**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 Джанго хранит сессии в базе данных, используя модель **Session** приложения **django.contrib.sessions**.

Джанго предлагает следующие варианты хранения данных сессий:

* **Database sessions**: Данные сессии хранятся в базе данных. Это defaultsession 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. Джанго 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](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 Джанго 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** прректа **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 нужнаются 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 }}">

<img src="

{% if product.image %}

{{ product.image.url }}

{% else %}

{% static "img/no\_image.png" %}

{% endif %}">

</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>

{% endwith %}

{% 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">

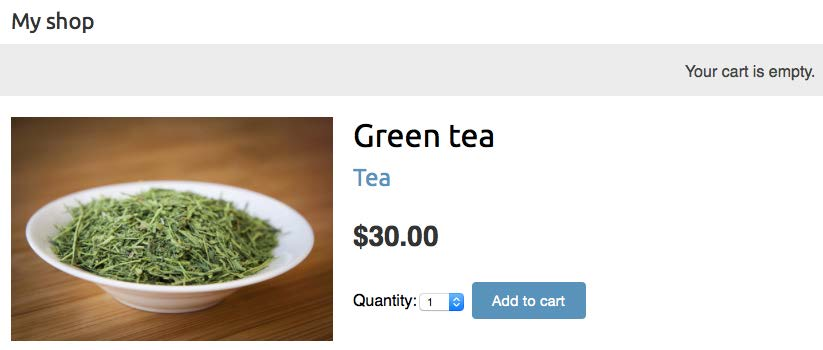
{{ cart\_product\_form }}

{% csrf\_token %}

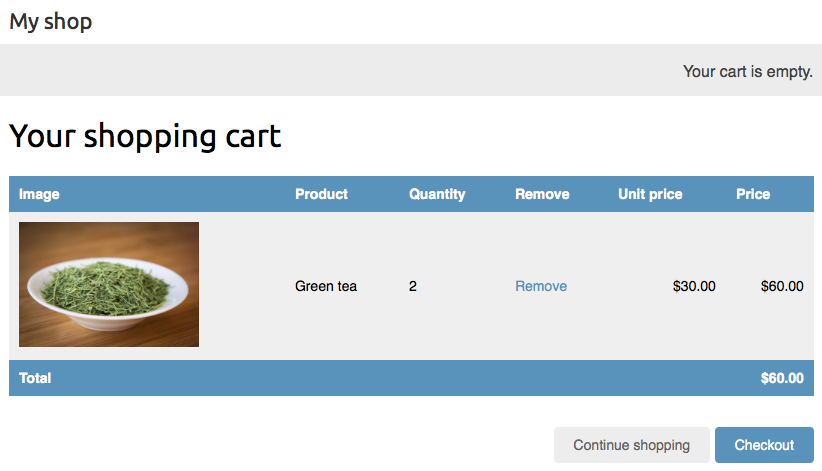
<input type="submit" value="Add to cart">

</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:



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:



## Создание обработчика контекста для текущей корзины

Возможно, вы заметили, что в заголвке 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](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 парамертах **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>

Запкстите 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 стоимось:

