрабатывать асинхронные задания. Один или несколько работников могут работать и исполнять задания в фоновом режиме. Эти работники могут обращаться к базе данных, обрабатывать файлы, отправлять электронные письма и т. д. Работники могут даже ставить будущие задания в очередь. При этом главный веб-сервер будет оставаться свободным для обработки HTTP-запросов.

Для того чтобы сообщать работникам о подлежащих исполнению заданиях, нужно отправлять **сообщения**. Связь осуществляется через брокеров путем добавления сообщения в **очередь сообщений**, которая в сущности представляет собой структуру данных, организованную по принципу «первым вошел – первым вышел» (**FIFO**). Когда брокер становится доступным, он берет первое сообщение из очереди и запускает исполнение соответствующего задания. По завершении брокер берет следующее сообщение из очереди и отправляет соответствующее задание на исполнение. Брокеры простаивают, когда очередь сообщений пуста. При использовании нескольких брокеров каждый брокер принимает первое доступное сообщение в том порядке, в котором они становятся доступными. Очередь обеспечивает, чтобы каждый брокер получал только по одному заданию за раз и чтобы ни одно задание не обрабатывалось более чем одним работником.

На рис. 8.16 показан процесс работы очереди сообщений:



Рис. 8.16. Асинхронное исполнение с использованием очереди сообщений и работников

Производитель отправляет сообщение в очередь, а работники обрабатывают сообщения в порядке очереди; первое добавленное в очередь сообщение является первым, которое должно быть обработано работником(ами).

Для того чтобы управлять очередью сообщений, нужен **брокер сообщений**. Брокер сообщений используется для транслирования сообщений в формаль-

¹ Англ. worker – Прим. перев.

ный протокол обмена сообщениями и управления очередями для нескольких получателей. Он обеспечивает надежное хранение и гарантированную доставку сообщений. Брокер сообщений позволяет создавать очереди, маршрутизировать сообщения, распределять сообщения между работниками и т. д.

Использование Django с Celery и RabbitMQ

Celery – это очередь заданий, которая может обрабатывать огромное количество сообщений. Она будет использоваться для формирования асинхронных заданий как функций Python в приложениях Django. Мы будем запускать работников Celery, которые будут прослушивать брокера сообщений, чтобы получать новые сообщения для обработки асинхронных заданий.

Используя очередь заданий Celery, можно не только легко создавать асинхронные задания и обеспечивать их наискорейшее исполнение работниками, но и планировать их запуск в конкретное время. Документация Celery находится на странице <https://docs.celeryq.dev/en/stable/index.html>.

Очередь заданий Celery общается посредством сообщений и требует, чтобы брокер сообщений был посредником между клиентами и работниками. С Celery могут работать несколько вариантов брокера сообщений, включая хранилища данных в формате ключ/значение, такие как Redis, или фактический брокер сообщений, такой как RabbitMQ.

Брокер сообщений RabbitMQ является наиболее широко используемым из всех. Он поддерживает несколько протоколов обмена сообщениями, таких как продвинутый протокол очередности сообщений AMQP¹, и является рекомендуемым обработчиком сообщений для очереди заданий Celery. Брокер сообщений RabbitMQ легковесен, легко разворачивается и приспособлен под конфигурирование с целью обеспечения масштабируемости и высокой доступности.

На рис. 8.17 показано, как Django, Celery и RabbitMQ будут использоваться для исполнения асинхронных заданий:



Рис. 8.17. Архитектура асинхронных заданий с участием Django, RabbitMQ и Celery

¹ Англ. Advanced Message Queuing Protocol. – Прим. перев.

Установка очереди заданий Celery

Давайте установим очередь заданий Celery и интегрируем ее в проект. Установите Celery посредством `pip`, используя следующую ниже команду:

```
pip install celery==5.2.7
```

Введение в очередь заданий Celery находится на странице <https://docs.celeryq.dev/en/stable/getting-started/introduction.html>.

Установка брокера сообщений RabbitMQ

Сообщество RabbitMQ предоставляет образ Docker, который упрощает развертывание сервера RabbitMQ со стандартной конфигурацией. Напомним, что вы научились устанавливать Docker в главе 7 «Отслеживание действий пользователя».

