# 9

# Managing Payments and Orders

In the previous chapter, you created a basic online shop with a product catalog and a shopping cart. You learned how to use Django sessions and built a custom context processor. You also learned how to launch asynchronous tasks using Celery and RabbitMQ.

In this chapter, you will learn how to integrate a payment gateway into your site to let users pay by credit card and manage order payments. You will also extend the administration site with different features.

In this chapter, you will:

- Integrate the Stripe payment gateway into your project
- Process credit card payments with Stripe
- Handle payment notifications and mark orders as paid
- Export orders to CSV files
- Create custom views for the administration site
- Generate PDF invoices dynamically

## Functional overview

*Figure 9.1* shows a representation of the views, templates, and functionalities that will be built in this chapter:

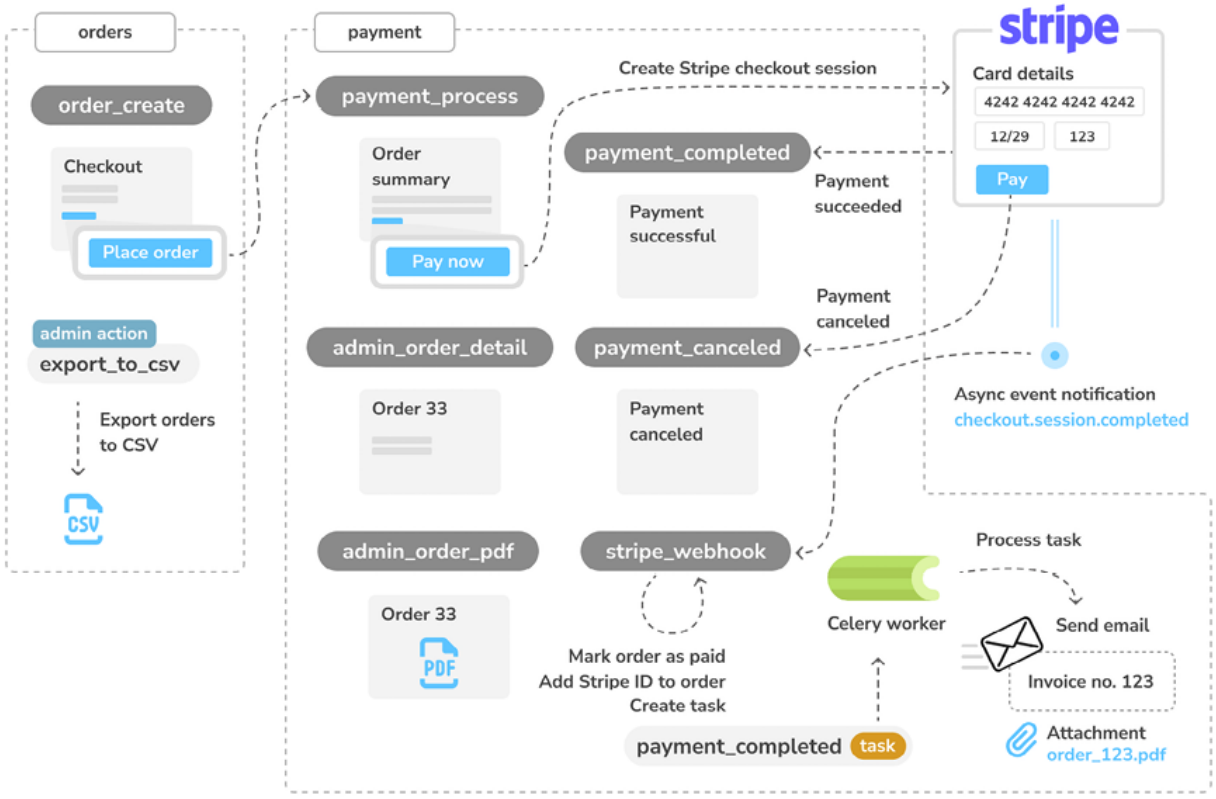



*Figure 9.1: Diagram of the functionalities built in Chapter 9*

In this chapter, you will create a new `payment` app, where you will implement the `payment_process` view to initiate a checkout session to pay orders with Stripe. You will build the `payment_completed` view to redirect users after successful payments and the `payment_canceled` view to redirect users if the payment is canceled. You will implement the `export_to_csv` admin action to export orders in CSV format in the administration site. You will also build the admin view `admin_order_detail` to display order details and the `admin_order_pdf` view to generate PDF invoices dynamically. You will implement the `stripe_webhook` webhook to receive asynchronous payment notifications from Stripe, and you will implement the

`payment_completed` asynchronous task to send invoices to clients when orders are paid.

The source code for this chapter can be found at https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter09.

All Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all the requirements at once with the command `python -m pip install -r requirements.txt`.

# Integrating a payment gateway

A payment gateway is a technology used by merchants to process payments from customers online. Using a payment gateway, you can manage customers' orders and delegate payment processing to a reliable, secure third party. By using a trusted payment gateway, you won't have to worry about the technical, security, and regulatory complexity of processing credit cards in your own system.

