

Создание проекта Интернет-Магазина

Вам необходимо создать интернет-магазин для продажи любых товаров (отличных от образца). Задание включает в себя как обязательную часть, так и дополнительное задание, выполнение которого положительно скажется на оценках за КР и экзамен. Сразу стоит отметить, что версии фреймворков указаны в качестве примера и вам необязательно использовать именно эти версии (так как возможно, что они уже устарели), поэтому перед выполнением, ознакомьтесь с актуальными версиями и используйте их.

Создайте новый проект Django, чтобы разработать Интернет-Магазин. Пользователи смогут просматривать каталог продуктов и добавлять продукты в корзину для покупок. Будут охвачены следующие функциональные возможности Интернет-магазина:

- Создание моделей каталога продуктов, добавление их на сайт администрирования и создание основных представлений для отображения каталога.
- Создание системы корзины для покупок с помощью сессий Django, чтобы пользователи могли сохранять выбранные продукты при просмотре сайта.
- Создание форм и функциональных возможностей для размещения заказов.
- Отправка по электронной почте подтверждение пользователям при размещении заказа.

Сначала создайте виртуальную среду для нового проекта и активируйте ее следующим образом:

```
mkdir env
virtualenv env/myshop
source env/myshop/bin/activate
```

Установите Django в виртуальной среде с помощью следующей команды (или другой, более новой версии):

```
pip install Django==1.8.6
```

Начните новый проект под названием **myshop** с приложением, называемым **shop**, открыв терминал и выполнив следующие команды:

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

Измените файл **settings.py** проекта и добавьте приложение к настройкам **INSTALLED_APPS**:

```
INSTALLED_APPS = (
    # ...
    'shop',
)
```

Теперь приложение активно для данного проекта. Давайте определим модели для каталога продуктов.

Создание моделей каталога продуктов

Каталог нашего магазина будет состоять из продуктов, сгруппированных по разным категориям. Каждый продукт будет иметь имя, необязательное описание, необязательное изображение, цену и доступный запас. Отредактируйте файл **models.py** только что созданного приложения **shop** и добавьте следующий код:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True, unique=True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'Категория'
        verbose_name_plural = 'Категории'

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category, related_name='products')
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    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)
    stock = models.PositiveIntegerField()
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ('name',)
        index_together = (('id', 'slug'),)

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

Это модели **Category** и **Product**. Модель **Category** состоит из поля **name** и **slug**. Рассмотрим поля модели **Product**:

- **category** : Это **ForeignKey** модели **Category**. Это отношение "многие к одному": продукт относится к одной категории, а категория содержит несколько продуктов
- **name** : Название продукта.
- **slug** : Алиас продукта(его URL).
- **image** : Изображение продукта.
- **description** : Необязательное описание для продукта.
- **price** : Это поле **DecimalField**. В нем используется десятичное число Python. Десятичный тип для хранения десятичного числа с фиксированной точностью. Максимальное число цифр (включая десятичные разряды) задается с помощью атрибута **max_digits** и десятичных знаков с атрибутом **decimal_places**
- **stock** : Это поле **PositiveIntegerField** для хранения остатков данного продукта.

- **available** : Это булево значение, указывающее, доступен ли продукт или нет. Позволяет включить/отключить продукт в каталоге.
- **created** : Это поле хранит дату когда был создан объект.
- **updated** : В этом поле хранится время последнего обновления объекта.

Для поля **price** мы используем **DecimalField** вместо **FloatField**, чтобы избежать проблем округления.

*Всегда используйте **DecimalField** для хранения денежных сумм. **FloatField** использует плавающий тип Python внутри, в то время как **DecimalField** использует десятичный тип Python. С помощью десятичного типа можно избежать проблем с округлением с плавающей точкой.*

В классе **Meta** модели **Product** мы используем параметр мета **index_together**, чтобы задать индекс для полей **id** и **slug**. Мы определим этот индекс, поскольку мы планируем запросить продукты с помощью **id** и **slug**. Оба поля индексируются вместе для улучшения представлений для запросов, использующих эти два поля.

Поскольку мы собираемся использовать изображения в наших моделях, откройте терминал и установите **Pillow** следующей командой:

```
pip install Pillow==2.9.0
```

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

```
python manage.py makemigrations
```

Вы увидите следующее:

```
Migrations for 'shop':
  0001_initial.py:
    - Create model Category
    - Create model Product
    - Alter index_together for product (1 constraint(s))
```

Выполните следующую команду для синхронизации базы данных:

```
python manage.py migrate
```

Вы увидите следующее:

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

Теперь база данных синхронизирована с моделями.

Регистрация моделей **catalog** на сайте администрирования

Давайте добавим наши модели на сайт администрирования, чтобы мы могли легко управлять категориями и продуктами. Измените файл **admin.py** приложения **shop** и добавьте в него следующий код:

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

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

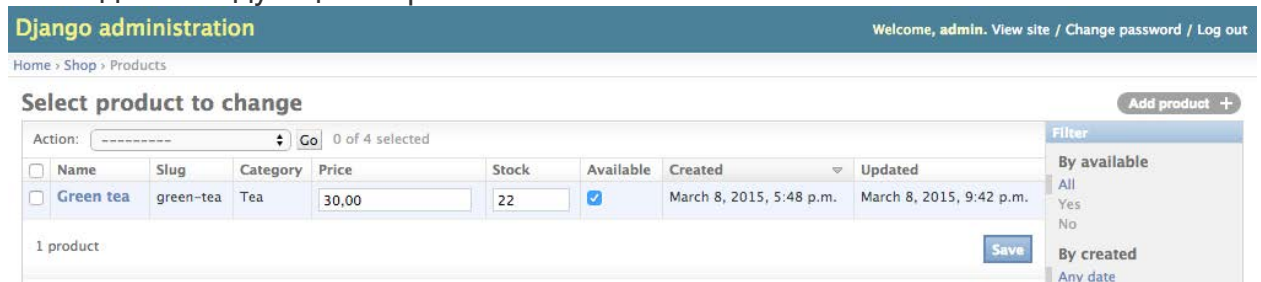
```
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price', 'stock', 'available', 'created',
'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'stock', 'available']
    prepopulated_fields = {'slug': ('name',)}
admin.site.register(Product,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

Помните, что мы используем атрибут **prepopulated_fields**, чтобы указать поля, в которых значение автоматически задается с использованием значения других полей. Как вы уже видели, это удобно для создания алиасов(slug). Атрибут **list_editable** в классе **ProductAdmin** используется для задания полей, которые могут быть отредактированы на странице отображения списка сайта администрирования. Это позволит редактировать несколько строк одновременно. Любое поле в **list_editable** также должно быть указано в атрибуте **list_display**, поскольку могут быть изменены только отображаемые поля.

Теперь создайте суперпользователя следующей командой:

```
python manage.py createsuperuser
```

Запустите сервер разработки командой `python manage.py runserver`. Откройте в браузере <http://127.0.0.1:8000/admin/shop/product/add/> и войдите в систему с помощью только что созданного пользователя. Добавьте новую категорию и продукт с помощью интерфейса администрирования. Страница Products будет выглядеть следующим образом:



Регистрация моделей catalog на сайте администрирования

Давайте добавим наши модели на сайт администрирования, чтобы мы могли легко управлять категориями и продуктами. Измените файл **admin.py** приложения **shop** и добавьте в него следующий код:

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

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

class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price', 'stock', 'available', 'created',
'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'stock', 'available']
```

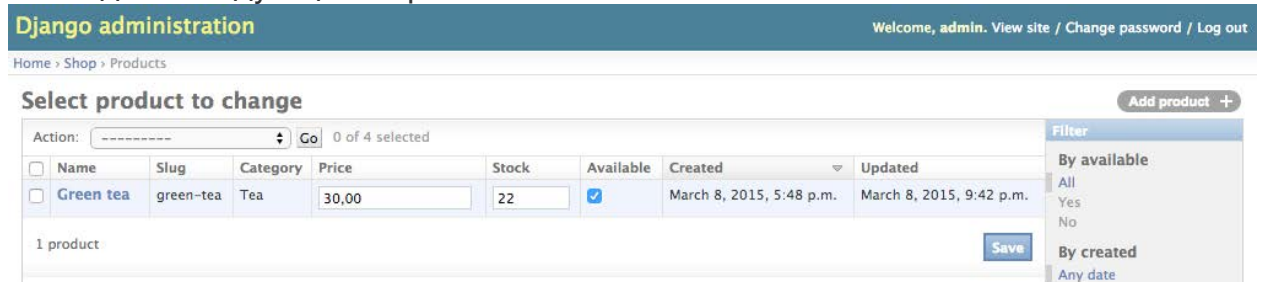
```
prepopulated_fields = {'slug': ('name',)}
admin.site.register(Product, ProductAdmin)
```

Помните, что мы используем атрибут **prepopulated_fields**, чтобы указать поля, в которых значение автоматически задается с использованием значения других полей. Как вы уже видели, это удобно для создания алиасов(slug). Атрибут **list_editable** в классе **ProductAdmin** используется для задания полей, которые могут быть отредактированы на странице отображения списка сайта администрирования. Это позволит редактировать несколько строк одновременно. Любое поле в **list_editable** также должно быть указано в атрибуте **list_display**, поскольку могут быть изменены только отображаемые поля.

Теперь создайте суперпользователя следующей командой:

```
python manage.py createsuperuser
```

Запустите сервер разработки командой `python manage.py runserver`. Откройте в браузере <http://127.0.0.1:8000/admin/shop/product/add/> и войдите в систему с помощью только что созданного пользователя. Добавьте новую категорию и продукт с помощью интерфейса администрирования. Страница Products будет выглядеть следующим образом:



Создание шаблонов каталога

Теперь необходимо создать шаблоны для списка товаров и одного товара. Создайте следующую структуру файлов в каталоге приложения **shop**:

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

Необходимо определить базовый шаблон, а затем расширить его в `product list` и в шаблоне `detail`. Отредактируйте шаблон **shop/base.html** и добавьте в него следующий 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 мы будем использовать для нашего магазина. Чтобы включить требуемые стили CSS и изображения, используемые шаблонами, необходимо скопировать статические файлы, которые входят в эту главу, расположенную в каталоге **static/** приложения **shop**. Скопируйте их в ту же папку проекта.

Измените шаблон **shop/product/list.html** и добавьте в него следующий 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 %}

```

Это шаблон списка продуктов. Он расширяет шаблон **shop/base.html** и использует переменную контекста категорий для отображения всех категорий на боковой панели и продуктов для отображения продуктов текущей страницы. Один и тот же шаблон используется для обоих типов: список всех доступных продуктов и список продуктов, отфильтрованных по категориям. Поскольку поле изображения модели продукта может быть пустым, мы должны предоставить изображение по умолчанию для продуктов, у которых

нет изображения. Изображение находится в каталоге статических файлов с относительным путем `img/no_image.png`.

Поскольку мы используем **ImageField** для хранения изображений продуктов, нам необходим сервер разработки для обслуживания загруженных файлов изображений. Отредактируйте файл **settings.py** myshop и добавьте следующие параметры:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

MEDIA_URL является базовым URL-адресом, который обслуживает файлы мультимедиа, загруженные пользователями. **MEDIA_ROOT** — это локальный путь, в котором находятся эти файлы, который мы строим динамически в зависимости от переменной **BASE_DIR**.

Чтобы Django обрабатывал загруженные мультимедийные файлы с помощью сервера разработки, отредактируйте файл **urls.py** myshop следующим образом:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ...
]

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

Помните, что в процессе разработки мы используем только статические файлы. В продакшн версии сайта никогда не следует использовать статические файлы с Django.

Добавьте в магазин несколько продуктов с помощью сайта администрирования и откройте в браузере <http://127.0.0.1:8000/>. Появится страница списка продуктов, которая выглядит следующим образом:

My shop


Your cart is empty.

Categories


All

Tea


Products



Green tea
\$30



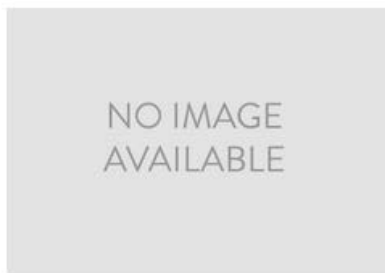
Red tea
\$45.5



Tea powder
\$21.2

Если вы создаете продукт с помощью сайта администрирования и не загрузите изображение, будет отображаться изображение по умолчанию `no_image.png`:

Products



Black tea
\$32.20



Tea powder
\$21.20



Red tea
\$45.50

Давайте отредактируем шаблон сведений о продукте. Измените шаблон **shop/product/detail.html** и добавьте в него следующий код:

```
{% extends "shop/base.html" %}
{% load static %}
{% block title %}
    {% if category %}{{ category.title }}{% else %}Products{% endif %}
{% 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 %}
```

Мы вызываем метод **get_absolute_url()** для объекта категории, чтобы отобразить доступные продукты, принадлежащие к одной и той же категории. Теперь откройте <http://127.0.0.1:8000/> в браузере и щелкните любой продукт, чтобы увидеть страницу сведений о продукте. Она будет выглядеть следующим образом:

My shop

Your cart is empty.



Red tea

Tea

\$45.5

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.

Мы создали базовый каталог продуктов.

Создание корзины

После создания каталога продуктов следующим шагом является создание корзины покупок, которая позволит пользователям выбирать продукты, которые они хотят приобрести. Корзина позволяет пользователям выбирать нужные продукты и временно хранить их во время просмотра сайта до тех пор, пока не будет размещен заказ. Корзина должна быть сохранена в сессии, чтобы элементы корзины хранились во время визита пользователя.

Мы будем использовать Django's session framework для сохранения товаров корзины. Корзина будет храниться в сессии до тех пор, пока она не завершится. Нам также потребуется построить дополнительные модели Джанго для корзины и ее товаров.

Использование сессий Django

Джанго предоставляет session framework, поддерживающую анонимные и пользовательские сессии. Session framework позволяет хранить произвольные данные для каждого посетителя. Данные сеанса хранятся на стороне сервера, а файлы cookie содержат session ID, если не используется обработчик сессий на основе файлов cookie. Промежуточное подпрограммное обеспечение управляет отправкой и получением файлов cookie. Обработчик сессий по умолчанию хранит данные сессии в базе данных, но, как вы увидите далее, можно выбрать между различными обработчиками сессий.

Чтобы использовать сессии, необходимо убедиться, что параметр **MIDDLEWARE_CLASSES** проекта содержит 'django.contrib.sessions.middleware.SessionMiddleware'. Это промежуточное программное обеспечение управляет сессиями и добавляется по умолчанию при создании нового проекта с помощью команды startproject.

Промежуточное программное обеспечение позволяет сделать текущую сессию доступной в объекте request. Доступ к текущей сессии можно получить с помощью **request.session**, используя его аналогично словарю Python для хранения и извлечения данных сессии. Словарь сессий по умолчанию принимает любой объект Python, который может быть сериализован в JSON. Можно задать переменную в сессии следующим образом:

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

Извлечение session key:

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

Удалить key, хранящийся в session:

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

Как вы видели, мы только что обрабатывали **request.session**, как стандартный словарь Python.

При входе пользователей в сайт их анонимная сессия теряется и создается новая сессия для пользователей, прошедших авторизацию. При хранении элементов в анонимной сессии, которую необходимо сохранить после входа пользователей в систему, необходимо будет скопировать старые данные сессии в новую сессию.

Настройки сессий

Существует несколько параметров, которые можно использовать для настройки сессий проекта. Самое главное — **SESSION_ENGINE**. Этот параметр позволяет задать место хранения сессий. По умолчанию Django хранит сессии в базе данных, используя модель **Session** приложения **django.contrib.sessions**.

Django предлагает следующие варианты хранения данных сессий:

- **Database sessions**: Данные сессии хранятся в базе данных. Это default session engine.
- **File-based sessions**: Данные сессии хранятся в файловой системе.
- **Cached sessions**: Данные сессии хранятся в серверной части кэша. Можно указать конечные точки кэша с помощью параметра "CACHES". Хранение данных сессии в системе кэша обеспечивает наилучшую производительность.
- **Cached database sessions**: Данные сессии хранятся в кэше и базе данных. При чтении база данных используется только в том случае, если данные еще не находятся в кэше.
- **Cookie-based sessions**: Данные сеанса хранятся в файлах cookie, отправляемых в браузер.

Для повышения производительности используйте обработчик сессий на основе кэша. Django поддерживает Memcached, а также другие конечные точки кэша сторонних производителей для Redis и других систем кэша.

Можно настроить сессии с другими параметрами. Ниже приводятся некоторые важные параметры, относящиеся к сессиям:

- **SESSION_COOKIE_AGE** : Длительность сессии "cookie" в секундах. Значение по умолчанию — 1209600 (2 недели).
- **SESSION_COOKIE_DOMAIN** : Этот домен используется для сеансов "cookie". Установите это значение . mydomain.com для включения междоменных файлов cookie.
- **SESSION_COOKIE_SECURE** : Логическое значение, указывающее, что файл cookie должен быть отправлен только в том случае, если соединение является соединением HTTPS.
- **SESSION_EXPIRE_AT_BROWSER_CLOSE** : Это булево значение, указывающее, что сессия должна истечь при закрытии браузера.
- **SESSION_SAVE_EVERY_REQUEST** : Это логическое значение, которое, в случае True, сохранит сессию в базе данных по каждому запросу. Срок действия сессии также обновляется каждый раз.

Можно просмотреть все параметры сессий

здесь: <https://docs.djangoproject.com/en/1.8/ref/settings/#sessions>

Срок действия сессии

Можно использовать сессии browser-length или постоянные сессии с помощью параметра **SESSION_EXPIRE_AT_BROWSER_CLOSE**. По умолчанию для этого параметра установлено значение **False**, что приводит к тому, что длительность

сессии задается значением, хранящимся в параметре **SESSION_COOKIE_AGE**. Если установить **SESSION_EXPIRE_AT_BROWSER_CLOSE** значение **True**, срок действия сессии истечет, когда пользователь закроет браузер, и параметр **SESSION_COOKIE_AGE** не будет иметь никакого эффекта.

Можно использовать метод **set_expiry()** `request.session` для перезаписи продолжительности текущего сеанса.

Хранение корзины покупок в сессиях

Необходимо создать простую структуру, которая может быть сериализована в JSON для хранения элементов корзины в сессии. Корзина должна включать следующие данные для каждого содержащегося в ней элемента:

- **id** экземпляра **Product**
- Количество товара, выбранное для данного продукта
- Цена единицы для данного продукта

Поскольку цены на продукцию могут различаться, мы приближаемся к сохранению цены продукта вместе с продуктом, когда он добавляется в корзину. Таким образом, мы будем сохранять ту же цену, которую пользователи увидели при добавлении товара в корзину, даже если цена продукта изменится после этого.

Теперь необходимо управлять созданием корзин и связывать их с сеансами. Корзина покупок должна работать следующим образом:

- Когда требуется корзина, мы проверяем, установлен ли пользовательский `session key`. Если в сессии не задана корзина, мы создадим новую корзину и сохраним ее в `session key` корзины.
- Для последовательных запросов мы выполняем одну и ту же проверку и получая номенклатуры корзины из `session key` корзины. Мы извлекаем элементы корзины из базы данных и связанные с ними объекты продукта.

Измените файл **settings.py** проекта и добавьте в него следующий параметр:

```
CART_SESSION_ID = 'cart'
```

Это ключ, который мы собираемся использовать для хранения корзины в сессии пользователя.

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

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

Затем отредактируйте файл **settings.py** проекта и добавьте **"cart"** к параметру **INSTALLED_APPS** следующим образом:

```
INSTALLED_APPS = (  
    # ...  
    'shop',  
    'cart',  
)
```

Создайте новый файл в каталоге приложения **cart** и назовите его **cart.py**. Добавьте в него следующий код:

```
from decimal import Decimal
from django.conf import settings
from shop.models import Product

class Cart(object):

    def __init__(self, request):
        """
        Инициализируем корзину
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # save an empty cart in the session
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart
```

Это класс **Cart**, который позволит нам управлять корзиной для покупок. Требуется инициализация корзины с помощью объекта **request**. Мы храним текущую сессию с помощью **self.session = request.session**, чтобы сделать его доступным для других методов класса **Cart**. Во-первых, мы пытаемся получить корзину с текущей сессии с помощью **self.session.get(settings.CART_SESSION_ID)**. Если в сессии отсутствует корзина, то мы создадим сессию с пустой корзиной, установив пустой словарь в сессии. Мы ожидаем, что наш словарь корзины будет использовать коды продуктов в качестве ключей и словарь с количеством и ценой в качестве значения для каждого ключа. Таким образом, мы можем гарантировать, что продукт не будет добавлен в корзину более одного раза; можно также упростить доступ к данным элементов корзины.

Создадим метод для добавления продуктов в корзину или обновления их количества. Добавьте следующие методы **add()** и **save()** в класс **Cart**:

```
def add(self, product, quantity=1, update_quantity=False):
    """
    Добавить продукт в корзину или 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 update_quantity:
        self.cart[product_id]['quantity'] = quantity
    else:
        self.cart[product_id]['quantity'] += quantity
    self.save()

def save(self):
    # Обновление сессии cart
    self.session[settings.CART_SESSION_ID] = self.cart
    # Отметить сеанс как "измененный", чтобы убедиться, что он сохранен
    self.session.modified = True
```

Метод **add()** принимает следующие параметры:

- **product** : Экземпляр **Product** для добавления или обновления в корзине

- **quantity** : Необязательное целое число для количества продукта. По умолчанию используется значение 1 .
- **update_quantity** : Это логическое значение, которое указывает, требуется ли обновление количества с заданным количеством (True), или же новое количество должно быть добавлено к существующему количеству (False).

id продукта используется в качестве ключа в словаре содержимого корзины. **id** продукта преобразуется в строку, так как Django использует JSON для сериализации данных сессии, а JSON разрешает только имена строк. **id** продукта — это ключ, а значение, которое мы сохраняем, — словарь с количеством и ценой для продукта. Цена продукта преобразуется из десятичного разделителя в строку, чтобы сериализовать его. Наконец, мы вызываем метод **save()**, чтобы сохранить корзину в сессии.

Метод **save()** сохраняет все изменения в корзине в сессии и помечает сессию как **modified** с помощью **session.modified = True**. Это говорит о том, что сессия **modified** и должна быть сохранена.

Нам также нужен метод для удаления продуктов из корзины. Добавьте следующий метод в класс **Cart**:

```
def remove(self, product):
    """
    Удаление товара из корзины.
    """
    product_id = str(product.id)
    if product_id in self.cart:
        del self.cart[product_id]
        self.save()
```

Метод **remove()** удаляет заданный продукт из словаря корзины и вызывает метод **save()** для обновления корзины в сессии.

Нам придется перебрать элементы, содержащихся в корзине, и получить доступ к соответствующим экземплярам **Product**. Для этого в классе можно определить метод **__iter__()**. Добавьте следующий метод в класс **Cart**:

```
def __iter__(self):
    """
    Перебор элементов в корзине и получение продуктов из базы данных.
    """
    product_ids = self.cart.keys()
    # получение объектов product и добавление их в корзину
    products = Product.objects.filter(id__in=product_ids)
    for product in products:
        self.cart[str(product.id)]['product'] = product

    for item in self.cart.values():
        item['price'] = Decimal(item['price'])
        item['total_price'] = item['price'] * item['quantity']
        yield item
```

В методе **__iter__()** мы извлекаем экземпляры продукта, присутствующие в корзине, чтобы включить их в номенклатуры корзины. Наконец, мы проходим по элементам корзины, преобразуя цену номенклатуры обратно в десятичное число и добавляя атрибут **total_price** к каждому элементу. Теперь можно легко выполнить итерацию по товарам в корзине.

Нам также нужен способ вернуть общее количество товаров в корзине. Когда функция **len()** выполняется на объекте, Python вызывает метод **__len__()** для извлечения ее длины. Мы собираемся определить пользовательский метод **__len__()**, чтобы вернуть общее количество элементов, хранящихся в корзине. Добавьте следующий метод **__len__()** в класс **Cart**:

```
def __len__(self):
    """
    Подсчет всех товаров в корзине.
    """
    return sum(item['quantity'] for item in self.cart.values())
```

Мы возвращаем сумму количества всех товаров.

Добавьте следующий метод для расчета общей стоимости товаров в корзине:

```
def get_total_price(self):
    """
    Подсчет стоимости товаров в корзине.
    """
    return sum(Decimal(item['price']) * item['quantity'] for item in
                self.cart.values())
```

И, наконец, добавьте метод для очистки сеанса корзины:

```
def clear(self):
    # удаление корзины из сессии
    del self.session[settings.CART_SESSION_ID]
    self.session.modified = True
```

Теперь наш класс **Cart** готов к управлению корзиной для покупок.

Создание представлений корзины покупок

Теперь, когда у нас есть класс **Cart** для управления корзиной, необходимо создать представления для добавления, обновления или удаления элементов из нее. Необходимо создать следующие представления:

- Представление для добавления или обновления номенклатур в корзине, которое может обрабатывать текущие и новые количества
- Представление для удаления товаров из тележки
- Представление для отображения элементов корзины и итоговых значений

Добавление элементов в корзину

Чтобы добавить элементы в корзину, нам нужна форма, позволяющая пользователю выбрать количество добавляемого товара. Создайте файл **forms.py** в каталоге приложения **cart** и добавьте в него следующий 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)
    update = forms.BooleanField(required=False, initial=False,
widget=forms.HiddenInput)
```


Эта форма будет использоваться для добавления продуктов в корзину. Класс **CartAddProductForm** содержит следующие поля:

quantity : позволяет пользователю выбрать количество между 1-20. Мы используем поле **TypedChoiceField** с **coerce=int** для преобразования ввода в целое число.

update : позволяет указать, следует ли добавлять сумму к любому существующему значению в корзине для данного продукта (False) или если существующее значение должно быть обновлено с заданным значением (True). Для этого поля используется графический элемент **HiddenInput**, поскольку не требуется показывать его пользователю.

Создадим представление для добавления элементов в корзину. Отредактируйте файл **views.py** приложения **cart** и добавьте в него следующий код:

```
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'],
                 update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

Это представление для добавления продуктов в корзину или обновления количества для существующих продуктов. Мы используем декоратор **require_POST**, чтобы разрешить только POST запросы, поскольку это представление изменит данные. Представление получает **ID** продукта в качестве параметра. Мы извлекаем экземпляр продукта с заданным **ID** и проверяем **CartAddProductForm**. Если форма валидна, мы либо добавляем, либо обновляем продукт в корзине. Представление перенаправляет по URL-адресу **cart_detail**, который будет отображать содержимое корзины. Мы собираемся создать **cart_detail** представление в ближайшее время.

Нам также требуется представление для удаления товаров из корзины. Добавьте следующий код в файл **views.py** приложения **cart**:

```
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 id продукта в качестве параметра. Мы извлекаем экземпляр продукта с заданным id и удаляем продукт из корзины. Затем мы перенаправляем пользователя на URL-адрес **cart_detail**.

Наконец, требуется представление для отображения корзины и ее товаров. Добавьте следующий вид в файл **views.py**:

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

Представление **cart_detail** выводит на экран текущее состояние корзины.

Мы создали представления для добавления товаров в корзину, обновления количества, удаления товаров из корзины и отображения корзины. Рассмотрим добавление шаблонов URL-адресов для этих представлений. Создайте новый файл в каталоге приложения **cart** и назовите его **urls.py**. Добавьте к нему следующие URL-адреса:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.cart_detail, name='cart_detail'),
    url(r'^add/(?P<product_id>\d+)/$', views.cart_add, name='cart_add'),
    url(r'^remove/(?P<product_id>\d+)/$', views.cart_remove, name='cart_remove'),
]
```

Измените основной файл **urls.py** преркта **myshop** и добавьте следующий шаблон URL-адреса для включения URL-адресов корзины:

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^cart/', include('cart.urls', namespace='cart')),
    url(r'^$', include('shop.urls', namespace='shop')),
]
```

Убедитесь, что этот шаблон URL-адреса был включен до **shop.urls**, поскольку он является более ограничительным, чем последний.

Создание шаблона для отображения корзины

Представления **cart_add** и **cart_remove** не нужны в шаблонах, но необходимо создать шаблон для представления **cart_detail** для отображения элементов корзины и итоговых значений.

Создайте следующую структуру файла в каталоге приложения **cart**:

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

Измените шаблон **cart/detail.html** и добавьте в него следующий 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><a href="{% url 'cart:cart_remove"
product.id%}">Remove</a></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 %}

```

Это шаблон, используется для отображения содержимого корзины. Он содержит таблицу с элементами, хранящимися в текущей корзине. Мы разрешаем пользователям изменять количество выбранных продуктов, используя форму, которая учитывается в представлении **cart_add**. Мы также разрешаем пользователям удалять элементы из корзины, предоставляя для каждого из них ссылку **Remove**.

Добавление товаров в корзину

Теперь необходимо добавить кнопку «Добавить в корзину» на страницу сведений о продукте. Отредактируйте файл **views.py** приложения **shop** и добавьте **CartAddProductForm** в представление **product_detail**, следующим образом:

```

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** и добавьте следующую форму цены продукта следующим образом:

```

<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>

```

Убедитесь, что сервер разработки работает командой `python manage.py runserver`. Теперь откройте в браузере <http://127.0.0.1:8000/> и перейдите к странице сведений о продукте. Теперь он содержит форму для выбора количества перед добавлением продукта в корзину. Страница будет выглядеть следующим образом:

My shop

Your cart is empty.



Green tea

Tea


\$30.00

Quantity:

Add to cart

Выберите количество и нажмите кнопку "Добавить в корзину". Форма передается в **cart_add** view через POST. Представление добавляет продукт в корзину в сессии, включая текущую цену и выбранное количество. Затем он перенаправляет пользователя на страницу сведений о корзине, которая будет выглядеть как на следующем снимке экрана:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Green tea	2	Remove	\$30.00	\$60.00
Total					\$60.00

[Continue shopping](#)[Checkout](#)

Создание обработчика контекста для текущей корзины

Возможно, вы заметили, что в заголовке сайта отображается сообщение, сообщающее о том, что корзина пуста. Когда мы начнем добавлять товар в корзину, мы увидим общее количество товаров в корзине и общую стоимость. Поскольку это должно отображаться на всех страницах, мы построим обработчик контекста для включения текущей корзины в контекст запроса, независимо от обрабатываемого представления.

Контекстные процессоры

Контекстный процессор(context processor) — это функция Python, которая принимает объект запроса в качестве аргумента и возвращает словарь, добавляемый в контекст запроса. Они удобны, когда необходимо сделать что-то доступным для всех шаблонов.

По умолчанию при создании нового проекта с помощью команды **startproject** в проекте будут содержаться следующие контекстные процессоры шаблона в параметре **context_processors** внутри параметров **TEMPLATES**:

- **django.template.context_processors.debug** : задает логические переменные **debug** и **sql_queries** в контексте, представляющем список запросов SQL, выполненных в запросе
- **django.template.context_processors.request** : задает переменную запроса в контексте
- **django.contrib.auth.context_processors.auth** : задает пользовательскую переменную в запросе
- **django.contrib.messages.context_processors.messages** : При использовании данного метода переменная **messages** устанавливается в контексте, содержащем все сообщения, отправленные с помощью **messages framework**

Джанго также включает **django.template.context_processors.csrf** во избежание нападений с помощью межузловых запросов. Этот обработчик контекста не присутствует в настройках, но он всегда включен и не может быть отключен по соображениям безопасности.