После установки Docker на свой компьютер можно легко получить образ брокера сообщений RabbitMQ платформы Docker, выполнив следующую ниже команду из командной оболочки:

```
docker pull rabbitmq
```

Она скачает образ RabbitMQ платформы Docker на ваш локальный компьютер. Информация об официальном образе RabbitMQ платформы Docker находится на странице https://hub.docker.com/_/rabbitmq.

Если вы хотите установить RabbitMQ исходно на свой компьютер вместо использования Docker, то подробные руководства по установке для различных операционных систем находятся на странице <https://www.rabbitmq.com/download.html>.

Исполните следующую ниже команду в оболочке, чтобы запустить сервер RabbitMQ с Docker:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672  
rabbitmq:management
```

Эта команда сообщает RabbitMQ о том, что нужно работать на порту 5672, и затем запускается его пользовательский веб-интерфейс управления на порту 15672.

Вы увидите результат, который содержит следующие ниже строки:

```
Starting broker...  
...  
completed with 4 plugins.  
Server startup complete; 4 plugins started.
```

Сервер брокера сообщений RabbitMQ работает на порту 5672 и готов принимать сообщения.

Доступ к встроенному в RabbitMQ интерфейсу управления

Пройдите по URL-адресу `http://127.0.0.1:15672/` в своем браузере. Вы увидите экран входа во встроенный в RabbitMQ пользовательский интерфейс управления. Он будет выглядеть так:

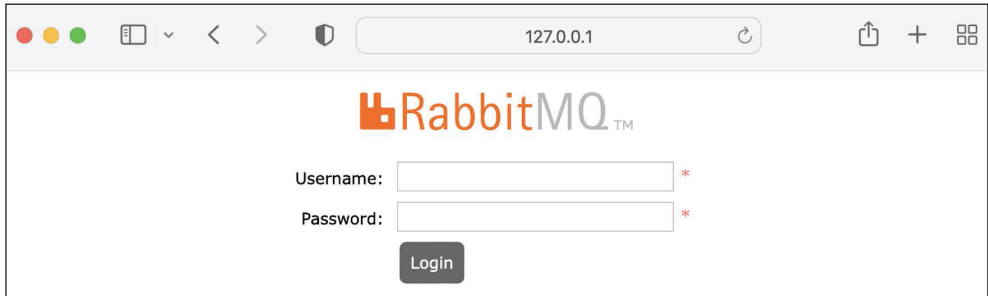


Рис. 8.18. Экран входа во встроенный в RabbitMQ пользовательский интерфейс управления

Введите `guest` в качестве пользовательского имени и пароль и кликните по **Login** (Войти). Вы увидите следующий ниже экран:

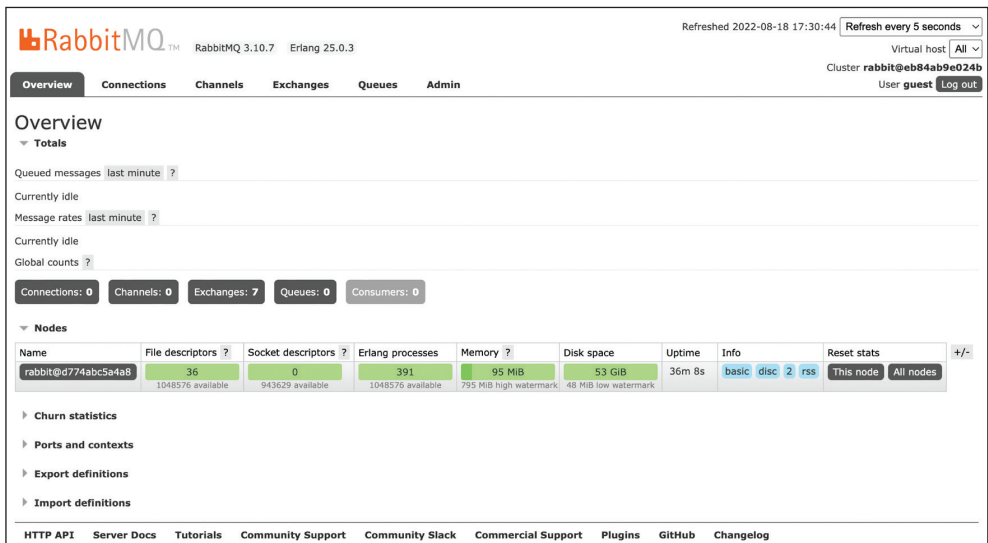


Рис. 8.19. Информационная панель пользовательского интерфейса управления

Это пользователь-администратор, используемый в RabbitMQ по умолчанию. На данном экране можно отслеживать текущую активность брокера сообщений RabbitMQ. Вы видите, что имеется один работающий узел без зарегистрированных соединений или очередей.

Если RabbitMQ работает в рабочей среде, то необходимо создать нового пользователя-администратора и удалить используемого по умолчанию гостевого пользователя. Это можно сделать в разделе **Admin** (Администратор) пользовательского интерфейса управления.

Теперь давайте добавим очередь заданий Celery в проект. Затем ее запустим и проверим соединение с брокером сообщений RabbitMQ.

Добавление очереди заданий Celery в проект

Экземпляру Celery необходимо предоставить конфигурацию. Рядом с файлом `settings.py` проекта `myshop` создайте новый файл и назовите его `celery.py`. Этот файл будет содержать конфигурацию Celery для вашего проекта. Добавьте в него следующий ниже исходный код:

```
import os
from celery import Celery

# задать стандартный модуль настроек Django
# для программы 'celery'.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')
app = Celery('myshop')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

В приведенном выше исходном коде делается следующее:

- задается переменная `DJANGO_SETTINGS_MODULE` для встроенной в Celery программы командной строки;
- посредством инструкции `app = Celery('myshop')` создается экземпляр приложения;
- используя метод `config_from_object()`, загружается любая конкретно-прикладная конфигурация из настроек проекта. Атрибут `namespace` задает префикс, который будет в вашем файле `settings.py` у настроек, связанных с Celery. Задав именованное пространство `CELERY`, все настройки Celery должны включать в свое имя префикс `CELERY_` (например, `CELERY_BROKER_URL`);
- наконец, сообщается, чтобы очередь заданий Celery автоматически обнаруживала асинхронные задания в ваших приложениях. Celery будет искать файл `tasks.py` в каждом каталоге приложений, добавленных в `INSTALLED_APPS`, чтобы загружать определенные в нем асинхронные задания.

Далее необходимо импортировать модуль `celery` в файл `__init__.py` проекта, чтобы он загружался при запуске Django.

Отредактируйте файл `myshop/__init__.py`, добавив в него следующий ниже исходный код:

```
# импортировать celery
from .celery import app as celery_app

__all__ = ['celery_app']
```

Вы добавили очередь заданий Celery в проект Django и теперь можете начать ее использовать.

Запуск работника Celery

Работник Celery – это процесс, или рабочий узел, который занимается служебными функциями, такими как отправка/получение сообщений очереди, регистрация заданий, уничтожение зависших заданий, отслеживание состояния и т. д. Экземпляр работника может потреблять любое число очередей сообщений.

Откройте еще одну оболочку и запустите работника Celery из каталога проекта, используя следующую ниже команду:

```
celery -A myshop worker -l info
```

Теперь работник Celery запущен и готов к обработке заданий. Давайте проверим наличие соединения между Celery и RabbitMQ.

Пройдите по URL-адресу <http://127.0.0.1:15672/> в своем браузере, чтобы получить доступ к встроенному в RabbitMQ пользовательскому интерфейсу управления. Вы увидите график в разделе **Queued messages** (Сообщения в очереди) и еще один в разделе **Message rates** (Частота сообщений), как на рис. 8.20.