There are several payment gateway providers to choose from. We are going to integrate Stripe, which is a very popular payment gateway used by online services such as Shopify, Uber, Twitch, and GitHub, among others.

Stripe provides an **Application Programming Interface** (**API**) that allows you to process online payments with multiple payment methods, such as credit card, Google Pay, and Apple Pay. You can learn more about Stripe at https://www.stripe.com/.

Stripe provides different products related to payment processing. It can manage one-off payments, recurring payments for subscription services, multiparty payments for platforms and marketplaces, and more.

Stripe offers different integration methods, from Stripe-hosted payment forms to fully customizable checkout flows. We will integrate the *Stripe Checkout* product, which consists of a payment page optimized for conversion. Users will be able to easily pay with a credit card or other payment methods for the items they order. We will receive payment notifications from Stripe. You can see the *Stripe Checkout* documentation at `https://stripe.com/docs/payments/checkout`.

By leveraging *Stripe Checkout* to process payments, you rely on a solution that is secure and compliant with **Payment Card Industry** (**PCI**) requirements. You will be able to collect payments from Google Pay, Apple Pay, Afterpay, Alipay, SEPA direct debits, Bacs direct debits, BECS direct debits, iDEAL, Sofort, GrabPay, FPX, and other payment methods.

# Creating a Stripe account

You need a Stripe account to integrate the payment gateway into your site. Let’s create an account to test the Stripe API. Open `https://dashboard.stripe.com/register` in your browser.

You will see a form like the following one:

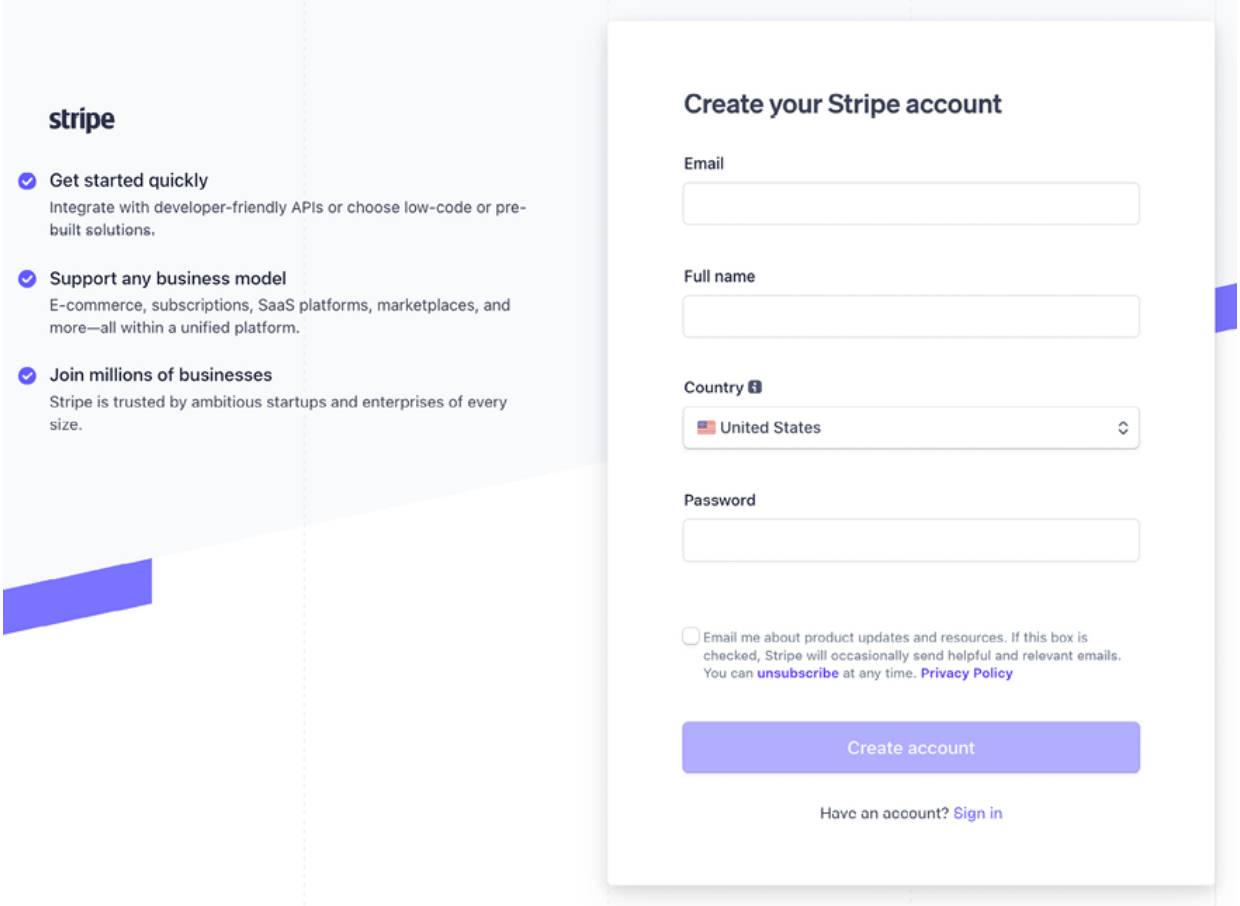



*Figure 9.2: The Stripe signup form*

Fill in the form with your own data and click on **Create account**. You will receive an email from Stripe with a link to verify your email address. The email will look like this:

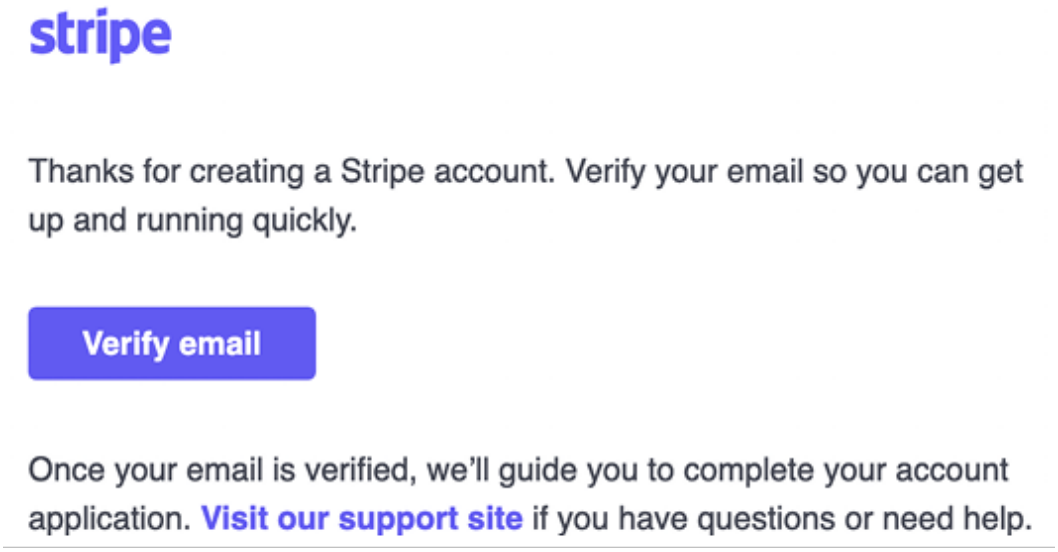



*Figure 9.3: The verification email to verify your email address*

Open the email in your inbox and click on **Verify email**.

You will be redirected to the Stripe dashboard screen, which will look like this:

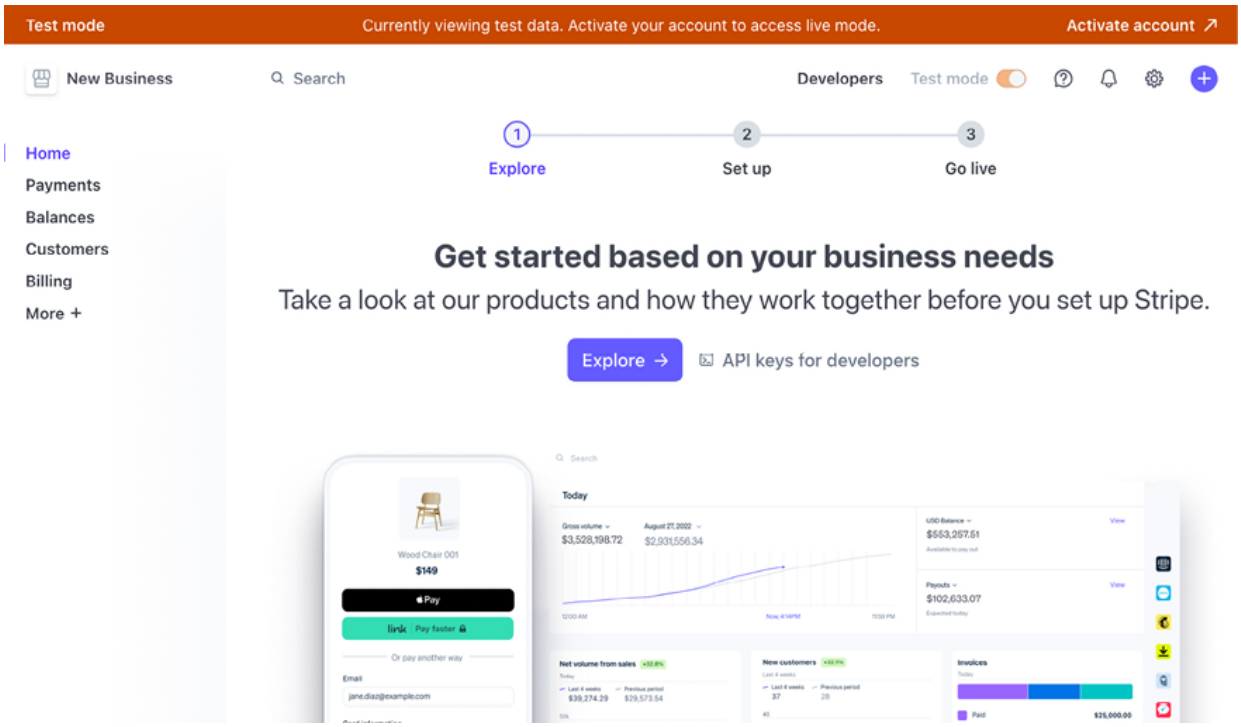


*Figure 9.4: The Stripe dashboard after verifying the email address*

At the top right of the screen, you can see that **Test mode** is activated. Stripe provides you with a test environment and a production environment. If you own a business or are a freelancer, you can add your business details to activate the account and get access to process real payments. However, this is not necessary to implement and test payments through Stripe, as we will be working in the test environment.

You need to add an account name to process payments. Open `https://dashboard.stripe.com/settings/account` in your browser.

You will see the following screen:

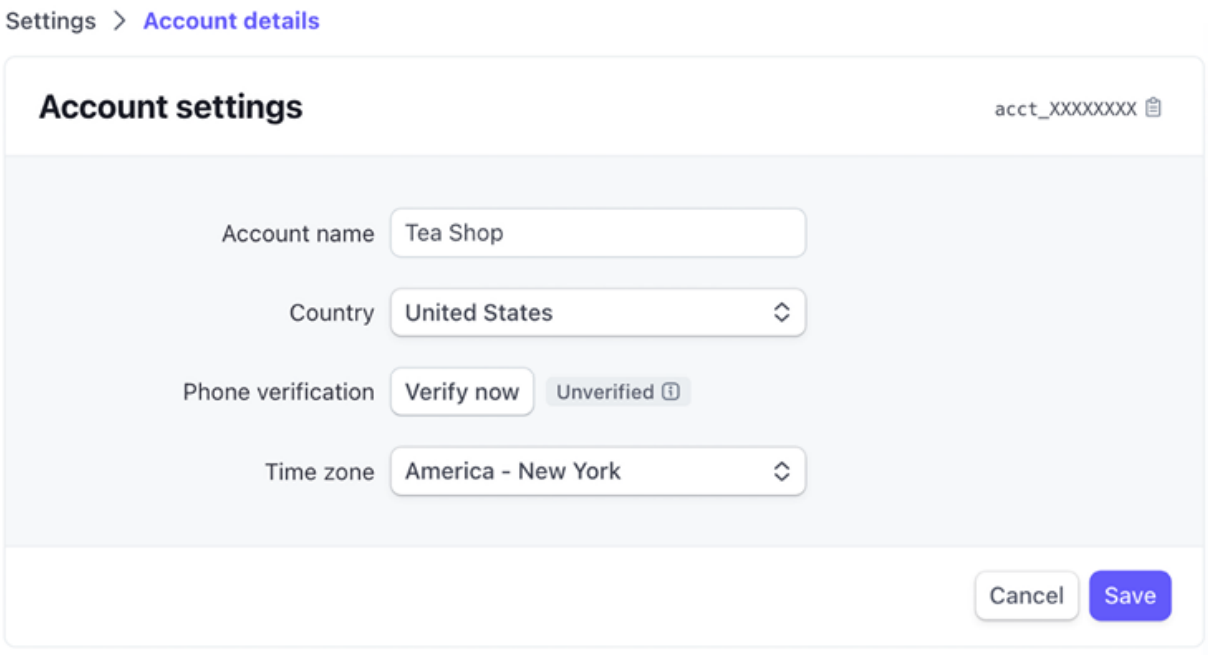



*Figure 9.5: The Stripe account settings*

Under **Account name**, enter the name of your choice and then click on **Save**. Go back to the Stripe dashboard. You will see your account name displayed in the header:

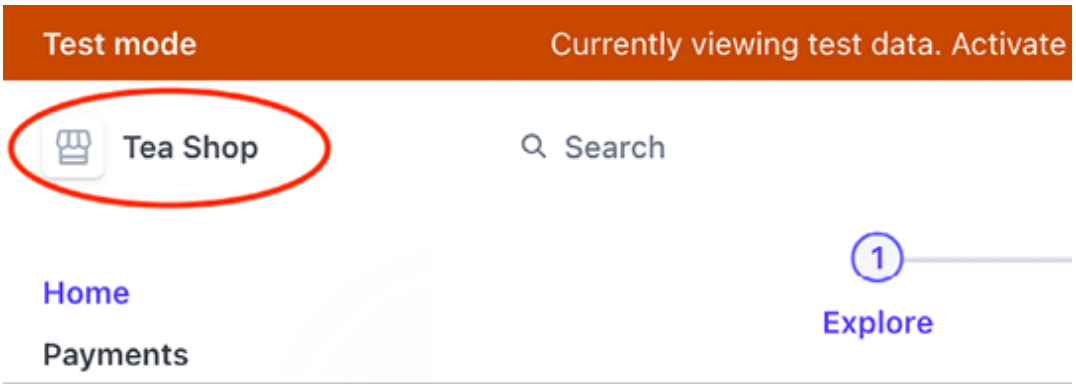



*Figure 9.6: The Stripe dashboard header including the account name*

We will continue by installing the Stripe Python SDK and adding Stripe to our Django project.