Вы можете прочитать больше про контекстные процессоры здесь: <https://docs.djangoproject.com/en/1.8/ref/templates/api/#built-in-template-context-processors>

Настройка корзины в контексте запроса

Создадим обработчик контекста для установки текущей корзины в контекст запроса для шаблонов. Мы сможем получить доступ к этой корзине в любом шаблоне.

Создайте новый файл в каталоге приложения **cart** и назовите его **context_processors.py**. Контекстные процессоры могут размещаться в любом месте кода, но создание их в отдельном документе поможет вам лучше организовать структуру проекта. Добавьте в файл следующий код:

```
from .cart import Cart

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

Как видно, контекстный процессор — это функция, которая получает объект запроса в качестве параметра и возвращает словарь объектов, которые будут доступны всем шаблонам, визуализированным с помощью **RequestContext**. В нашем обработчике контекста мы создаем объект корзины с помощью объекта **request** и делаем его доступным для шаблонов в виде переменной с именем **cart**.

Отредактируйте файл **Settings.py** проекта и добавьте **'cart.context_processors.cart'** для параметра **context_processors** в параметрах **TEMPLATES**. После изменения этот параметр будет выглядеть следующим образом:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')]
    },
    {
        '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',
            ],
        },
    },
]
```

Теперь контекстный процессор будет выполняться при каждом просмотре шаблона с использованием **RequestContext** Джанго. Переменная **cart** будет задана в контексте для шаблонов.

Контекстные процессоры выполняются во всех запросах, использующих `RequestContext`. Если вы собираетесь получить доступ к базе данных, может понадобиться создать пользовательский тег шаблона вместо контекстного процессора.

Теперь отредактируйте шаблон **shop/base.html** приложения **shop** и найдите:

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

Замените предыдущие строки следующим кодом:

```
<div class="cart">
    {% with total_items=cart|length %}
        {% if cart|length > 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>
```

Запустите сервер с помощью команды `python manage.py runserver`. Откройте в браузере <http://127.0.0.1:8000/> и добавьте в корзину несколько товаров. Теперь в заголовке сайта можно будет увидеть общее количество товаров в корзине и их общую стоимость:

My shop

Your cart: 2 items, \$60.00

Регистрация заказов

При извлечении товаров из корзины для покупок необходимо сохранить заказ в базе данных. Заказы будут содержать информацию о клиентах и продуктах, которые они покупают.

Создайте новое приложение для управления заказами клиентов, используя следующую команду:

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

Отредактируйте **settings.py** вашего проекта и добавьте приложение **'orders'** в параметры **INSTALLED_APPS**:

```
INSTALLED_APPS = (
    # ...
    'orders',
)
```

Вы активировали новое приложение.

Создание модели order

Потребуется модель для хранения сведений о заказе и второй модели для хранения купленных товаров, включая их количество и цену. Измените файл **models.py** приложения **orders** и добавьте в него следующий 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',)
        verbose_name = 'Заказ'
        verbose_name_plural = 'Заказы'

    def __str__(self):
        return 'Order {}'.format(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')
    product = models.ForeignKey(Product, related_name='order_items')
    price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return '{}'.format(self.id)

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

```

Модель **Order** содержит несколько полей для сведений о клиенте и поле **paid**, которое по умолчанию имеет значение **False**. Позже мы будем использовать это поле для различения оплаченных и неоплаченных заказов. Мы также определяем метод **get_total_cost()**, чтобы получить общую стоимость товаров, купленных в этом заказе.

Модель **OrderItem** позволяет хранить продукт, количество и цену, уплаченную за каждый товар. Мы включаем **get_cost()** для возврата стоимости товара.

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

```
python manage.py makemigrations
```

Будут выведены следующие данные:

```

Migrations for 'orders':
  0001_initial.py:
    - Create model Order
    - Create model OrderItem

```

Для применения новой миграции выполните следующую команду:

```
python manage.py migrate
```

Теперь модели заказа синхронизируются с базой данных.

Включение моделей заказов на сайте администрирования

Давайте добавим модели заказов на сайт администрирования. Измените файл **admin.py** приложения **orders**:

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

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

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]
```

```
admin.site.register(Order, OrderAdmin)
```

Мы используем **ModelInline** для модели **OrderItem**, чтобы включить ее в качестве *inline* встроенного в класс **OrderAdmin**. Inline режим позволяет включить модель для отображения на той же странице редактирования, что и родительская модель.

Запустите сервер разработки командой `python manage.py runserver`, а затем откройте в браузере <http://127.0.0.1:8000/admin/orders/order/add/>. Появится следующая страница:

Django administration Welcome, admin. View site / Change password / Log out

Home > Orders > Orders > Add order

Add order

First name:

Last name:

Email:

Address:

Postal code:

City:

☐ Paid

Order items

Product	Price	Quantity	Delete?
<input type="text"/> 🔍	<input type="text"/>	<input type="text" value="1"/>	
<input type="text"/> 🔍	<input type="text"/>	<input type="text" value="1"/>	
<input type="text"/> 🔍	<input type="text"/>	<input type="text" value="1"/>	

[+ Add another Order item](#)

Save and add another

Save and continue editing

Save

Создание заказов клиентов

Нам нужно использовать только что созданные модели заказов для сохранения товаров, содержащихся в корзине для покупок, когда пользователь наконец пожелает разместить заказ. Функции создания нового заказа будут работать следующим образом:

1. Мы предоставляем форму заказа для заполнения пользовательских данных.
2. Создается новый экземпляр заказа с данными, введенными пользователями, а затем создается связанный экземпляр **OrderItem** для каждого товара в корзине.
3. Очищаем все содержимое корзины и перенаправляем пользователей на страницу success

Во-первых, нам нужна форма для ввода сведений о заказе. Создайте новый файл в каталоге приложения **orders** и назовите его **forms.py**. Добавьте в него следующий 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, которую мы собираемся использовать для создания новых объектов **Order**. Теперь нам нужно представление, чтобы обработать форму и создать новый заказ. Измените файл **views.py** приложения **orders** и добавьте в него следующий код:

```
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'])

            # очистка корзины
            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** мы получаем текущую корзину из сессии с **cart = Cart(request)**. В зависимости от метода запроса мы будем выполнять следующие задачи:

- **GET request** : Создается экземпляр формы **OrderCreateForm** и отображается шаблон **orders/order/create.html**
- **POST request** : Проверяет валидность введенных данных. Если данные являются допустимыми, то для создания нового экземпляра заказа будет использоваться **order = form.save()**. Затем мы сохраняем его в базу данных, а затем храним в переменной **order**. После создания заказа мы перейдем по товарам корзины и создадим **OrderItem** для каждого из них. Наконец, мы очищаем содержимое корзины

Теперь создайте новый файл в каталоге приложения **orders** и назовите его **urls.py**. Добавьте в него следующий код:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^create/$', views.order_create, name='order_create'),
]
```

Это шаблон URL-адреса для представления **order_create**. Отредактируйте файл **urls.py myshop** и включите следующий шаблон. Не забывайте, его следует разместить перед паттерном **shop.urls**:

```
url(r'^orders/', include('orders.urls', namespace='orders')),
```

Отредактируйте шаблон **cart/detail.html** приложения **cart** и замените эту строку:

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

На следующий код:

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

Теперь пользователи могут перейти от страницы **cart detail** к странице **order form**. Нам по-прежнему необходимо определить шаблоны для размещения заказов. Создайте следующую структуру файла в каталоге приложения **orders**:

```
templates/
  orders/
    order/
      create.html
      created.html
```

Отредактируйте шаблон **orders/order/create.html**:

```
{% 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 action="." method="post" class="order-form">
        {{ form.as_p }}
        <p><input type="submit" value="Place order"></p>
        {% csrf_token %}
    </form>
{% endblock %}

```

Этот шаблон отображает товары корзины, включая итоговую сумму, и форму для размещения заказа.

Отредактируйте шаблон **orders/order/created.html**:

```

{% 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 мы показываем при успешном создании заказа.

Запустите сервер веб-разработки. Откройте / в браузере <http://127.0.0.1:8000>, добавьте в корзину несколько товаров и нажмите на ссылку **checkout**.

Появится страница, подобная следующей:

Checkout

First name:

Antonio

Last name:

Melé

Email:

Address:

Postal code:

City:

Place order

Your order

- 2x Green tea \$60.00
- 1x Red tea \$45.50

Total: \$105.50

Заполните форму валидными данными и нажмите кнопку **Place order**. Заказ будет создан, и появится страница сообщающая об успешном выполнении действия:

My shop

Thank you

Your order has been successfully completed. Your order number is 1.

Выше была описана обязательная часть курсовой работы, далее будет описано дополнительное задание, которое необязательно для выполнения, но желательно.

Запуск асинхронных задач с Celery

Все, что выполняется в представлении, влияет на время отклика. Во многих случаях может понадобиться вернуть ответ пользователю как можно быстрее и позволить серверу выполнить некоторый процесс асинхронно. Это особенно актуально для трудоемких процессов или процессов, подверженных сбою, что может потребовать политики повторных попыток.

Например, платформа совместного использования видео позволяет пользователям загружать видео, но требует длительного времени для перекодирования загруженных видеороликов. Сайт может возвращать пользователю ответ, сообщать ему, что скоро начнется перекодирование, и начнется асинхронное перекодирование видео. Другим примером является отправка сообщений электронной почты пользователям. Если сайт отправляет уведомления по электронной почте от представления, то подключение SMTP может завершиться сбоем или замедлить ответ. Запуск асинхронных задач является необходимым для предотвращения блокирования выполнения.

Celery — это распределенная очередь задач, которая может обрабатывать большие объемы сообщений. Она выполняет обработку в реальном времени, но также поддерживает планирование задач. С помощью Celery можно не только легко создавать асинхронные задачи, но и выполнять их как можно быстрее, но можно также планировать их запуск в определенное время.

Вы можете ознакомиться с документацией к Celery здесь: <http://celery.readthedocs.org/en/latest/>

Установка Celery

Давайте установим Celery и интегрируем его в наш проект. Установить Celery через PIP можно с помощью следующей команды:

```
pip install celery==3.1.18
```

Celery требует message broker для обработки запросов из внешнего источника. Message broker заботится о направлении сообщений работникам Celery, который обрабатывает задачи по мере их получения. Давайте установим message broker.

Установка RabbitMQ

Существует несколько вариантов выбора в качестве message broker для Celery, включая магазины с key-value, такие как Redis или реальная система сообщений, например RabbitMQ. Мы настроили Celery с RabbitMQ, поскольку это рекомендуемый пакет для Celery.

Вы можете скачать RabbitMQ с официального сайта: <https://www.rabbitmq.com/download.html>

После установки запустите RabbitMQ с помощью следующей команды:

```
rabbitmq-server
```

Будет выведен результат, заканчивающийся следующей строкой:

```
Starting broker... completed with 10 plugins.
```

RabbitMQ работает и готов к приему сообщений.

Добавление Celery в проект

Необходимо сконфигурировать Celery. Создайте новый файл рядом с файлом **settings.py myshop** и назовите его **celery.py**. Этот файл будет содержать конфигурацию Celery для проекта. Добавьте в него следующий код:

```
import os
from celery import Celery
from django.conf import settings

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')

app = Celery('myshop')

app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

В этом коде мы установили переменную **DJANGO_SETTINGS_MODULE** для программы командной строки Celery. Затем создадим экземпляр нашего приложения с помощью **app = Celery('myshop')**. Мы загружаем любую настраиваемую конфигурацию из параметров проекта с помощью метода **config_from_object()**. Наконец, мы говорим Celery автоматически обнаруживать асинхронные задачи для приложений, перечисленных в параметрах **INSTALLED_APPS**. Celery будет искать файл **tasks.py** в каждом каталоге приложения для загрузки определенных в нем асинхронных задач.

Необходимо импортировать модуль **celery** в файл **__init__.py**, чтобы убедиться, что он загружается при запуске Django. Измените файл **myshop/__init__.py** и добавьте в него следующий код:

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

Теперь можно запускать программирование асинхронных задач для приложений.

Параметр **CELERY_ALWAYS_EAGER** позволяет выполнять задачи локально на синхронной основе, а не отправлять их в очередь. Это полезно для выполнения модульных тестов или проекта в локальной среде без запуска Celery.

Добавление асинхронных задач в приложение

Мы собираемся создать асинхронную задачу для отправки уведомления по электронной почте нашим пользователям при размещении заказа.

Конвенция говорит включить асинхронные задачи для приложения в модуль **tasks** в каталоге приложения. Создайте новый файл в приложении **orders** и назовите его **tasks.py**. Это место, где Celery будет искать асинхронные задачи. Добавьте в него следующий код:

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

@task
def order_created(order_id):
    """
```

Задача для отправки уведомления по электронной почте при успешном создании заказа.

```
"""
order = Order.objects.get(id=order_id)
subject = 'Order nr. {}'.format(order_id)
message = 'Dear {},\n\nYou have successfully placed an order.\n
          Your order id is {}'.format(order.first_name,
                                     order.id)

mail_sent = send_mail(subject,
                      message,
                      'admin@myshop.com',
                      [order.email])

return mail_sent
```

Мы определяем задачу **order_created** с помощью декоратора **task**. Как вы видите, Celery-это просто функция Python, декорированная **task**. Функция **task** получает параметр **order_id**. При выполнении задачи всегда рекомендуется передавать только IDS функциям задач и объектам поиска. Мы используем функцию **send_mail()**, предоставленную с помощью Django для отправки уведомления по электронной почте пользователю, который разместил заказ. Если вы не хотите настраивать параметры электронной почты, можно сказать Django записывать электронные сообщения в консоль, добавив в файл **settings.py** следующий параметр:

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

Асинхронные задачи используются не только для процессов, требующих длительного времени выполнения, но и для других процессов, подверженных сбою, которые не занимают много времени для выполнения, но могут быть связаны с ошибками подключения или требуют политики повторных попыток.

Теперь мы должны добавить задачу к нашему **order_create view**. Откройте файл **views.py** приложения **orders** и импортируйте task следующим образом:

```
from .tasks import order_created
```

Затем вызовите **order_created** асинхронную задачу после корзины следующим образом:

```
# очистка корзины
cart.clear()
# запуск асинхронной задачи
order_created.delay(order.id)
```

Мы называем метод **delay()** задачи, чтобы выполнить ее асинхронно. Задача будет добавлена в очередь и будет выполнена как можно скорее.

Откройте еще один терминал и запустите **celery**, используя следующую команду:

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

Теперь Celery worker работает и готов к обработке задач. Убедитесь, что сервер разработки Django также работает. Откройте <http://127.0.0.1:8000/> в браузере, добавьте несколько товаров в корзину для покупок и завершите заказ. В терминале, в котором вы запустили Celery worker, вы увидите результат, аналогичный этому:

```
[2015-09-14 19:43:47,526: INFO/MainProcess] Received task: orders.
tasks.order_created[933e383c-095e-4cbd-b909-70c07e6a2ddf]
[2015-09-14 19:43:50,851: INFO/MainProcess] Task orders.tasks.
order_created[933e383c-095e-4cbd-b909-70c07e6a2ddf] succeeded in
```

3.318835098994896s: 1

Задача выполнена, и вы получите уведомление по электронной почте о своем заказе.

Celery мониторинг

Возможно, потребуется отслеживать выполняемые асинхронные задачи. Flower — это веб-инструмент для наблюдения за Celery. Можно установить Flower с помощью команды:

```
pip install flower
```

После установки можно запустить Flower, выполнив следующую команду из каталога проекта:

```
celery -A myshop flower
```

Откройте в браузере <http://localhost:5555/dashboard>. Вы сможете увидеть активные Celery workers и статистику асинхронных задач:

Celery Flower

DashboardTasksBrokerMonitorDocsCode

Active: 0

Processed: 1

Failed: 0

Succeeded: 1

Retried: 0

Shut Down

	Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
<div><div></div></div>	celery@MacBook-Air-de-Antonio.local	Online	0	1	0	1	0	2.95, 3.64, 3.28

Вы можете ознакомиться с документацией Flower здесь: <http://flower.readthedocs.org/en/latest/>

Экспорт заказов в CSV-файлы

Иногда может понадобиться экспортировать данные, содержащиеся в модели, в файл, чтобы можно было импортировать его в любую другую систему. Одним из наиболее распространенных форматов для экспорта/импорта данных является **Comma-Separated Values (CSV)**. CSV-файл — это обычный текстовый файл, состоящий из нескольких записей. Обычно существует одна запись в строке и разделитель, обычно это литеральная запятая, она отделяет поля записей друг от друга. Мы собираемся настроить сайт администрирования, чтобы он мог экспортировать заказы в CSV-файлы.

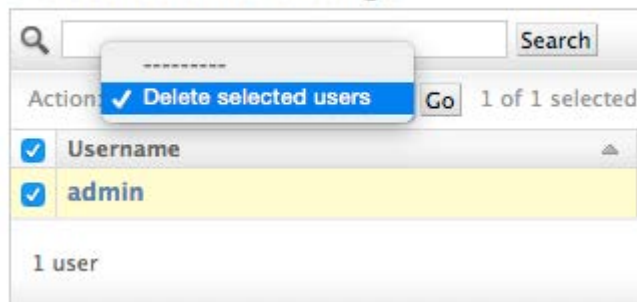
Добавление настраиваемых действий на сайт администрирования

Джанго предлагает широкий выбор параметров для настройки сайта администрирования. Мы собираемся изменить представление списка объектов, включив в него пользовательское действие администратора.

Действие администратора работает следующим образом: пользователь выбирает объекты на странице списка объектов с чекбоксами, затем выбирает

действие для выполнения всех выбранных элементов и выполняет его. На следующем снимке экрана показаны действия, расположенные на сайте администрирования:

Select user to change



Создавайте настраиваемые действия администратора, чтобы позволить сотрудникам применять действия к нескольким элементам одновременно.

Можно создать настраиваемое действие, записывая обычную функцию, которая получает следующие параметры:

- Текущий **ModelAdmin**
- Текущий объект запроса как экземпляр **HttpRequest**
- **QuerySet** для объектов, выбранных пользователем

Эта функция будет выполняться, при выполнении действия с сайта администрирования.

Мы собираемся создать настраиваемое действие администратора для загрузки списка заказов в виде CSV-файла. Измените файл **admin.py** приложения **orders** и добавьте следующий код перед классом **OrderAdmin**:

```
import csv
import datetime
from django.http import HttpResponse

def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment;
filename={}.csv'.format(opts.verbose_name)
    writer = csv.writer(response)
    fields = [field for field in opts.get_fields() if not field.many_to_many and
not field.one_to_many]
    # Write a first row with header information
    writer.writerow([field.verbose_name for field in fields])
    # Write data rows
    for obj in queryset:
        data_row = []
        for field in fields:
            value = getattr(obj, field.name)
            if isinstance(value, datetime.datetime):
                value = value.strftime('%d/%m/%Y')
            data_row.append(value)
        writer.writerow(data_row)
```



```
return response
export_to_csv.short_description = 'Export to CSV'
```

В этом коде выполняются следующие задачи:

- Мы создаем экземпляр **HttpResponse**, включающий кастомный **text/csv**-тип контента, чтобы сообщить браузеру, что ответ должен обрабатываться как файл CSV. Также добавляется заголовок **Content-Disposition**, указывающий, что HTTP-ответ содержит вложенный файл.
- Мы создаем объект **CSV writer**, который будет записывать в объект **response**.
- Поля **model** получаются динамически с помощью метода **get_fields()** параметров модели **_meta**. Мы исключаем связи "многие ко многим" и "один ко многим".
- Мы пишем строку заголовка, включая имена полей.
- Мы переходим по заданному запросу и записываем строку для каждого объекта, возвращаемого запросом. Мы осторожно форматируем объекты **datetime**, поскольку выходное значение для CSV должно быть строкой.
- Мы настраиваем отображаемое имя для действия в шаблоне, установив атрибут **short_description** для нашей функции.

Мы создали универсальное действие, которое можно добавить к любому классу **ModelAdmin**.

Наконец, добавьте новое действие **export_to_csv** admin к классу **OrderAdmin** следующим образом:

```
class OrderAdmin(admin.ModelAdmin):
    # ...
    actions = [export_to_csv]
```

Откройте в браузере <http://127.0.0.1:8000/admin/orders/order/>. Результирующее действие администратора должно выглядеть следующим образом:

Django administration

Home > Orders > Orders

Select order to change

Action: Export to CSV Go 2 of 11 selected

<input type="checkbox"/>	ID	First name	Last name	Email	Address
<input checked="" type="checkbox"/>	11	Antonio	Melé	antonio.mele@gmail.com	Bank Street
<input checked="" type="checkbox"/>	10	Antonio	Melé	antonio.mele@gmail.com	Bank Street
<input type="checkbox"/>	9	Antonio	Mele	antonio.mele@gmail.com	Bank Street
<input type="checkbox"/>	8	Antonio	Melé	antonio.mele@gmail.com	Bank Street

Отметьте некоторые заказы и выберите действие **Export to CSV** из поля **select**, затем нажмите кнопку **Go**. В браузере будет загружен созданный CSV-файл с именем **order.csv**. Откройте загруженный файл с помощью текстового редактора. Содержимое может быть приведено в следующем формате, включая строку заголовка и строку для каждого выбранного объекта **Order**:

```
ID,first name,last name,email,address,postal
code,city,created,updated,paid
3,Antonio,Melé,antonio.mele@gmail.com,Bank Street 33,WS J11,London,25/
05/2015,25/05/2015,False
...
```

Как вы можете видеть, создание действий администратора довольно просто.

Расширение сайта администрирования с кастомными представлениями

Иногда может понадобиться настроить сайт администрирования сверх того, что возможно с помощью конфигурации **ModelAdmin**, создания действий администратора и переопределения шаблонов администратора. В этом случае необходимо создать настраиваемое административное представление. С помощью кастомного представления можно создавать любые необходимые функциональные возможности.

Создадим кастомное представление для отображения сведений о заказе. Измените файл **views.py** приложения **orders** и добавьте в него следующий код:

```
from django.contrib.admin.views.decorators import staff_member_required
from django.shortcuts import get_object_or_404
from .models import Order

@staff_member_required
def admin_order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request,
                  'admin/orders/order/detail.html',
                  {'order': order})
```

Декоратор **staff_member_required** проверяет, что как **is_active**, так и **is_staff** поля пользователя, запрашивающего страницу, имеют значение **True**. В этом представлении можно получить объект **Order** с заданным идентификатором и визуализировать шаблон для отображения заказа.

Теперь отредактируйте файл **urls.py** приложения **orders** и добавьте к нему следующий шаблон URL-адреса:

```
url(r'^admin/order/(?P<order_id>\d+)/$', views.admin_order_detail,
    name='admin_order_detail'),
```

Создайте следующую структуру файла в каталоге **templates/** приложения **orders**:

```
admin/
  orders/
    order/
      detail.html
```

Измените шаблон **detail.html** и добавьте в него следующее содержимое:

```
{% extends "admin/base_site.html" %}
{% load static %}
{% block extrastyle %}
    <link rel="stylesheet" type="text/css" href="{% static "css/admin.css" %}" />
```

```

{% endblock %}
{% block title %}
    Order {{ order.id }} {{ block.super }}
{% endblock %}
{% block breadcrumbs %}
    <div class="breadcrumbs">
        <a href="{% url 'admin:index' %}">Home</a> &rsaquo;
        <a href="{% url 'admin:orders_order_changelist' %}">Orders</a>
        &rsaquo;
        <a href="{% url 'admin:orders_order_change' order.id %}">Order {{ order.id }}</a>
        &rsaquo; Detail
    </div>
{% endblock %}
{% block content %}
    <h1>Order {{ order.id }}</h1>
    <ul class="object-tools">
        <li>
            <a href="#" onclick="window.print();">Print order</a>
        </li>
    </ul>
    <table>
        <tr>
            <th>Created</th>
            <td>{{ order.created }}</td>
        </tr>
        <tr>
            <th>Customer</th>
            <td>{{ order.first_name }} {{ order.last_name }}</td>
        </tr>
        <tr>
            <th>E-mail</th>
            <td><a href="mailto:{{ order.email }}">{{ order.email }}</a></td>
        </tr>
        <tr>
            <th>Address</th>
            <td>{{ order.address }}, {{ order.postal_code }} {{ order.city }}</td>
        </tr>
        <tr>
            <th>Total amount</th>
            <td>${{ order.get_total_cost }}</td>
        </tr>
        <tr>
            <th>Status</th>
            <td>{% if order.paid %}Paid{% else %}Pending payment{% endif %}</td>
        </tr>
    </table>
    <div class="module">
        <div class="tabular inline-related last-related">
            <table>
                <h2>Items bought</h2>
                <thead>
                    <tr>
                        <th>Product</th>
                        <th>Price</th>
                        <th>Quantity</th>
                        <th>Total</th>
                    </tr>
                </thead>
                <tbody>
                    {% for item in order.items.all %}
                        <tr class="row{% cycle "1" "2" %}">

```

```

        <td>{{ item.product.name }}</td>
        <td class="num">${{ item.price }}</td>
        <td class="num">{{ item.quantity }}</td>
        <td class="num">${{ item.get_cost }}</td>
    </tr>
    {% endfor %}
    <tr class="total">
        <td colspan="3">Total</td>
        <td class="num">${{ order.get_total_cost }}</td>
    </tr>
</tbody>
</table>
</div>
</div>
{% endblock %}

```

Это шаблон для отображения сведений о заказе на сайте администрирования. Он расширяет шаблон **admin/base_site.html** сайта администрирования Django, который содержит основную структуру HTML и стили CSS для администратора. Мы загружаем пользовательский статический файл **css/admin.css**.

Создание системы купонов

Многие интернет-магазины выдают купоны клиентам, которые могут быть погашены за счет скидок на покупки. Интерактивный купон обычно состоит из кода, предоставляемого пользователям, который действителен в период определенного времени. Купон может быть погашен один или несколько раз.

Мы собираемся создать купонную систему для нашего магазина. Купоны не будут иметь каких-либо ограничений в отношении количества времени, в течение которого они могут быть погашены, и будут применяться к общей сумме корзины для покупок. Для этой функциональности необходимо создать модель для хранения кода купона, допустимых временных рамок и применяемой скидки.

Создайте новое приложение в проекте **myshop**, используя следующую команду:

```
python manage.py startapp coupons
```

Отредактируйте файл **settings.py** проекта **myshop** и добавьте приложение в настройки **INSTALLED_APPS**:

```

INSTALLED_APPS = (
    # ...
    'coupons',
)

```

Новое приложение теперь активно в нашем проекте.

Построение моделей купонов

Начнем с создания модели **Coupon**. Измените файл **models.py** приложения **coupons** и добавьте в него следующий код:

```

from django.db import models
from django.core.validators import MinValueValidator, MaxValueValidator

```

```
class Coupon(models.Model):
    code = models.CharField(max_length=50, unique=True)
    valid_from = models.DateTimeField()
    valid_to = models.DateTimeField()
    discount = models.IntegerField(validators=[MinValueValidator(0),
MaxValueValidator(100)])
    active = models.BooleanField()

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

Это модель, которую мы собираемся использовать для хранения купонов. Модель **Coupon** содержит следующие поля:

- **code** : Код, который пользователи должны ввести для применения купона к покупке.
- **valid_from** : Значение datetime, указывающее, когда купон становится действительным.
- **valid_to** : Значение datetime, указывающее, когда купон становится недействительным.
- **discount** : Применяемая ставка дисконта (это процент, поэтому она принимает значения от 0 до 100). Средства проверки для этого поля используются для ограничения минимальных и максимальных допустимых значений.
- **active** : Логическое значение, указывающее, активен ли купон.

Выполните следующую команду, чтобы создать начальную миграцию для приложения купонов:

```
python manage.py makemigrations
```

Выходные данные должны включать следующие строки:

```
Migrations for 'coupons':
  0001_initial.py:
    - Create model Coupon
```

Затем мы выполняем следующую команду для применения миграции:

```
python manage.py migrate
```

Следует увидеть выходные данные, включающие следующую строку:

```
Applying coupons.0001_initial... OK
```

Миграция теперь применена в базе данных. Давайте добавим модель **Coupon** на сайт администрирования. Измените файл **admin.py** приложения **coupons** и добавьте в него следующий код:

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

class CouponAdmin(admin.ModelAdmin):
    list_display = ['code', 'valid_from', 'valid_to', 'discount', 'active']
    list_filter = ['active', 'valid_from', 'valid_to']
    search_fields = ['code']
admin.site.register(Coupon, CouponAdmin)
```

Модель **Coupon** теперь зарегистрирована на сайте администрирования. Откройте в браузере <http://127.0.0.1:8000/admin/coupons/coupon/add/>. Вы увидите следующую форму:

The screenshot shows the Django administration interface for adding a new coupon. The page title is "Add coupon". The form includes the following fields:

- Code:** A text input field.
- Valid from:** A section containing a "Date" field with a "Today" button and a calendar icon, and a "Time" field with a "Now" button and a clock icon. Below these fields is a note: "Note: You are 2 hours ahead of server time."
- Valid to:** A section containing a "Date" field with a "Today" button and a calendar icon, and a "Time" field with a "Now" button and a clock icon. Below these fields is a note: "Note: You are 2 hours ahead of server time."
- Discount:** A text input field.
- Active:** A checkbox that is currently checked.

At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "Save".

Заполните форму для создания нового купона, который действителен для текущей даты, убедитесь, что флажок **Active** установлен, и нажмите кнопку Сохранить.

Применение купона к корзине

Мы можем хранить новые купоны и создавать запросы для получения существующих купонов. Теперь нам нужен способ, с помощью которого клиенты могли бы применять купоны к своим покупкам. Подумайте о том, как будет работать эта функция. Способ применения купона будет следующим:

1. Пользователь добавляет товары в корзину для покупок.
2. Пользователь может ввести код купона в форме, отображаемой на странице корзины для покупок.
3. Когда пользователь вводит код купона и отправляет форму, мы ищем существующий купон с заданным кодом, который в настоящее время действителен. Мы должны проверить, что код купона совпадает с введенным пользователем, атрибут **active** имеет значение True, а текущее значение datetime находится между значениями **valid_from** и **valid_to**.
4. Если купон найден, мы его сохраняем в сессии пользователя и выводим на экран корзину, включая применяемую к ней скидку, и обновленную общую сумму.
5. Когда пользователь размещает заказ, мы сохраняем купон в этот заказ.

Создайте новый файл в каталоге приложения **coupons** и назовите его **forms.py**. Добавьте в него следующий код:

```
from django import forms
```

```
class CouponApplyForm(forms.Form):  
    code = forms.CharField()
```

Это форма, которая будет использоваться пользователем для ввода кода купона. Измените файл **views.py** в приложении **coupons** и добавьте в него следующий код:

```
from django.shortcuts import render, redirect  
from django.views.decorators.http import require_POST  
from django.utils import timezone  
from .models import Coupon  
from .forms import CouponApplyForm  
  