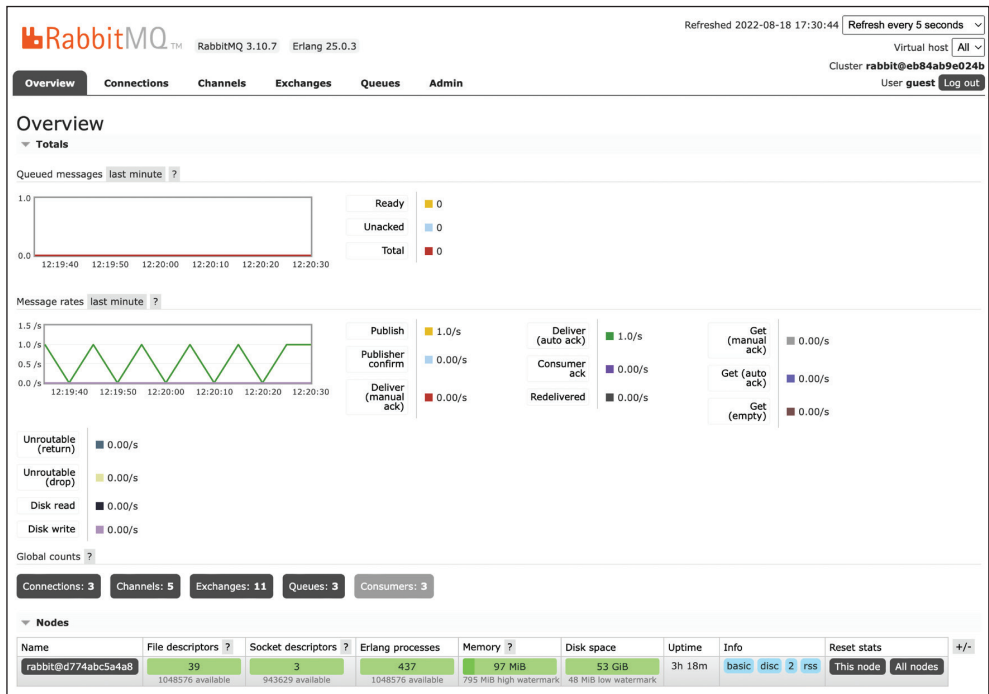


Рис. 8.20. Информационная панель управления RabbitMQ, отображающая соединения и очереди

Очевидно, что сообщений в очереди нет, так как мы еще не отправляли сообщения в очередь сообщений. График в разделе **Message rates** (Частота сообщений) должен обновляться каждые пять секунд; частоту обновления можно увидеть в правом верхнем углу экрана. На этот раз и **Connections** (Соединения), и **Queues** (Очереди) должны отображать число больше нуля.

Теперь можно начать программировать асинхронные задания.



Параметр `CELERY_ALWAYS_EAGER` позволяет исполнять задания локально в синхронном режиме, не отправляя их в очередь. Это удобно при выполнении модульных тестов или исполнении приложения в локальной среде без запуска программы Celery.

Добавление асинхронных заданий в приложение

Давайте будем отправлять пользователю электронное письмо о подтверждении всякий раз, когда заказ размещается в интернет-магазине. Мы реализуем отправку электронной почты в функции Python и зарегистрируем ее в Celery как задание. Затем мы добавим ее в представление `order_create` для асинхронного исполнения заданий.

При исполнении представления `order_create` Celery будет отправлять сообщение в очередь сообщений, управляемую брокером сообщений RabbitMQ, а затем брокер будет исполнять асинхронное задание, которое мы определили в функции Python.

По традиции простое обнаружение заданий Celery заключается в формулировании асинхронных заданий для приложения в модуле `tasks` в каталоге приложения.

Создайте новый файл внутри приложения `orders` и назовите его `tasks.py`. Это место, где Celery будет искать асинхронные задания. Добавьте в него следующий ниже исходный код:

```
from celery import shared_task
from django.core.mail import send_mail
from .models import Order

@shared_task
def order_created(order_id):
    """
    Задание по отправке уведомления по электронной почте
    при успешном создании заказа.
    """
    order = Order.objects.get(id=order_id)
    subject = f'Order nr. {order.id}'
    message = f'Dear {order.first_name},\n\n \
              f'You have successfully placed an order.' \
              f'Your order ID is {order.id}.'
    mail_sent = send_mail(subject,
```

```
        message,  
        'admin@myshop.com',  
        [order.email])  
  
    return mail_sent
```

Задание `order_created` было определено с помощью декоратора `@shared_task`. Как видите, задание Celery – это просто функция Python, декорированная функцией-декоратором `@shared_task`. Функция задания `order_created` получает параметр `order_id`. При исполнении задания рекомендуется всегда передавать идентификаторы функциям задания и извлекать объекты из базы данных. Тем самым избегается доступ к устаревшей информации, поскольку данные в базе данных могли измениться за то время, пока задание стояло в очереди. Для отправки уведомления по электронной почте разместившему заказ пользователю была использована предоставляемая веб-фреймворком Django функция `send_mail()`.