# Installing the Stripe Python library

Stripe provides a Python library that simplifies dealing with its API. We are going to integrate the payment gateway into the project using the `stripe` library.

You can find the source code for the Stripe Python library at https://github.com/stripe/stripe-python.

Install the `stripe` library from the shell using the following command:

```
python -m pip install stripe==9.3.0
```

# Adding Stripe to your project

Open https://dashboard.stripe.com/test/apikeys in your browser. You can also access this page from the Stripe dashboard by clicking

on **Developers** and then clicking on **API keys**. You will see the following screen:

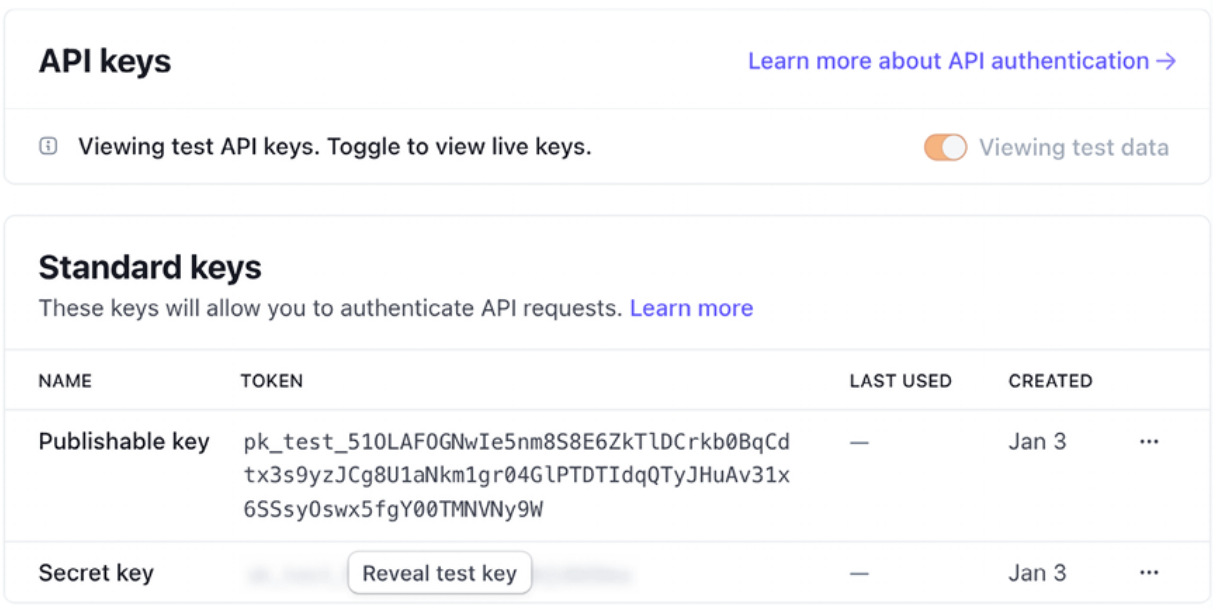



*Figure 9.7: The Stripe test API keys screen*

Stripe provides a key pair for the two different environments, test and production. There is a **Publishable key** and a **Secret key** for each environment. Test mode publishable keys have the prefix `pk_test_` and live mode publishable keys have the prefix `pk_live_`. Test mode secret keys have the prefix `sk_test_` and live mode secret keys have the prefix `sk_live_`.

You will need this information to authenticate requests to the Stripe API. You should always keep your private key secret and store it securely. The publishable key can be used in client-side code such as JavaScript scripts. You can read more about Stripe API keys at https://stripe.com/docs/keys.

To facilitate separating configuration from code, we are going to use `python-decouple`. You already used this library in *Chapter 2, Enhancing Your Blog and Adding Social Features*.

Create a new file inside your project's root directory and name it `.env`. The `.env` file will contain key-value pairs of environment variables. Add the Stripe credentials to the new file, as follows:

```
STRIPE_PUBLISHABLE_KEY=pk_test_XXXX
STRIPE_SECRET_KEY=sk_test_XXXX
```

Replace the `STRIPE_PUBLISHABLE_KEY` and `STRIPE_SECRET_KEY` values with the test **Publishable key** and **Secret key** values provided by Stripe.

If you are using a `git` repository for your code, make sure to include `.env` in the `.gitignore` file of your repository. By doing so, you ensure that credentials are excluded from the repository.

Install `python-decouple` via `pip` by running the following command:

```
python -m pip install python-decouple==3.8
```

Edit the `settings.py` file of your project and add the following code to it:

```
from decouple import config
# ...
STRIPE_PUBLISHABLE_KEY = config('STRIPE_PUBLISHABLE_KEY')
STRIPE_SECRET_KEY = config('STRIPE_SECRET_KEY')
STRIPE_API_VERSION = '2024-04-10'
```

You will use Stripe API version `2024-04-10`. You can see the release notes for this API version at https://stripe.com/docs/upgrades#2024-04-10.