@require_POST  
def coupon_apply(request):  
    now = timezone.now()  
    form = CouponApplyForm(request.POST)  
    if form.is_valid():  
        code = form.cleaned_data['code']  
        try:  
            coupon = Coupon.objects.get(code__iexact=code,  
                                         valid_from__lte=now,  
                                         valid_to__gte=now,  
                                         active=True)  
            request.session['coupon_id'] = coupon.id  
        except Coupon.DoesNotExist:  
            request.session['coupon_id'] = None  
    return redirect('cart:cart_detail')
```

Представление **coupon_apply** проверяет купон и сохраняет его в сессии пользователя. Мы применяем декоратор **require_POST** к этому представлению, чтобы ограничить его учетом запросов. В представлении мы выполняем следующие задачи:

1. Мы создаем экземпляр формы **CouponApplyForm**, используя учтенные данные, и проверяем, что форма является валидной.
2. Если форма является валидной, мы получим код, введенный пользователем из формы **cleaned_data**. Мы пытаемся извлечь объект **Coupon** с данным кодом. Мы используем поиск в поле **iexact**, чтобы проверить точное совпадение без учета регистра. Купон должен быть активен в данный момент (**active=True**) и действителен для текущего **datetime**. Мы используем функцию Django **timezone.now()**, чтобы получить текущую дату и время, сопоставленные с часовым поясом, и сравнить ее с полями **valid_from** и **valid_to**, выполняющими **lte**(меньше или равными) и **gte**(больше или равным).
3. Идентификатор купона хранится в сессии пользователя.
4. Мы перенаправим пользователя на URL-адрес **cart_detail**, чтобы отобразить корзину с примененным купоном.

Нам нужен шаблон URL-адреса для представления **coupon_apply**. Создайте новый файл в каталоге приложения **coupons** и назовите его **urls.py**. Добавьте в него следующий код:

```
from django.conf.urls import url
```



```
from . import views
```

```
urlpatterns = [  
    url(r'^apply/$', views.coupon_apply, name='apply'),  
]
```

Затем отредактируйте основной **urls.py** проекта **myshop** и включите шаблоны URL-адресов купонов следующим образом:

```
url(r'^coupons/', include('coupons.urls', namespace='coupons')),
```

Не забудьте поместить этот шаблон перед шаблоном **shop.urls**.

Теперь отредактируйте файл **cart.py** приложения **cart**. Включите следующий импорт:

```
from coupons.models import Coupon
```

Добавьте следующий код в конец метода **__init__()** класса **Cart** для инициализации купона из текущей сессии:

```
# сохранение текущего примененного купона  
self.coupon_id = self.session.get('coupon_id')
```

В этом коде мы пытаемся получить session key **coupon_id** из текущей сессии и сохранить его значение в объекте **Cart**. Добавьте в объект **Cart** следующие методы:

```
@property  
def coupon(self):  
    if self.coupon_id:  
        return Coupon.objects.get(id=self.coupon_id)  
    return None  
  
def get_discount(self):  
    if self.coupon:  
        return (self.coupon.discount / Decimal('100')) * self.get_total_price()  
    return Decimal('0')  
  
def get_total_price_after_discount(self):  
    return self.get_total_price() - self.get_discount()
```

Вот что делают эти методы:

- **coupon()** : Этот метод определяется как **property**. Если корзина содержит функцию **coupon_id**, возвращается объект **Coupon** с заданным id.
- **get_discount()** : Если корзина содержит купон, мы получаем скидку по ставке и возвращаем сумму, которая будет вычтена из общей суммы корзины.
- **get_total_price_after_discount()** : Мы возвращаем общую сумму корзины после вычета суммы, возвращенной методом **get_discount()**.

Теперь класс **Cart** готов обработать купон, примененный к текущей сессии, и применить соответствующую скидку.

Давайте включим систему купонов в detail view корзины. Отредактируйте файл **views.py** приложения **cart** и добавьте следующий импорт в верхнюю часть файла:

```
from coupons.forms import CouponApplyForm
```

Далее, отредактируйте представление **cart_detail** и добавьте в него новую форму следующим образом:

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

Измените шаблон **cart/detail.html** приложения **cart** и найдите следующие строки:

```
<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price }}</td>
</tr>
```

Замените их следующими:

```
{% if cart.coupon %}
    <tr class="subtotal">
        <td>Subtotal</td>
        <td colspan="4"></td>
        <td class="num">${{ cart.get_total_price }}</td>
    </tr>
    <tr>
        <td>
            "{{{ cart.coupon.code }}" coupon
            ({{{ cart.coupon.discount }}}% off)
        </td>
        <td colspan="4"></td>
        <td class="num neg">
            - ${{ cart.get_discount|floatformat:"2" }}
        </td>
    </tr>
{% endif %}
<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">
        ${{ cart.get_total_price_after_discount|floatformat:"2" }}
    </td>
</tr>
```

Это код для отображения дополнительного купона и его скидки. Если корзина содержит купон, мы выводим первую строку, включая общую сумму корзины в качестве промежуточного итога. Затем мы используем вторую строку для отображения текущего купона, примененного к корзине. Наконец, мы отображаем общую цену, включая скидки, вызвав метод **get_total_price_after_discount()** объекта **cart**.

В том же файле следует включить следующий код после тега **</table>**:


```
<form action="{% url 'coupons:apply' %}" method="post">
    {{ coupon_apply_form }}
    <input type="submit" value="Apply">
```

```
{% csrf_token %}
</form>
```

Будет отображена форма для ввода кода купона и применения его к текущей корзине.

Откройте в браузере <http://127.0.0.1:8000/>, добавьте товар в корзину и примените купон, созданный с помощью ввода его кода в форму. В корзине отобразится скидка по купону:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	2 <input type="button" value="Update"/>	Remove	\$21.20	\$42.40
Subtotal					\$42.40
"SUMMER" coupon (10% off)					- \$4.24
Total					\$38.16

Apply a coupon:

Code:

[Continue shopping](#)

[Checkout](#)

Давайте добавим купон к следующему шагу процесса покупки. Измените шаблон **orders/order/create.html** приложения **orders** и найдите следующие строки:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price }}</span>
    </li>
  {% endfor %}
</ul>
```

Замените их следующим кодом:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price }}</span>
    </li>
  {% endfor %}
  {% if cart.coupon %}
    <li>
      "{{ cart.coupon.code }}" ({{ cart.coupon.discount }}% off)
      <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
    </li>
  {% endif %}
</ul>
```

```
</li>
{% endif %}
</ul>
```

Заказ должен теперь включать примененный купон, если он есть. Теперь найдите следующую строку:

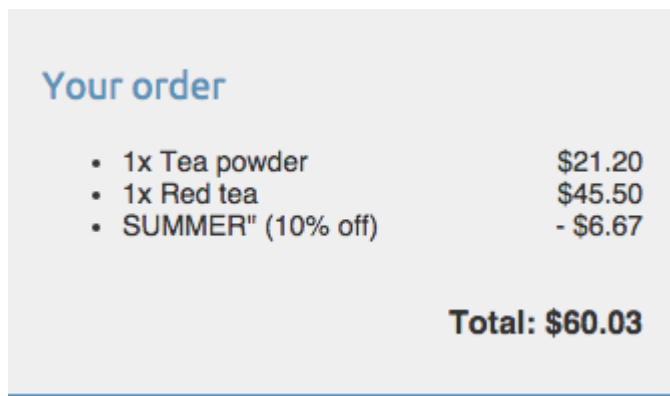
```
<p>Total: ${{ cart.get_total_price }}</p>
```

Замените ее на следующий код:

```
<p>Total: ${{ cart.get_total_price_after_discount|floatformat:"2" }}</p>
```

При этом общая цена будет также рассчитываться путем применения скидки купона.

Откройте в браузере <http://127.0.0.1:8000/orders/create/>. Вы увидите, что заказ примененный купон:



Теперь пользователи могут применять купоны к своим покупкам.

Применение купонов к заказам

Мы собираемся сохранять купон, который был применен для каждого заказа. Во-первых, необходимо изменить модель заказа, чтобы сохранить связанный с ней объект купона, если таковой имеется.

Измените файл **models.py** приложения **orders** и добавьте в него следующие импортируемые компоненты:

```
from decimal import Decimal
from django.core.validators import MinValueValidator, MaxValueValidator
from coupons.models import Coupon
```

Затем добавьте в модель **Order** следующие поля:

```
coupon = models.ForeignKey(Coupon,
                           related_name='orders',
                           null=True,
                           blank=True)
discount = models.IntegerField(default=0,
                               validators=[MinValueValidator(0),
                                           MaxValueValidator(100)])
```

Эти поля позволят сохранить дополнительный купон, примененный к заказу, и скидку, примененную купоном. Скидка хранится в соответствующем объекте купона, но мы включаем ее в модель заказа, чтобы сохранить ее при изменении или удалении купона.

После изменения модели заказа необходимо создать миграцию. Выполните следующую команду из командной строки:

```
python manage.py makemigrations
```

На выходе вы увидите следующие строки:

```
Migrations for 'orders':
  0002_auto_20150606_1735.py:
    - Add field coupon to order
    - Add field discount to order
```

Примените новую миграцию следующей командой:

```
python manage.py migrate orders
```

Вы получите подтверждение применения новой миграции. Изменения полей модели заказа теперь синхронизируются с базой данных.

Вернитесь в файл **models.py** и измените метод **get_total_cost()** модели **Order**:

```
def get_total_cost(self):
    total_cost = sum(item.get_cost() for item in self.items.all())
    return total_cost - total_cost * (self.discount / Decimal('100'))
```

В методе **get_total_cost()** модели **Order** теперь учитывается примененная скидка, если таковая имеется.

Измените файл **views.py** приложения **orders** и измените представление **order_create**, чтобы сохранить связанный купон и его скидку при создании нового заказа. Найдите следующую строку:

```
order = form.save()
```

Замените ее следующим кодом:



```
order = form.save(commit=False)
if cart.coupon:
    order.coupon = cart.coupon
    order.discount = cart.coupon.discount
order.save()
```

В новом коде мы создаем объект **Order** с помощью метода **save()** формы **OrderCreateForm**. Мы не можем сохранить его в базе данных с помощью команды **commit=False**. Если корзина содержит купон, мы сохраняем связанный купон и применяемую скидку. Затем мы сохраняем объект **order** в базу данных.



Запустите Ngrok из терминала, используя следующую команду:

```
./ngrok http 8000
```

Откройте URL-адрес, предоставленный Ngrok в браузере, и завершите покупку с помощью созданного купона. По завершении успешной покупки можно перейти к <http://127.0.0.1:8000/admin/orders/order/> и проверить, что заказ содержит купон и примененную скидку:

Coupon: SUMMER  

Discount:

Order Items	
Product	Price
52 <input type="text" value="2"/>  Red tea	<input type="text" value="45,50"/>
53 <input type="text" value="3"/>  Tea powder	<input type="text" value="21,20"/>

Добавление интернационализации и локализации

Джанго обеспечивает полную поддержку интернационализации и локализации. Она позволяет перевести приложение на несколько языков и обрабатывает форматирование дат, времени, чисел и часовых поясов, зависящее от языка. Давайте проясним разницу между интернационализацией и локализацией. Интернационализации (часто сокращаются до **i18n**) — это процесс адаптации программного обеспечения для потенциального использования различных языков и языковых стандартов, с тем чтобы он не был жестко подключен к определенному языку или языку. Локализация (сокращенная до **l10n**) — это процесс реального перевода программного обеспечения и его адаптации к определенному языку. Сам Джанго переводится более чем на 50 языков, используя свои рамки интернационализации.

Интернационализация с Джанго

internationalization framework позволяет легко помечать строки для перевода как в коде Python, так и в шаблонах. Он опирается на набор инструментов GNU gettext, предназначенный для создания и управления файлами сообщений. Файл сообщений представляет собой обычный текстовый файл, представляющий язык. Он содержит часть или все строки перевода, найденные в приложении, и их соответствующие переводы для одного языка. Файлы сообщений имеют расширение **.po**.

После перевода файлы сообщений компилируются, чтобы обеспечить быстрый доступ к переведенным строкам. Скомпилированные файлы переводов имеют расширение **.mo**.

Параметры интернационализации и локализации

Джанго предоставляет несколько параметров интернационализации. Наиболее актуальными являются следующие параметры:

- **USE_I18N** : Логическое значение, указывающее, включена ли система перевода Джанго. По умолчанию это значение **True**.
- **USE_L10N** : Логическое значение, указывающее, включено ли локализованное форматирование. Когда активно, локализованные

форматы используются для представления дат и чисел. По умолчанию это значение **False**.

- **USE_TZ** : Логическое значение, указывающее, являются ли **datetimes** известными часовыми поясами. При создании проекта с командой **startproject** это значение равно **True**.
- **LANGUAGE_CODE** : Код языка по умолчанию для проекта. Идет в стандартном формате идентификатора языка, например "en-us" для американского английского или "en-gb" для британского английского. Для выполнения этого параметра необходимо, чтобы **USE_I18N** был установлен в значение **True**. Список допустимых кодов языка можно найти тут: <http://www.i18nguy.com/unicode/language-identifiers.html> .
- **LANGUAGES** : Кортеж, содержащий доступные языки для проекта. Они приходят в два кортежа **language code** и **language name**. Список доступных языков можно просмотреть в **django.conf.global_settings**. При выборе языков, на которых будет доступен сайт, можно задать **LANGUAGES** для подмножества этого списка.
- **LOCALE_PATHS** : Список каталогов, в которых Джанго ищет файлы сообщений, содержащие переводы для данного проекта.
- **TIME_ZONE** : Строка, представляющая часовой пояс проекта. При создании нового проекта с помощью команды **startproject**, по умолчанию устанавливается значение "UTC". Можно установить его в любой другой часовой пояс, например 'Europe/Madrid'.

Это некоторые из доступных параметров интернационализации и локализации. Полный список можно найти здесь: <https://docs.djangoproject.com/en/1.8/ref/settings/#globalization-i18n-l10n>

Команды управления интернационализации

Джанго включает следующие команды для управления переводом с помощью **manage.py** или **django-admin**:

- **makemessages** – Выполняется над деревом исходного кода, чтобы найти все строки, помеченные для перевода, и создать или обновить файлы сообщений **.po** в каталоге языков. Для каждого языка создается один файл **.po**.
- **compilemessages** – Выполняется компиляция существующих файлов сообщений **.po** в файлы **.mo**, которые используются для извлечения переводов.

Для создания, обновления и компиляции файлов сообщений потребуется набор инструментов **gettext**. Большинство дистрибутивов Linux включают в себя **gettext**. При использовании Mac OS X, возможно, самый простой способ установить его — через Homebrew в <http://brew.sh/> с помощью команды `install gettext`. Также может понадобиться принудительно связать его с командой `brew link gettext --force`. Для Windows выполните следующие шаги <https://docs.djangoproject.com/en/1.8/topics/i18n/translation/#gettext-on-windows>

Как добавить переводы в проект Django

Дjango поставляется с программным обеспечением, определяющим текущий язык на основе данных request. Это **LocaleMiddleware** промежуточное программное обеспечение, которое находится

в **django.middleware.locale.LocaleMiddleware** выполняет следующие задачи:

1. Если используется **i18n_patterns**, то есть, вы используете преобразованные шаблоны URL-адресов, он ищет префикс языка в запрошенном URL-адресе для определения текущего языка.
2. Если префикс языка не найден, он ищет существующий **LANGUAGE_SESSION_KEY** в сессии текущего пользователя.
3. Если язык не задан в сессии, он ищет существующий объект cookie с текущим языком. Пользовательское имя этого файла cookie может быть предоставлено в параметре **LANGUAGE_COOKIE_NAME**. По умолчанию имя этого файла cookie - **django_language**.
4. Если файл cookie не найден, он ищет HTTP-заголовок запроса **Accept-Language**.
5. Если в заголовке **Accept-Language** не указан язык, Django использует язык, определенный в параметре **LANGUAGE_CODE**.

По умолчанию Django будет использовать язык, определенный в параметре **LANGUAGE_CODE**, если не используется **LocaleMiddleware**. Описанный выше процесс применяется только при использовании данного промежуточного ПО.

Подготовка проекта к интернационализации

Давайте подготовим проект к использованию различных языков. Мы собираемся создать английский и испанский вариант для нашего магазина. Отредактируйте файл **settings.py** вашего проекта и добавьте в него следующие настройки **LANGUAGES**. Поместите их рядом с параметром **LANGUAGE_CODE**:

```
LANGUAGES = (  
    ('en', 'English'),  
    ('es', 'Spanish'),  
)
```

Параметр **LANGUAGES** содержит два кортежа, состоящие из кода языка и имени. Коды языков могут быть специфичными для конкретного языка, например **en-us** или **en-gb** или универсальными, например **en**. С помощью этого параметра мы указываем, что наше приложение будет доступно только на английском и испанском языках. Если не определить пользовательские языки, сайт будет доступен на всех языках, на которых будет переведен Django.

Отредактируйте **LANGUAGE_CODE** настройку следующим образом:

```
LANGUAGE_CODE = 'en'
```

Добавьте **django.middleware.locale.LocaleMiddleware** в настройки **MIDDLEWARE_CLASSES**. Убедитесь, что эта строка вставлена после **SessionMiddleware**, потому что **LocaleMiddleware** необходимо

использовать данные сессии. Она также должна быть помещена перед **CommonMiddleware**, поскольку для разрешения запрошенного URL-адреса требуется активный язык.

Параметры **MIDDLEWARE_CLASSES** должны выглядеть следующим образом:

```
MIDDLEWARE_CLASSES = (  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.locale.LocaleMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    # ...  
)
```

*Порядок промежуточного программного обеспечения очень важен, так как каждое промежуточное программное обеспечение может зависеть от данных, которые были заданы другим программным обеспечением, выполненным ранее. Промежуточное программное обеспечение применяется для запросов в порядке их появления в **MIDDLEWARE_CLASSES** и в обратном порядке для ответов.*

Создайте следующую структуру каталогов в главном каталоге проекта рядом с файлом **manage.py**:

```
locale/  
    en/  
    es/
```

Каталог **locale** — это место, где будут находиться файлы сообщений для приложения. Отредактируйте файл **settings.py** и добавьте в него следующий параметр:

```
LOCALE_PATHS = (  
    os.path.join(BASE_DIR, 'locale/'),  
)
```

Параметр **LOCALE_PATHS** указывает каталоги, в которых Django должен искать файлы перевода. Пути языка, которые появляются первыми, имеют наивысший приоритет.

При использовании команды **makemessages** из каталога проекта файлы сообщений будут создаваться в каталогах созданных внутри директории **locale/**. Однако для приложений, содержащих язык и региональные стандарты, в этом каталоге будут созданы файлы сообщений.

Перевод литералов в коде Python

Чтобы перевести литералы в коде Python, можно пометить строки для перевода с помощью функции **gettext()**, включенной в **django.utils.translation**. Эта функция преобразует сообщение и возвращает строку. Конвенция должна импортировать эту функцию как более короткий псевдоним с именем **_** (символ подчеркивания).

Вы можете найти всю документацию о переводе здесь: <https://docs.djangoproject.com/en/1.8/topics/i18n/translation/>

Стандартные переводы

В следующем коде показано, как пометить строку для перевода:

```
from django.utils.translation import gettext as _
output = _('Text to be translated.')
```

Lazy переводы

Джанго включает **lazy** версии для всех функций перевода, которые имеют суффикс **_lazy()**. При использовании **lazy** функций строки переводятся при доступе к значению, а не при вызове функции (поэтому они переводятся **lazily**). Функции **lazy** перевода удобны, если строки, помеченные для перевода, находятся в путях, которые выполняются при загрузке модулей.

*При использовании **gettext_lazy()** вместо **gettext()** строки переводятся при доступе к значению, а не при вызове функции. Джанго предлагает **lazy** версию для всех функций перевода.*

Перевод с помощью переменных

Строки, помеченные для перевода, могут содержать placeholders для включения переменных в переводы. Следующий код является примером строки перевода с placeholder:

```
from django.utils.translation import gettext as _
month = _('April')
day = '14'
output = _('Today is %(month)s %(day)s') % {'month': month, 'day': day}
```

Используя placeholder, можно переупорядочить текстовые переменные. Например, перевод на английский язык для предыдущего примера может быть "Today is April 14", а испанский - "Hoy es 14 de Abril". Всегда используйте интерполяцию строк вместо позиционной интерполяции, если для строки перевода имеется несколько параметров. Таким образом, можно будет переупорядочить текст placeholder-a.

Множественные формы в переводе

Для множественных форм можно использовать **ngettext()** и **ngettext_lazy()**. Эти функции преобразуют формы единственного и множественного числа в зависимости от аргумента, указывающего количество объектов. В следующем примере показано, как использовать их:

```
output = ngettext('there is %(count)d product',
                  'there are %(count)d products',
                  count) % {'count': count}
```

Теперь, когда вы знаете основы перевода литералов в коде Python, пришло время применить переводы к нашему проекту.

Перевод собственного кода

Отредактируйте файл **settings.py** проекта **myshop**, импортируйте функцию **gettext_lazy()** и измените параметры **LANGUAGES** следующим образом для преобразования имен языков:

```
from django.utils.translation import gettext_lazy as _
```

```
LANGUAGES = (  
    ('en', _('English')),  
    ('es', _('Spanish')),  
)
```

Здесь мы используем функцию **gettext_lazy()** вместо **gettext()**, чтобы избежать циклического импорта, тем самым изменяя имена языков при доступе к ним.

Откройте терминал и выполните следующую команду из каталога проекта:

```
django-admin makemessages --all
```

Вы должны увидеть следующий вывод:

```
processing locale es  
processing locale en
```

Взгляните на каталог `locale/`. Вы увидите следующую структуру файлов:

```
en/  
  LC_MESSAGES/  
    django.po  
es/  
  LC_MESSAGES/  
    django.po
```

Для каждого языка создан файл сообщений.

Откройте **es/LC_MESSAGES/django.po**. В конце файла вы увидите следующее:

```
#: settings.py:104  
msgid "English"  
msgstr ""
```

```
#: settings.py:105  
msgid "Spanish"  
msgstr ""
```

Каждой строке трансляции предшествует комментарий, отображающий подробные сведения о файле и строке, где он был найден. Каждый перевод включает две строки:

- **msgid** : Строка перевода, как она отображается в исходном коде.
- **msgstr** : Перевод языка, который по умолчанию пуст. Здесь необходимо ввести фактический перевод для данной строки.

Заполните **msgstr** переводы для данной строки **msgid** следующим образом:

```
#: settings.py:104  
msgid "English"  
msgstr "Inglés"
```

```
#: settings.py:105  
msgid "Spanish"  
msgstr "Español"
```

Сохраните измененный файл сообщения, откройте терминал и выполните следующую команду:

```
django-admin compilemessages
```

Если все пройдет хорошо, вы увидите следующий результат:

```
processing file django.po in myshop/locale/en/LC_MESSAGES  
processing file django.po in myshop/locale/es/LC_MESSAGES
```

Выходные данные предоставляют сведения о компилируемых файлах сообщений. Снова взгляните на каталог **locale/** проекта **myshop**. Вы найдете в нем следующие файлы:

```
en/
  LC_MESSAGES/
    django.mo
    django.po
es/
  LC_MESSAGES/
    django.mo
    django.po
```

Можно увидеть, что для каждого языка был создан скомпилированный файл сообщений **.mo**.

Мы перевели имена самих языков. Теперь давайте преобразуем имена полей модели, отображаемые на сайте. Измените файл **models.py** приложения **orders** и добавьте имена, помеченные для перевода для полей модели **Order**, как показано ниже:

```
from django.utils.translation import gettext_lazy as _

class Order(models.Model):
    first_name = models.CharField(_('first name'), max_length=50)
    last_name = models.CharField(_('last name'), max_length=50)
    email = models.EmailField(_('e-mail'),)
    address = models.CharField(_('address'), max_length=250)
    postal_code = models.CharField(_('postal code'), max_length=20)
    city = models.CharField(_('city'), max_length=100)
    #...
```

Добавлены имена полей, отображаемых при размещении пользователем нового заказа. Это `first_name`, `last_name`, `email`, `address`, `postal_code`, и `city`. Помните, что для присвоения имен полям можно также использовать атрибут **verbose_name**.

Создайте следующую структуру в каталоге приложения **orders**:

```
locale/
  en/
  es/
```

При создании каталога **locale/** строки перевода этого приложения будут храниться в файле сообщений в этом каталоге вместо основного файла сообщений. Таким образом можно создать отдельные файлы трансляции для каждого приложения.

Откройте терминал из каталога проекта и выполните следующую команду:

```
django-admin makemessages --all
```

Вы должны увидеть следующий вывод:

```
processing locale es
processing locale en
```

Откройте файл **es/LC_MESSAGES/django.po**. Строки перевода для модели заказа будут просматриваться. Заполните следующие `msgstr` переводы для заданной `msgid` строки:

```
#: orders/models.py:10
```

```

msgid "first name"
msgstr "nombre"

#: orders/models.py:12
msgid "last name"
msgstr "apellidos"

#: orders/models.py:14
msgid "e-mail"
msgstr "e-mail"

#: orders/models.py:15
msgid "address"
msgstr "dirección"

#: orders/models.py:17
msgid "postal code"
msgstr "código postal"

#: orders/models.py:19
msgid "city"
msgstr "ciudad"

```

После завершения добавления переводов сохраните файл.

Кроме текстового редактора, можно использовать **Poedit** для редактирования переводов. **Poedit** — это программное обеспечение для редактирования переводов, и gettext использует его. Он доступен для Linux, Windows и Mac OS x. Вы можете загрузить **Poedit** здесь <http://poedit.net/>.

Давайте также переведем формы нашего проекта. **OrderCreateForm** приложения **orders** не требуется переводить, поскольку он является **ModelForm** и использует атрибут **verbose_name** полей модели **Order** для описания полей формы. Мы собираемся перевести формы **cart** и **coupons** на заявки.

Отредактируйте файл **forms.py** в каталоге приложения **cart** и добавьте атрибут **label** в поле **quantity** у **CartAddProductForm**, а затем пометьте это поле для перевода следующим образом:

```

from django import forms
from django.utils.translation import gettext_lazy as _

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, label=_('Quantity'))
    update = forms.BooleanField(required=False, initial=False,
widget=forms.HiddenInput)

```

Измените файл **forms.py** приложения **coupons** и переведите форму **CouponApplyForm** следующим образом:

```

from django import forms
from django.utils.translation import gettext_lazy as _

class CouponApplyForm(forms.Form):
    code = forms.CharField(label=_('Coupon'))

```

Мы добавили **label** в поле **code** и поместили его для перевода.

Перевод в шаблонах

Джанго предлагает теги шаблонизатора `{% trans %}` и `{% blocktrans %}` для преобразования строк в шаблонах. Чтобы использовать теги перевода шаблонизатора, необходимо добавить в верхнюю часть шаблона `{% load i18n %}`, чтобы загрузить их.

Тег шаблона `{% trans %}`

Тег шаблона `{% trans %}` позволяет пометить строку, константу или содержимое переменной для перевода. На внутреннем уровне Джанго выполняет **gettext()** в заданном месте. Вот как пометить строку для перевода в шаблоне:

```
{% trans "Text to be translated" %}
```

Вы можете использовать **as** для хранения переведенного содержимого в переменной, которая может использоваться во всех шаблонах. В следующем примере переведенный текст хранится в переменной с именем **greeting**:

```
{% trans "Hello!" as greeting %}  
<h1>{{ greeting }}</h1>
```

Тег `{% trans %}` полезен для простых строк перевода, но он не может обрабатывать содержимое для перевода, включающее переменные.

Тег шаблона `{% blocktrans %}`

Тег шаблона `{% blocktrans %}` позволяет пометить содержимое, включающее литералы и изменяемое содержимое с помощью placeholders. В следующем примере показано, как использовать тег `{% blocktrans %}`, включая переменную **name** в содержимом для перевода:

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

Можно использовать для включения таких выражений шаблона, как доступ к атрибутам объекта или применение фильтров шаблонов к переменным. Для них всегда необходимо использовать placeholders. Нельзя получить доступ к выражениям или атрибутам объекта внутри блока **blocktrans**. В следующем примере показано, как использовать для включения атрибута `object`, к которому применяется фильтр **capfirst**:

```
{% blocktrans with name=user.name|capfirst %}  
    Hello {{ name }}!  
{% endblocktrans %}
```

Используйте тег `{% blocktrans %}` вместо `{% trans %}`, когда требуется включить в строку перевода динамически изменяемое содержимое.

Перевод шаблонов магазина

Отредактируйте шаблон **shop/base.html** приложения **shop**. Убедитесь, что в верхней части шаблона загружен тег **i18n** и пометьте строки для перевода следующим образом:

```
{% load i18n %}  
{% load static %}  
<!DOCTYPE html>
```



```

<html>
<head>
  <meta charset="utf-8"/>
  <title>
    {% block title %}{% trans "My shop" %}{% endblock %}
  </title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
<div id="subheader">
  <div class="cart">
    {% with total_items=cart|length %}
      {% if cart|length > 0 %}
        {% trans "Your cart" %}:
        <a href="{% url "cart:cart_detail" %}">
          {% blocktrans with total_items_plural=total_items|pluralize
total_price=cart.get_total_price %}
            {{ total_items }} item{{ total_items_plural }},
            ${{ total_price }}
          {% endblocktrans %}
        </a>
      {% else %}
        {% trans "Your cart is empty." %}
      {% endif %}
    {% endwith %}
  </div>
</div>
<div id="content">
  {% block content %}
  {% endblock %}
</div>
</body>
</html>

```

Обратите внимание на тег `{% blocktrans %}` для отображения общей суммы корзины. Ранее он отображалась следующим образом:

```

{{ total_items }} item{{ total_items|pluralize }},
${{ cart.get_total_price }}

```

Мы использовали `{% blocktrans with ... %}` чтобы использовать placeholders для **total_items|pluralize** (тег шаблона, примененный здесь) и **cart.get_total_price** (метод объекта, доступ к которому был получен), в результате чего:

```

{% blocktrans with total_items_plural=total_items|pluralize
total_price=cart.get_total_price %}
  {{ total_items }} item{{ total_items_plural }},
  ${{ total_price }}
{% endblocktrans %}

```

Теперь отредактируйте шаблон **shop/product/detail.html** приложения **shop** и загрузите теги **i18n**, но после тега `{% extends %}`, который всегда должен быть первым тегом в шаблоне:

```
{% load i18n %}
```

Затем найдите следующую строку:

```
<input type="submit" value="Add to cart">
```

И замените ее на следующий код:

```
<input type="submit" value="{% trans "Add to cart" %}">
```

Теперь давайте переведем шаблоны приложения **orders**. Измените шаблон **orders/order/create.html** приложения **orders**:

```
{% extends "shop/base.html" %}
{% load i18n %}

{% block title %}
    {% trans "Checkout" %}
{% endblock %}

{% block content %}
    <h1>{% trans "Checkout" %}</h1>

    <div class="order-info">
        <h3>{% trans "Your order" %}</h3>
        <ul>
            {% for item in cart %}
                <li>{{ item.quantity }}x {{ item.product.name }} <span>${{
item.total_price }}</span></li>
            {% endfor %}
            {% if cart.coupon %}
                <li>
                    {% blocktrans with code=cart.coupon.code
discount=cart.coupon.discount %}
                        "{{ code }}" ({{ discount }}% off)
                    {% endblocktrans %}
                    <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
                </li>
            {% endif %}
        </ul>
        <p>{% trans "Total" %}: ${{
cart.get_total_price_after_discount|floatformat:"2" }}</p>
    </div>

    <form action="." method="post" class="order-form">
        {{ form.as_p }}
        <p><input type="submit" value="{% trans "Place order" %}"></p>
        {% csrf_token %}
    </form>
{% endblock %}
```

Посмотрите на следующие файлы, которые предоставляются вместе с этой главой, чтобы увидеть, как строки помечаются для перевода:

- Приложение **shop**, шаблон **shop/product/list.html**
- Приложение **orders**, шаблон **orders/order/created.html**
- Приложение **cart**, шаблон **cart/detail.html**

Давайте обновим файлы сообщений, чтобы включить новые строки перевода. Откройте терминал и выполните следующую команду:

```
django-admin makemessages --all
```

Загляните в файлы **.po** в каталоге **locale** проекта **myshop**, и вы увидите, что приложение **orders** теперь содержит все строки, которые были помечены для перевода.

Отредактируйте файлы перевода **.po** проекта и приложения **orders** и включите перевод на Испанский язык. В исходном коде, поставляемом вместе с данной главой, можно ссылаться на переведенные файлы **.po**.

Откройте терминал из каталога проекта и выполните следующие команды:

```
cd orders/  
django-admin compilemessages  
cd ../
```

Мы скомпилировали файлы перевода для приложения **order**.

Выполните следующую команду, чтобы переводы для приложений, не содержащих каталога **locale**, включились в файл сообщений проекта:

```
django-admin compilemessages
```

Использование интерфейса переводов Rosetta

Rosetta — это приложение стороннего производителя, позволяющее редактировать переводы с помощью того же интерфейса, что и сайт администрирования Джанго. **Rosetta** упрощает редактирование файлов **.po** и обновляет скомпилированные файлы переводов. Давайте добавим его в наш проект.

Установите **Rosetta** через **pip** с помощью этой команды:

```
pip install django-rosetta==0.7.6
```

Затем добавьте **'rosetta'** к параметрам **INSTALLED_APPS** в файле **settings.py**.

Необходимо добавить URL-адреса Rosetta в конфигурацию основного URL. Измените основной файл **urls.py** проекта и добавьте в него следующий шаблон URL-адреса:

```
url(r'^rosetta/', include('rosetta.urls')),
```

Убедитесь, что вы поместите его перед шаблоном **shop.urls**.

Откройте <http://127.0.0.1:8000/admin/> и войдите в систему с помощью суперпользователя. Затем перейдите к <http://127.0.0.1:8000/rosetta/>. Вы увидите список существующих языков:

Rosetta						
Home > Language selection						
Filter: Project Third party Django All						
English						
Application	Progress	Messages	Translated	Fuzzy	Obsolete	File
Myshop	0.00%	32	0	0	0	/Users/zenx/Django by Example/Chapter 9 – Extending your shop/code/myshop/myshop/locale/en/LC_MESSAGES/django.po
Spanish						
Application	Progress	Messages	Translated	Fuzzy	Obsolete	File
Myshop	100.00%	32	32	0	1	/Users/zenx/Django by Example/Chapter 9 – Extending your shop/code/myshop/myshop/locale/es/LC_MESSAGES/django.po

В разделе **Filter** выберите **All**, чтобы отобразить все доступные файлы сообщений, в том числе те, которые принадлежат приложению **orders**. Кликните по ссылке **Myshop** в разделе **Spanish**, чтобы изменить переводы на Испанский язык. Вы увидите список строк перевода:

Rosetta
Pick another file / Download this catalog

Home > Spanish > Myshop > Progress: 100.00%

Translate into Spanish

Display:
Untranslated only
Translated only
Fuzzy only
All

Original	Spanish	[–] Fuzzy	Occurrences(s)
Quantity	Cantidad	<input type="checkbox"/>	cart/forms.py:11 cart/templates/cart/detail.html:16
Your shopping cart	Tu carro	<input type="checkbox"/>	cart/templates/cart/detail.html:6 cart/templates/cart/detail.html.py:10
Image	Imagen	<input type="checkbox"/>	cart/templates/cart/detail.html:14
Product	Producto	<input type="checkbox"/>	cart/templates/cart/detail.html:15
Remove	Eliminar	<input type="checkbox"/>	cart/templates/cart/detail.html:17 cart/templates/cart/detail.html.py:40
Unit price	Precio unitario	<input type="checkbox"/>	cart/templates/cart/detail.html:18
Price	Precio	<input type="checkbox"/>	cart/templates/cart/detail.html:19
Update	Actualizar	<input type="checkbox"/>	cart/templates/cart/detail.html:36
Subtotal	Subtotal	<input type="checkbox"/>	cart/templates/cart/detail.html:49
<td>"%(code)s" coupon %(discount)s%% off</td>	<td>Cupón "%(code)s" %(discount)s%% de descuento</td>	<input type="checkbox"/>	cart/templates/cart/detail.html:54

Skip to page: 1 2 3 4 32/33 messages
Save and translate next block

Можно ввести переводы в колонке **Spanish**. В столбце **Occurrences** отображаются файлы и строки кода, в которых была найдена каждая строка перевода.

Переводы, включающие placeholders, будут выглядеть следующим образом:

%(total_items)s item%(total_items_plural)s, \$%(total_price)s

%(total_items)s producto%(total_items_plural)s, \$%(total_price)s

Росетта использует заполнение другим цветом фона для отображения placeholders. При переводе содержимого убедитесь, что placeholders не переведены. Например, переведите следующую строку:

```
%(total_items)s item%(total_items_plural)s, $%(total_price)s
```

Она переводится на Испанский язык следующим образом:

```
%(total_items)s producto%(total_items_plural)s, $%(total_price)s
```

Можно взглянуть на исходный код, который поставляется вместе с этой главой, чтобы использовать те же переводы на Испанский язык для проекта.

По завершении редактирования переводов нажмите кнопку **Save and translate next block**, чтобы сохранить переводы в файл .po. Rosetta компилирует файл сообщения при сохранении переводов, поэтому нет необходимости запускать команду **compilemessages**. Однако для записи файлов сообщений Rosetta требуется доступ на запись в каталоги **locale/**. Убедитесь, что каталоги имеют допустимые разрешения.

Если требуется, чтобы другие пользователи могли редактировать переводы, откройте в браузере <http://127.0.0.1:8000/admin/auth/group/add/> и создайте новую группу с именем **translators**. Затем перейдите к <http://127.0.0.1:8000/admin/auth/user/> для редактирования пользователей, которым требуется предоставить разрешения, чтобы они могли редактировать переводы. При редактировании пользователя в разделе **Permissions** добавьте группу **translators** в выбранные группы для каждого пользователя. Росетта доступен только для суперпользователей или пользователей, принадлежащих к группе **translators**.

Вы можете ознакомиться с документацией к Rosetta здесь: <http://django-rosetta.readthedocs.org/en/latest/>

*При добавлении в продакшн новых переводов, если вы работаете с реальным веб-сервером, вам придется перезагрузить сервер после запуска команды **compilemessages** или после сохранения переводов с Росетта, чтобы изменения вступили в силу.*

Fuzzy переводы

Вы могли заметить, что в Росетта есть столбец Fuzzy. Эта функция не принадлежит Rosetta; Она представлена в виде gettext. Если флаг fuzzy активен для перевода, он не будет включен в скомпилированные файлы сообщений. Этот флаг используется для строк перевода, требующих изменения от переводчика. Когда файлы .po обновляются с новыми строками перевода, возможно, что некоторые строки перевода автоматически помечаются как fuzzy.

Это происходит, когда gettext находит некоторые **msgid**, которые слегка изменились и применяет gettext, с тем, что он считает, что это старый перевод, и помечает его как fuzzy для просмотра. Переводчик должен затем проверить fuzzy переводы, удалить флаг fuzzy и снова скомпилировать файл сообщения.

Шаблоны URL-адресов для интернационализации

Джанго предлагает возможности интернационализации для URL-адресов. Он включает две основные функции для интернационализации URL-адресов:

- **Language prefix in URL patterns:** Добавление префикса языка к URL-адресам для обслуживания каждой версии языка под другим базовым URL
- **Translated URL patterns:** Пометка URL-шаблонов для перевода, чтобы один и тот же URL-адрес был разным для каждого языка

Причина перевода URL-адресов заключается в оптимизация сайта для поисковых систем. Добавление префикса языка к шаблонам позволяет индексировать URL-адрес для каждого языка, а не один URL-адрес для всех этих языков. Кроме того, при переводе URL-адресов на каждый язык будет предоставляться поисковым системам с URL-адресами, которые будут лучше ранжироваться для каждого языка.

Добавление префикса языка к шаблонам URL-адресов

Джанго позволяет добавить префикс языка к шаблонам URL-адресов. Например, английская версия веб-узла может начинаться с `/en/`, а испанская с `/es/`.

Чтобы использовать языки в шаблонах URL-адресов, необходимо убедиться в том, что **django.middleware.locale.LocaleMiddleware** указан в параметрах **MIDDLEWARE_CLASSES** в файле **settings.py**. Джанго будет использовать его для идентификации текущего языка по запрошенному URL-адресу.

Давайте добавим префикс языка к шаблонам URL-адресов. Отредактируйте основной файл **urls.py** проекта **myshop** и добавьте следующий импорт:

```
from django.conf.urls.i18n import i18n_patterns
```

Затем добавьте **i18n_patterns()** следующим образом:

```
urlpatterns = i18n_patterns(
    url(r'^admin/', include(admin.site.urls)),
    url(r'^cart/', include('cart.urls', namespace='cart')),
    url(r'^orders/', include('orders.urls', namespace='orders')),
    url(r'^payment/', include('payment.urls', namespace='payment')),
    url(r'^paypal/', include('paypal.standard.ipn.urls')),
    url(r'^coupons/', include('coupons.urls', namespace='coupons')),
    url(r'^rosetta/', include('rosetta.urls')),
    url(r'^', include('shop.urls', namespace='shop')),
)
```

Шаблоны URL-адресов можно комбинировать в **patterns()** и в разделе **i18n_patterns()**, чтобы некоторые шаблоны включали префикс языка, а другие — нет. Однако лучше использовать переведенные URL-адреса только для того, чтобы избежать возможности того, что неосторожно переведенный URL-адрес соответствует непереуведенному шаблону URL-адреса.

Запустите сервер разработки и откройте в браузере <http://127.0.0.1:8000/>. Поскольку вы используете **LocaleMiddleware** Джанго, выполнит действия, описанные в главе "Как добавить переводы в проект Джанго", чтобы определить текущий язык, а затем перенаправит вас на тот же URL-адрес, включая префикс языка. Рассмотрим URL-адрес в браузере; Теперь он должен выглядеть как <http://127.0.0.1:8000/en/>. Текущий язык будет установлен в заголовке **Accept-Language** для браузера, если он является испанским или английским, или по умолчанию **LANGUAGE_CODE**(английский), в противном случае - определяемый в настройках.

Перевод шаблонов URL-адресов

Джанго поддерживает перевод строк в шаблонах URL-адресов. Для каждого языка можно использовать свой перевод для одного шаблона URL-адреса. Можно пометить шаблоны URL для перевода так же, как и литералы, используя функцию **gettext_lazy()**.

Отредактируйте основной файл **urls.py** проекта **myshop** и добавьте строки перевода в регулярные выражения шаблонов URL-адресов для приложений **cart**, **orders**, **payment**, и **coupons**, как показано ниже:

```
from django.utils.translation import gettext_lazy as _
```

```
urlpatterns = i18n_patterns(
    url(r'^admin/', include(admin.site.urls)),
    url(_(r'^cart/'), include('cart.urls', namespace='cart')),
    url(_(r'^orders/'), include('orders.urls', namespace='orders')),
    url(_(r'^payment/'), include('payment.urls', namespace='payment')),
    url(r'^paypal/', include('paypal.standard.ipn.urls')),
    url(_(r'^coupons/'), include('coupons.urls', namespace='coupons')),
    url(r'^rosetta/', include('rosetta.urls')),
    url(r'^', include('shop.urls', namespace='shop')),
)
```

Отредактируйте файл **urls.py** приложения **orders** и пометьте шаблоны URL-адресов для перевода:

```
from django.conf.urls import url
from . import views
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    url(_(r'^create/$'), views.order_create, name='order_create'),
    # ...
]
```

Измените **urls.py** - файл приложения **payment**:

```
from django.conf.urls import url
from . import views
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    url(_(r'^process/$'), views.payment_process, name='process'),
    url(_(r'^done/$'), views.payment_done, name='done'),
    url(_(r'^canceled/$'), views.payment_canceled, name='canceled'),
]
```

Нам не нужно переводить шаблоны URL-адресов приложения - **shop**, поскольку они построены с помощью переменных и не содержат каких-либо других литералов.

Откройте терминал и выполните следующую команду, чтобы обновить файлы сообщений с помощью новых переводов:

```
django-admin makemessages --all
```

Убедитесь, что сервер разработки запущен. Откройте в вашем браузере <http://127.0.0.1:8000/en/rosetta/> и перейдите по ссылке **Myshop** под секцией **Spanish**.

Вы можете использовать фильтр для просмотра только тех строк, которые еще не были переведены. Убедитесь в том, что в переводе URL-адреса хранятся специальные символы регулярных выражений. Перевод URL-адресов является деликатной задачей; При изменении регулярного выражения можно разорвать URL-адрес.

Выбор языка пользователем

Поскольку мы обслуживаем содержимое, доступное на нескольких языках, мы должны позволить пользователям переключаться на любой язык. Мы собираемся добавить селектор языка на наш сайт. Селектор языка будет состоять из списка доступных языков, отображаемых с помощью ссылок.

Откройте шаблон **shop/base.html** и найдите в нем следующие строки:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```

Замените их следующим кодом:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
  {% get_current_language as LANGUAGE_CODE %}
  {% get_available_languages as LANGUAGES %}
  {% get_language_info_list for LANGUAGES as languages %}
  <div class="languages">
    <p>{% trans "Language" %}</p>
    <ul class="languages">
      {% for language in languages %}
        <li>
          <a href="{% language.code %}" {% if language.code ==
LANGUAGE_CODE %} class="selected"{% endif %}>
            {{ language.name_local }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </div>
</div>
```

Вот как мы создаем селектор языка:

Сначала мы загружаем теги мультиязычности, используя `{% load i18n %}`. Для извлечения текущего языка используется тег `{% get_current_language %}`. Мы получаем языки, определенные в параметре **LANGUAGES**, используя тег шаблона `{% get_available_languages %}`.

Для обеспечения простого доступа к атрибутам языка используется тег `{% get_language_info_list %}`.

Мы строим HTML список для отображения всех доступных языков и добавляем выбранный атрибут класса к текущему активному языку.

Мы используем теги шаблонов, предоставленные **i18n** на основе языков, доступных в настройках проекта. Теперь откройте <http://127.0.0.1:8000/>. Селектор языка в правой верхней части страницы должен выглядеть следующим образом:

My shop

Language: English Spanish

Your cart: 4 items, \$95,80

Теперь пользователи могут легко переключать языки.

Перевод моделей с django-parler

Джанго не предоставляет решения для перевода моделей из коробки. Необходимо реализовать собственное решение для управления контентом, хранящимся на разных языках, или использовать модуль стороннего производителя для преобразования модели. Существует несколько приложений сторонних разработчиков, которые позволяют перевести поля

модели. Каждый из них использует свой подход к хранению и доступу к переводу. Одно из этих приложений — **django-parler**. Этот модуль обеспечивает очень эффективный способ перевода моделей и легко интегрируется с сайтом администрирования Django.

django-parler генерирует отдельную таблицу базы данных для каждой модели, содержащей переводы. Эта таблица содержит все переведенные поля и foreign key для исходного объекта, к которому принадлежит перевод. Он также содержит поле языка, поскольку каждая строка хранит содержимое одного языка.