Вы научились конфигурировать Django под использование SMTP-сервера в главе 2 «Усовершенствование блога за счет продвинутых функциональностей». Если вы не хотите устанавливать настроечные параметры электронной почты, то можете сообщить Django, что нужно писать электронные письма в консоль, добавив следующий ниже настроечный параметр в файл `settings.py`:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```



Асинхронные задания следует использовать не только для времязатратных процессов, но и для других процессов, исполнение которых не занимает так много времени, но которые подвержены сбоям соединения либо требуют политики повторных попыток соединения.

Теперь необходимо добавить задание в представление `order_create`. Отредактируйте файл `views.py` приложения `orders`, импортировав задание и вызвав асинхронное задание `order_created` после очистки корзины, как показано ниже:

```
from .tasks import order_created  
#...  
  
def order_create(request):  
    # ...  
    if request.method == 'POST':  
        # ...  
        if form.is_valid():  
            # ...  
            cart.clear()  
            # запустить асинхронное задание
```

```
order_created.delay(order.id)
# ...
```

Метод `delay()` задания вызывается для его асинхронного исполнения. Задание будет добавлено в очередь сообщений и выполнено работником Celery как можно скорее.

Проверьте, чтобы RabbitMQ был запущен. Затем остановите процесс работника Celery и следующей ниже командой снова его запустите:

```
celery -A myshop worker -l info
```

Теперь работник Celery зарегистрировал задание. В еще одной оболочке следующей ниже командой запустите сервер разработки из каталога проекта:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/` в браузере, добавьте товары в корзину и завершите заказ. В оболочке, в которой вы запустили работника очереди заданий Celery, вы увидите примерно такой результат:

```
[2022-02-03 20:25:19,569: INFO/MainProcess] Task orders.tasks.order_
created[a94dc22e-372b-4339-bff7-52bc83161c5c] received
...
[2022-02-03 20:25:19,605: INFO/ForkPoolWorker-8] Task orders.tasks.
order_created[a94dc22e-372b-4339-bff7-52bc83161c5c] succeeded in
0.015824042027816176s: 1
```

Задание `order_created` было исполнено, и уведомление о заказе отправлено по электронной почте. Если вы используете почтовый бэкэнд `console.EmailBackend`, то электронная почта не отправляется, но вы будете видеть электронную почту в консоли.

Отслеживание Celery с помощью инструмента Flower

Помимо встроенного в RabbitMQ пользовательского интерфейса управления, можно использовать другие инструменты мониторинга асинхронных заданий, исполняемых с помощью Celery. Flower представляет собой удобный веб-инструмент для отслеживания работы Celery.

Следующей ниже командой установите Flower:

```
pip install flower==1.1.0
```

После установки мониторингового инструмента Flower можно запустить следующей ниже командой в новой оболочке из каталога проекта:

```
celery -A myshop flower
```

Пройдите по URL-адресу `http://localhost:5555/dashboard` в своем браузере. Вы увидите активных работников Celery и статистику асинхронных заданий. Экран должен выглядеть следующим образом:

The screenshot shows the Flower dashboard interface. At the top, there are tabs for 'Dashboard', 'Tasks', and 'Broker'. Below the tabs, there are five summary boxes: 'Active: 0', 'Processed: 0', 'Failed: 0', 'Succeeded: 0', and 'Retried: 0'. A search bar is located below these boxes. The main content is a table with the following columns: Worker Name, Status, Active, Processed, Failed, Succeeded, Retried, and Load Average. One worker is listed: 'celery@Antonios-MBP.home' with a status of 'Online' and a load average of '2.91, 3.11, 3.88'. A 'Showing 1 to 1 of 1 entries' message is at the bottom of the table.

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@Antonios-MBP.home	Online	0	0	0	0	0	2.91, 3.11, 3.88

Рис. 8.21. Информационная панель Flower

Вы увидите активного работника, имя которого начинается с **celery@** и статус которого будет **Online**.

Кликните по имени работника, а затем перейдите на вкладку **Queues** (Очереди). Вы увидите следующий ниже экран:

The screenshot shows the 'Queues' page for the worker 'celery@Antonios-MBP.home'. It features a 'Refresh' button and a navigation menu with 'Pool', 'Broker', 'Queues', 'Tasks', 'Limits', 'Config', 'System', and 'Other'. Below the menu, there is a section for 'Active queues being consumed from' with an 'Add Consumer' button. A table lists the active queue with the following columns: Name, Exclusive, Durable, Routing key, No ACK, Alias, Queue arguments, Binding arguments, and Auto delete. The table contains one entry: 'celery' with Exclusive: False, Durable: True, Routing key: celery, No ACK: False, Alias: None, Queue arguments: None, Binding arguments: None, and Auto delete: False. A 'Cancel Consumer' button is located at the bottom right of the table.

Name	Exclusive	Durable	Routing key	No ACK	Alias	Queue arguments	Binding arguments	Auto delete
celery	False	True	celery	False	None	None	None	False

Рис. 8.22. Flower – Очереди заданий работника Celery

Здесь можно увидеть активную очередь с именем **celery**. Это активный потребитель очереди, соединенный с брокером сообщений.

Откройте вкладку **Tasks** (Задания). Вы увидите следующий ниже экран:

Worker: **celery@Antonios-MBP.home**

Refresh

Pool Broker Queues **Tasks** Limits Config System Other

Processed number of completed tasks

orders.tasks.order_created	5
----------------------------	---

Рис. 8.23. Flower – Задания работника Celery

Здесь вы увидите обработанные задания и количество их исполнений. Вы должны увидеть задание `order_created` и общее время его исполнения. Это число может варьироваться в зависимости от того, сколько заказов было размещено.

Пройдите по URL-адресу `http://localhost:8000/` в своем браузере. Добавьте несколько товаров в корзину, а затем завершите процесс оформления заказа.

Пройдите по URL-адресу `http://localhost:5555/dashboard` в браузере. Мониторинговый инструмент Flower зарегистрировал задание как обработанное. Теперь вы должны увидеть 1 в столбце **Processed** (Обработанное) и 1 в столбце **Succeeded** (Успешное):

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@Antonios-MBPHome	Online	0	1	0	1	0	5.51, 4.14, 4.2

Рис. 8.24. Flower – Работники Celery

В разделе **Tasks** (Задания) вы увидите дополнительные сведения о каждом задании, зарегистрированном в Celery:

Flower Dashboard **Tasks** Broker Docs Code

Show 10 entries Search:

Name	UUID	State	args	kwargs	Result	Received	Started	Runtime	Worker
orders.tasks.order_created	abb1048f-9a2a-4b59-8b39-8c6b804bebbc	SUCCESS	(10,)	{}	1	2022-02-06 15:54:55.803	2022-02-06 15:54:55.999	0.086	celery@Antonios-MBPHome

Showing 1 to 1 of 1 entries Previous 1 Next

Рис. 8.25. Flower – Задания Celery

Документация по мониторинговому инструменту Flower находится на странице <https://flower.readthedocs.io/>.

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/Chapter08>.
- Статические файлы проекта: <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter08/myshop/shop/static>.
- Настройки сеанса Django: <https://docs.djangoproject.com/en/4.1/ref/settings/#sessions>.
- Встроенные в Django процессоры контекста: <https://docs.djangoproject.com/en/4.1/ref/templates/api/#built-in-template-context-processors>.
- Информация о RequestContext: <https://docs.djangoproject.com/en/4.1/ref/templates/api/#django.template.RequestContext>.
- Документация Celery: <https://docs.celeryq.dev/en/stable/index.html>.
- Введение в очередь заданий Celery: <https://docs.celeryq.dev/en/stable/getting-started/introduction.html>.
- Официальный образ брокера сообщений RabbitMQ платформы Docker: https://hub.docker.com/_/rabbitmq.
- Инструкции по установке брокера сообщений RabbitMQ: <https://www.rabbitmq.com/download.html>.
- Документация по мониторинговому инструменту Flower: <https://flower.readthedocs.io/>.

Резюме

В этой главе вы создали базовое приложение электронной коммерции. Вы разработали каталог товаров и сформировали корзину на основе сеансов. Вы имплементировали конкретно-прикладной процессор контекста, чтобы сделать корзину доступной для всех шаблонов, и создали форму для размещения заказов. Вы также научились реализовывать асинхронные задания с помощью очереди заданий Celery и брокера сообщений RabbitMQ.

В следующей главе вы научитесь интегрировать платежный шлюз в свой магазин, добавлять конкретно-прикладные действия на сайт администрирования, экспортировать данные в формате CSV и динамически создавать PDF-файлы.