Установка django-parler

Установите **django-parler** через **pip**, используя следующую команду:

```
pip install django-parler==1.5.1
```

Затем отредактируйте файл **settings.py** проекта **myshop** и добавьте **'parler'** в параметры **INSTALLED_APPS**. Добавьте также следующий код в файл **settings.py**:

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en'},
        {'code': 'es'},
    ),
    'default': {
        'fallback': 'en',
        'hide_untranslated': False,
    }
}
```

Этот параметр определяет доступные языки **en** и **es** для **django-parler**. Мы указываем язык по умолчанию **en** и указываем, что **django-parler** не должен скрывать непереуведенное содержимое.

Перевод полей модели

Давайте добавим переводы для нашего каталога товаров. **django-parler** предоставляет класс модели **TranslatedModel** и **TranslatedFields** - оболочку для преобразования полей модели. Отредактируйте файл **models.py** в каталоге приложения **shop** и добавьте следующий импорт:

```
from parler.models import TranslatableModel, TranslatedFields
```

Затем измените модель **Category**, чтобы перевести поля **name** и **slug**:

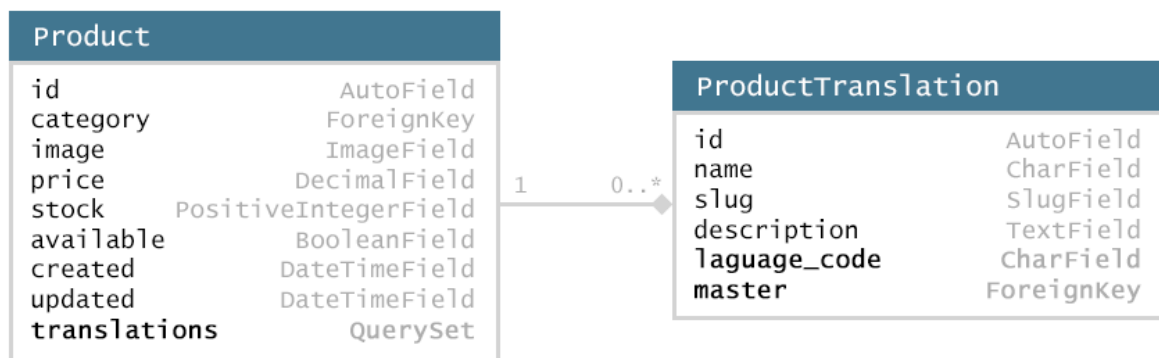
```
class Category(TranslatableModel):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True, unique=True)
    translations = TranslatedFields(
        name=models.CharField(max_length=200, db_index=True),
        slug=models.SlugField(max_length=200, db_index=True, unique=True)
    )
```

Модель **Category** наследуется теперь от **TranslatedModel** вместо **models.Model**, поля **name** и **slug** включены в **TranslatedFields**.

Измените модель **Product**, чтобы добавить переводы для полей **name**, **slug**, и **description**. Также оставьте непереведенные поля на данный момент:

```
class Product(TranslatableModel):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    description = models.TextField(blank=True)
    translations = TranslatedFields(
        name=models.CharField(max_length=200, db_index=True),
        slug=models.SlugField(max_length=200, db_index=True),
        description=models.TextField(blank=True)
    )
    category = models.ForeignKey(Category, related_name='products')
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.PositiveIntegerField()
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

django-parler генерирует новую модель для каждой переводимой модели. На следующем изображении можно увидеть поля модели **Product**, и поля сгенерированные моделью **ProductTranslation**:



Модель **ProductTranslation**, созданная с помощью **django-parler**, включает поля **name**, **slug**, и **description**, поле **language_code** и **ForeignKey** для основного объекта **Product**. Связь "один ко многим" от **Product** до **ProductTranslation**. Объект **ProductTranslation** будет существовать для каждого из доступных языков каждого объекта **Product**.

Поскольку Django использует отдельную таблицу для перевода, есть некоторые функции, которые мы не можем использовать. Невозможно использовать упорядочение по умолчанию в преобразованном поле. Можно фильтровать по переводимым полям в запросах, но нельзя включить переведенное поле в параметрах **ordering Meta**. Отредактируйте файл **models.py** приложения **shop** и закомментируйте атрибут упорядочивания класса **Category Meta**:

```
class Meta:
    # ordering = ('name',)
    verbose_name = 'category'
    verbose_name_plural = 'categories'
```

Мы также должны закомментировать атрибут `index_together` класса **Product Meta**, поскольку текущая версия **django-parler** не обеспечивает поддержку его проверки. Отредактируйте класс **Product Meta**:

```
class Meta:
    ordering = ('-created',)
    # index_together = (('id', 'slug'),)
```

Вы можете прочитать дополнительные сведения о совместимости **django-parler** с Django здесь: <http://django-parler.readthedocs.org/en/latest/compatibility.html>

Создание кастомной миграции

При создании новых моделей с переводом необходимо выполнить `makemigrations` для создания миграции моделей, а затем выполнить миграцию для синхронизации изменений в базе данных. Однако при преобразовании существующих полей, вероятно, появятся данные в базе данных, которые необходимо сохранить. Мы собираемся перенести наши текущие данные в новые модели перевода. Поэтому мы добавили переведенные поля, но мы преднамеренно сохранили исходные поля. Для добавления переводов в существующие поля выполните следующие действия:

1. Мы создаем перенос новых полей модели для перевода, сохраняя исходные поля.
2. Мы создаем пользовательский перенос для копирования данных из существующих полей в модели перевода.
3. Мы удаляем существующие поля из исходных моделей.

Выполните следующую команду, чтобы создать миграцию для полей переводов, добавленных в модели **Category** и **Product**:

```
python manage.py makemigrations shop --name "add_translation_model"
```

Вы должны увидеть следующий вывод:

```
Migrations for 'shop':
  0002_add_translation_model.py:
    - Create model CategoryTranslation
    - Create model ProductTranslation
    - Change Meta options on category
    - Alter index_together for product (0 constraint(s))
    - Add field master to producttranslation
    - Add field master to categorytranslation
    - Alter unique_together for producttranslation (1 constraint(s))
    - Alter unique_together for categorytranslation (1 constraint(s))
```

Перенос существующих данных

Теперь необходимо создать кастомную миграцию для копирования существующих данных в новые модели перевода. Создайте пустую миграцию с помощью этой команды:

```
python manage.py makemigrations --empty shop --name "migrate_translatable_fields"
```

Вы получите следующий результат:

Migrations for 'shop':

0003_migrate_translatable_fields.py

Отредактируйте файл **shop/migrations/0003_migrate_translatable_fields.py** и добавьте в него следующий код:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models, migrations
from django.apps import apps
from django.conf import settings
from django.core.exceptions import ObjectDoesNotExist

translatable_models = {
    'Category': ['name', 'slug'],
    'Product': ['name', 'slug', 'description'],
}

def forwards_func(apps, schema_editor):
    for model, fields in translatable_models.items():
        Model = apps.get_model('shop', model)
        ModelTranslation = apps.get_model('shop', '{}Translation'.format(model))

        for obj in Model.objects.all():
            translation_fields = {field: getattr(obj, field) for field in fields}
            translation = ModelTranslation.objects.create(
                master_id=obj.pk,
                language_code=settings.LANGUAGE_CODE,
                **translation_fields)

def backwards_func(apps, schema_editor):
    for model, fields in translatable_models.items():
        Model = apps.get_model('shop', model)
        ModelTranslation = apps.get_model('shop', '{}Translation'.format(model))

        for obj in Model.objects.all():
            translation = _get_translation(obj, ModelTranslation)
            for field in fields:
                setattr(obj, field, getattr(translation, field))
            obj.save()

def _get_translation(obj, MyModelTranslation):
    translations = MyModelTranslation.objects.filter(master_id=obj.pk)
    try:
        # Try default translation
        return translations.get(language_code=settings.LANGUAGE_CODE)
    except ObjectDoesNotExist:
        return translations.get()

class Migration(migrations.Migration):

    dependencies = [
        ('shop', '0002_add_translation_model'),
    ]

    operations = [
```

```
migrations.RunPython(forwards_func, backwards_func),  
]
```

Эта миграция включает функции **forwards_func()** и **backwards_func()**, содержащие код, который должен быть выполнен для применения/изменения миграции.

1. Процесс миграции работает следующим образом:
2. Мы определим модели и их переводимые поля в словаре **translatable_models**.
3. Чтобы применить миграцию, мы перейдем к моделям, включающим переводы для получения модели и ее преобразованных классов модели с **app.get_model()**.
4. Мы перейдем все существующие объекты в базе данных и создадим объект перевода для **LANGUAGE_CODE**, определенного в параметрах проекта. Мы включаем **ForeignKey** в исходный объект и копию для каждого поля преобразования из исходных полей.

Функция **backward** извлекает объект перевода по умолчанию и копирует значения переведенных полей обратно в исходный объект.

Мы создали миграцию для добавления полей перевода, а затем переноса для копирования содержимого из существующих полей в новые модели перевода.

Наконец, нам нужно удалить исходные поля, которые нам больше не нужны. Отредактируйте файл **models.py** приложения **shop** и удалите поля **name** и **slug** модели **Category**. Поля модели **Category** теперь должны выглядеть следующим образом:

```
class Category(TranslatableModel):  
    translations = TranslatedFields(  
        name=models.CharField(max_length=200, db_index=True),  
        slug=models.SlugField(max_length=200, db_index=True, unique=True)  
    )
```

Удалите поля **name**, **slug**, и **description** модели **Product**. Теперь она должна выглядеть следующим образом:

```
class Product(TranslatableModel):  
    translations = TranslatedFields(  
        name=models.CharField(max_length=200, db_index=True),  
        slug=models.SlugField(max_length=200, db_index=True),  
        description=models.TextField(blank=True)  
    )  
    category = models.ForeignKey(Category, related_name='products')  
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)  
    price = models.DecimalField(max_digits=10, decimal_places=2)  
    stock = models.PositiveIntegerField()  
    available = models.BooleanField(default=True)  
    created = models.DateTimeField(auto_now_add=True)  
    updated = models.DateTimeField(auto_now=True)
```

Теперь необходимо создать окончательную миграцию, которая отражала бы это изменение модели. Однако, если мы попытаемся запустить утилиту **manage.py**, мы получим ошибку, потому что мы еще не адаптировали админ-панель к моделям, которые можно перевести.

Интеграция переводов в админ-панель

Django-parler легко интегрируется с админ-панелью Джанго. Он включает класс **TranslatableAdmin**, который переопределяет класс **ModelAdmin**, предоставленный Джанго для управления переводом моделей.

Отредактируйте файл **admin.py** приложения **shop** и добавьте в него следующий импорт:

```
from parler.admin import TranslatableAdmin
```

Измените классы **CategoryAdmin** и **ProductAdmin** на наследование от **TranslatableAdmin** вместо **ModelAdmin**. **Django-parler** еще не поддерживает атрибут **prepopulated_fields**, но он поддерживает метод **get_prepopulated_fields()**, предоставляющий те же функциональные возможности. Теперь файл **admin.py** должен выглядеть следующим образом:

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

class CategoryAdmin(TranslatableAdmin):
    list_display = ['name', 'slug']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}
admin.site.register(Category, CategoryAdmin)

class ProductAdmin(TranslatableAdmin):
    list_display = ['name', 'slug', 'category', 'price', 'stock', 'available', 'created',
'updated']
    list_filter = ['available', 'created', 'updated', 'category']
    list_editable = ['price', 'stock', 'available']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}
admin.site.register(Product, ProductAdmin)
```

Мы адаптировали админ-панель для работы с новыми переведенными моделями. Теперь можно синхронизировать базу данных с внесенными изменениями модели.

Применение миграций для модели перевода

Мы удалили старые поля из наших моделей перед адаптацией админ-панели. Теперь необходимо создать миграцию для этого изменения. Откройте терминал и выполните следующую команду:

```
python manage.py makemigrations shop --name "remove_untranslated_fields"
```

Будут выведены следующие данные:

```
Migrations for 'shop':
  0004_remove_untranslated_fields.py:
    - Remove field name from category
    - Remove field slug from category
    - Remove field description from product
    - Remove field name from product
    - Remove field slug from product
```

С помощью этой миграции мы удаляем исходные поля и сохраняем переводимые поля.

Таким образом мы создали следующие миграции:

1. Добавлены поля для перевода в модели.
2. Перенос существующих данных из исходных полей в поля перевода.

3. Удалены исходные поля из моделей.

Выполните следующую команду для применения трех созданных нами миграций:

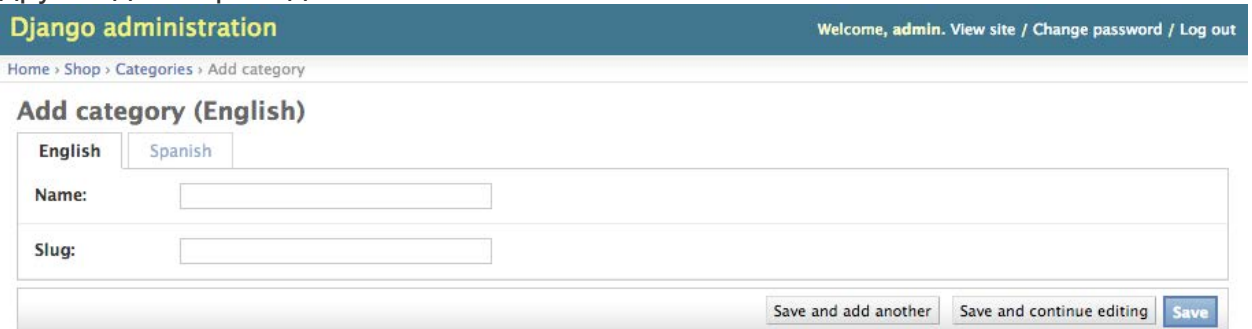
```
python manage.py migrate shop
```

Будет выведен результат, включающий следующие строки:

```
Applying shop.0002_add_translation_model... OK
Applying shop.0003_migrate_translatable_fields... OK
Applying shop.0004_remove_untranslated_fields... OK
```

Теперь наши модели синхронизированы с базой данных. Давайте переведем объект.

Запустите сервер разработки с помощью `python manage.py runserver` и откройте в браузере <http://127.0.0.1:8000/en/admin/shop/category/add/>. Вы увидите, что страница **Add category** содержит две вкладки: одна для английского языка, другая для переводов на испанский.



Django administration Welcome, admin. View site / Change password / Log out

Home > Shop > Categories > Add category

Add category (English)

English **Spanish**

Name:

Slug:

Теперь можно добавить перевод и нажать кнопку **Save**. Перед изменением убедитесь, что внесенные изменения сохранены.

Адаптация представлений для переводов

Мы должны адаптировать свои **shop** views, чтобы использовать переводы QuerySets. Выполните в терминале `python manage.py shell` и посмотрите, как можно получить поля перевода и запросить их. Чтобы получить содержимое поля для текущего активного языка, нужно просто получить доступ к полю таким же образом, как и к любому полю обычной модели:

```
>>> from shop.models import Product
>>> product=Product.objects.first()
>>> product.name
'Black tea'
```

При доступе к переведенным полям они отдаются с использованием текущего языка. Можно задать другой текущий язык для объекта, чтобы получить доступ к определенному переводу:

```
>>> product.set_current_language('es')
>>> product.name
'Té negro'
>>> product.get_current_language()
'es'
```

При выполнении запроса с помощью **filter()** можно фильтровать использование связанных объектов перевода с помощью синтаксиса **translations__** следующим образом:

```
>>> Product.objects.filter(translations__name='Black tea')
[<Product: Black tea>]
```

Также можно использовать **language()** чтобы установить определенный язык для объектов, получаемых следующим образом:

```
>>> Product.objects.language('es').all()
[<Product: Té negro>, <Product: Té en polvo>, <Product: Té rojo>,
<Product: Té verde>]
```

Как вы видите, способ доступа к полям и запрос на их перевод довольно прост.

Давайте адаптируем представления каталога продуктов. Измените файл **views.py** приложения **shop** и в представлении **product_list** найдите следующую строку:

```
category = get_object_or_404(Category, slug=category_slug)
```

Замените ее на следующий код:

```
category = get_object_or_404(Category,
                             translations__language=language,
                             translations__slug=category_slug)
```

Затем измените в представлении **product_detail** следующую строку:

```
product = get_object_or_404(Product,
                             id=id,
                             slug=slug,
                             available=True)
```

На этот код:

```
product = get_object_or_404(Product,
                             id=id,
                             translations__language_code=language,
                             translations__slug=slug,
                             available=True)
```

Представления **product_list** и **product_detail** теперь адаптированы для извлечения объектов с помощью переводимых полей. Запустите сервер разработки и откройте в браузере <http://127.0.0.1:8000/es/> . Вы увидите список продуктов, включая все продукты, переведенные на Испанский язык:

Categorías

Todos

Té

Productos



Té negro
\$32,20



Té en polvo
\$21,20



Té rojo
\$45,50



Té verde
\$30,00