You are using the test environment keys for the project. Once you go live and validate your Stripe account, you will obtain

the production environment keys. In *Chapter 17*, *Going Live*, you will learn how to configure settings for multiple environments.

Let's integrate the payment gateway into the checkout process. You can find the Python documentation for Stripe at https://stripe.com/docs/api?lang=python.

# Building the payment process

The checkout process will work as follows:

1. Add items to the shopping cart.
2. Check out the shopping cart.
3. Enter credit card details and pay.

We are going to create a new application to manage payments. Create a new application in your project using the following command:

```
python manage.py startapp payment
```

Edit the `settings.py` file of the project and add the new application to the `INSTALLED_APPS` setting, as follows. The new line is highlighted in bold:

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

The `payment` application is now active in the project.

Currently, users are able to place orders but they cannot pay for them. After clients place an order, we need to redirect them to the payment process.

Edit the `views.py` file of the `orders` application and include the following import:

```
from django.shortcuts import redirect, render
```

In the same file, find the following lines of the `order_create` view:

```
# launch asynchronous task
order_created.delay(order.id)
return render(
  request, 'orders/order/created.html', {'order': order}
)
```

Replace them with the following code:

```
# launch asynchronous task
order_created.delay(order.id)
# set the order in the session
request.session['order_id'] = order.id
# redirect for payment
return redirect('payment:process')
```

The edited view should look as follows:

```
from django.shortcuts import redirect, render
# ...
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']
                    )
                # clear the cart
                cart.clear()
                # launch asynchronous task
                order_created.delay(order.id)
 # set the order in the session
                request.session['order_id'] = order.id
 # redirect for payment
 return redirect('payment:process')
else:
            form = OrderCreateForm()
        return render(
            request,
            'orders/order/create.html',
            {'cart': cart, 'form': form}
        )
```

Instead of rendering the template `orders/order/created.html` when placing a new order, the order ID is stored in the user session and the user is redirected to the `payment:process` URL. We are going to implement this URL later. Remember that Celery has to be running for the `order_created` task to be queued and executed.

Let's integrate the payment gateway.

## Integrating Stripe Checkout

The Stripe Checkout integration consists of a checkout page hosted by Stripe that allows the user to enter the payment details, usually a credit card, and then it collects the payment. If the payment is successful, Stripe redirects the

client to a success page. If the payment is canceled by the client, it redirects the client to a cancel page.

We will implement three views:

- `payment_process`: Creates a Stripe **Checkout Session** and redirects the client to the Stripe-hosted payment form. A checkout session is a programmatic representation of what the client sees when they are redirected to the payment form, including the products, quantities, currency, and amount to charge.
- `payment_completed`: Displays a message for successful payments. The user is redirected to this view if the payment is successful.
- `payment_canceled`: Displays a message for canceled payments. The user is redirected to this view if the payment is canceled.

*Figure 9.8* shows the checkout payment flow:

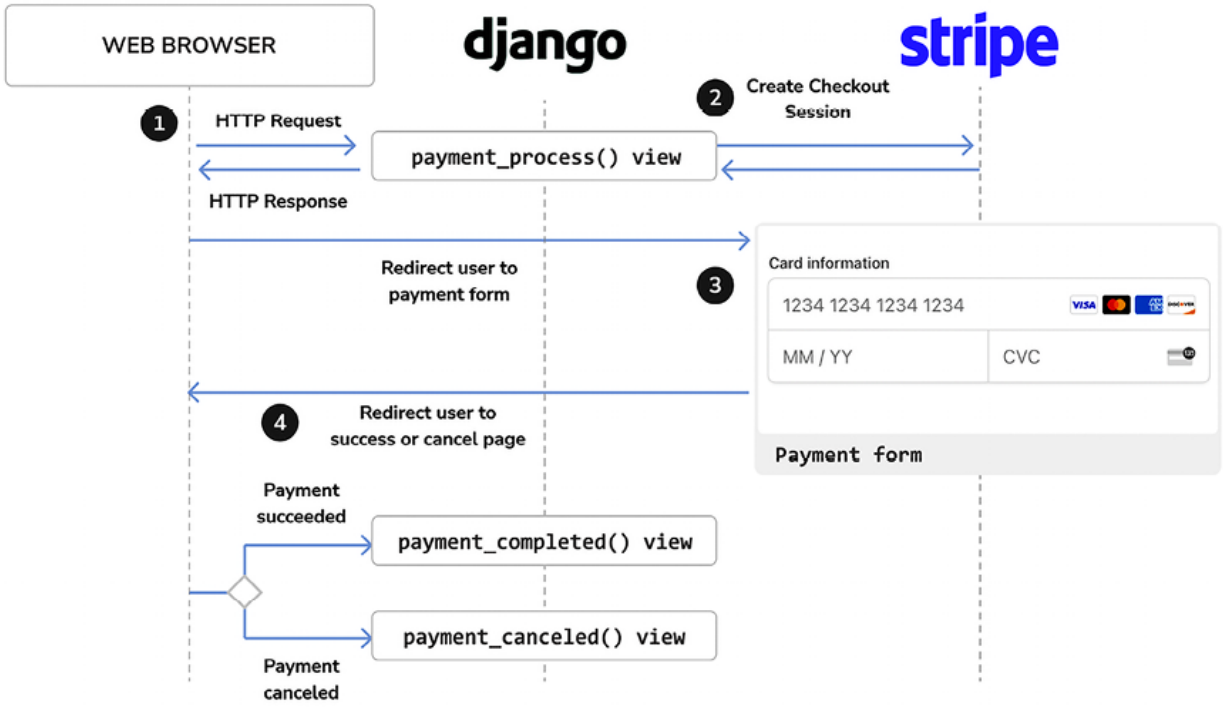



*Figure 9.8: The checkout payment flow*

The complete checkout process will work as follows:

1. After an order is created, the user is redirected to the `payment_process` view. The user is presented with an order summary and a button to proceed with the payment.
2. When the user proceeds to pay, a Stripe checkout session is created. The checkout session includes the list of items that the user will purchase, a URL to redirect the user to after a successful payment, and a URL to redirect the user to if the payment is canceled.
3. The view redirects the user to the Stripe-hosted checkout page. This page includes the payment form. The client enters their credit card details and submits the form.
4. Stripe processes the payment and redirects the client to the `payment_completed` view. If the client doesn't complete the payment, Stripe redirects the client to the `payment_canceled` view instead.

Let's start building the payment views. Edit the `views.py` file of the `payment` application and add the following code to it:

```
from decimal import Decimal
import stripe
from django.conf import settings
from django.shortcuts import get_object_or_404, redirect, render
from django.urls import reverse
from orders.models import Order
# create the Stripe instance
stripe.api_key = settings.STRIPE_SECRET_KEY
stripe.api_version = settings.STRIPE_API_VERSION
def payment_process(request):
    order_id = request.session.get('order_id')
    order = get_object_or_404(Order, id=order_id)
    if request.method == 'POST':
        success_url = request.build_absolute_uri(
            reverse('payment:completed')
        )
```

```
            cancel_url = request.build_absolute_uri(
                reverse('payment:canceled')
            )
            # Stripe checkout session data
            session_data = {
                'mode': 'payment',
                'client_reference_id': order.id,
                'success_url': success_url,
                'cancel_url': cancel_url,
                'line_items': []
            }
            # create Stripe checkout session
            session = stripe.checkout.Session.create(**session_data)
            # redirect to Stripe payment form
    return redirect(session.url, code=303)
        else:
            return render(request, 'payment/process.html', locals())
```

In the previous code, the `stripe` module is imported and the Stripe API key is set using the value of the `STRIPE_SECRET_KEY` setting. The API version to use is also set using the value of the `STRIPE_API_VERSION` setting.

The `payment_process` view performs the following tasks:

1. The current `Order` object is retrieved from the database using the `order_id` session key, which was stored previously in the session by the `order_create` view.
2. The `Order` object for the given ID is retrieved. By using the shortcut function `get_object_or_404()`, an `Http404` (page not found) exception is raised if no order is found with the given ID.
3. If the view is loaded with a `GET` request, the template `payment/process.html` is rendered and returned. This template will include the order summary and a button to proceed with the payment, which will generate a `POST` request to the view.

4. Alternatively, if the view is loaded with a `POST` request, a Stripe checkout session is created with `stripe.checkout.Session.create()` using the following parameters:
    - `mode`: The mode of the checkout session. We use `payment` for a one-time payment. You can see the different values accepted for this parameter at https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-mode.
    - `client_reference_id`: The unique reference for this payment. We will use this to reconcile the Stripe checkout session with our order. By passing the order ID, we link Stripe payments to orders in our system and we will be able to receive payment notifications from Stripe to mark the orders as paid.
    - `success_url`: The URL for Stripe to redirect the user to if the payment is successful. We use `request.build_absolute_uri()` to generate an absolute URI from the URL path. You can see the documentation for this method at https://docs.djangoproject.com/en/5.0/ref/request-response/#django.http.HttpRequest.build_absolute_uri.
    - `cancel_url`: The URL for Stripe to redirect the user to if the payment is canceled.
    - `line_items`: This is an empty list. We will next populate it with the order items to be purchased.
5. After creating the checkout session, an HTTP redirect with status code `303` is returned to redirect the user to Stripe. The status code `303` is

recommended to redirect web applications to a new URI after an HTTP `POST` has been performed.

You can see all the parameters to create a Stripe `session` object at https://stripe.com/docs/api/checkout/sessions/create.

Let's populate the `line_items` list with the order items to create the checkout session. Each item will contain the name of the item, the amount to charge, the currency to use, and the quantity purchased.

Add the following code highlighted in bold to the `payment_process` view:

```
def payment_process(request):
    order_id = request.session.get('order_id')
    order = get_object_or_404(Order, id=order_id)
    if request.method == 'POST':
        success_url = request.build_absolute_uri(
            reverse('payment:completed')
        )
        cancel_url = request.build_absolute_uri(
            reverse('payment:canceled')
        )
        # Stripe checkout session data
        session_data = {
            'mode': 'payment',
            'success_url': success_url,
            'cancel_url': cancel_url,
            'line_items': []
        }
 # add order items to the Stripe checkout session
 for item in order.items.all():
            session_data['line_items'].append(
                {
 'price_data': {
 'unit_amount': int(item.price * Decimal('100')),
 'currency': 'usd',
 'product_data': {
 'name': item.product.name,
```

```
                            },
                        },
                'quantity': item.quantity,
                    }
                )
        # create Stripe checkout session
                session = stripe.checkout.Session.create(**session_data)
                # redirect to Stripe payment form
        return redirect(session.url, code=303)
            else:
                return render(request, 'payment/process.html', locals())
```

We use the following information for each item:

- `price_data`: Price-related information:
  - `unit_amount`: The amount in cents to be collected by the payment. This is a positive integer representing how much to charge in the smallest currency unit with no decimal places. For example, to charge $10.00, this would be `1000` (that is, 1,000 cents). The item price, `item.price`, is multiplied by `Decimal('100')` to obtain the value in cents, and then it is converted into an integer.
  - `currency`: The currency to use in the three-letter ISO format. We use `usd` for US dollars. You can see a list of supported currencies at https://stripe.com/docs/currencies.
- `product_data`: Product-related information:
  - `name`: The name of the product
- `quantity`: The number of units to purchase

The `payment_process` view is now ready. Let's create simple views for the payment success and cancel pages.

Add the following code to the `views.py` file of the `payment` application:

```
def payment_completed(request):
    return render(request, 'payment/completed.html')
def payment_canceled(request):
    return render(request, 'payment/canceled.html')
```

Create a new file inside the `payment` application directory and name it `urls.py`. Add the following code to it:

```
from django.urls import path
from . import views
app_name = 'payment'
urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('completed/', views.payment_completed, name='completed'
    path('canceled/', views.payment_canceled, name='canceled'),
]
```

These are the URLs for the payment workflow. We have included the following URL patterns:

- `process`: The view that displays the order summary to the user, creates the Stripe checkout session, and redirects the user to the Stripe-hosted payment form
- `completed`: The view for Stripe to redirect the user to if the payment is successful
- `canceled`: The view for Stripe to redirect the user to if the payment is canceled

Edit the main `urls.py` file of the `myshop` project and include the URL patterns for the `payment` application, as follows:

```
urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('payment/', include('payment.urls', namespace='payment'
    path('', include('shop.urls', namespace='shop')),
]
```

We have placed the new path before the `shop.urls` pattern to avoid an unintended pattern match with a pattern defined in `shop.urls`. Remember that Django runs through each URL pattern in order and stops at the first one that matches the requested URL.

Let's build a template for each view. Create the following file structure inside the `payment` application directory:

```
templates/
    payment/
        process.html
        completed.html
        canceled.html
```

Edit the `payment/process.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% load static %}
{% block title %}Pay your order{% endblock %}
{% block content %}
  <h1>Order summary</h1>
<table class="cart">
<thead>
<tr>
<th>Image</th>
<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>
<img src="{% if item.product.image %}{{ item.product.image.url }
            {% else %}{% static "img/no_image.png" %}{% endif %}
</td>
<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="4">Total</td>
<td class="num">${{ order.get_total_cost }}</td>
</tr>
</tbody>
</table>
<form action="{% url "payment:process" %}" method="post">
<input type="submit" value="Pay now">
    {% csrf_token %}
  </form>
{% endblock %}
```

This is the template to display the order summary to the user and allow the client to proceed with the payment. It includes a form and a **Pay now** button to submit it via `POST`. When the form is submitted, the `payment_process` view creates the Stripe checkout session and redirects the user to the Stripe-hosted payment form.

Edit the `payment/completed.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% block title %}Payment successful{% endblock %}
{% block content %}
  <h1>Your payment was successful</h1>
```

```
  <p>Your payment has been processed successfully.</p>
  {% endblock %}
```

This is the template for the page that the user is redirected to after a successful payment.

Edit the `payment/canceled.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% block title %}Payment canceled{% endblock %}
{% block content %}
  <h1>Your payment has not been processed</h1>
<p>There was a problem processing your payment.</p>
{% endblock %}
```

This is the template for the page that the user is redirected to when the payment is canceled.

We have implemented the necessary views to process payments, including their URL patterns and templates. It's time to try out the checkout process.

# Testing the checkout process

Execute the following command in the shell to start the RabbitMQ server with Docker:

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

This will run RabbitMQ on port `5672` and the web-based management interface on port `15672`.

Open another shell and start the Celery worker from your project directory with the following command:

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

Open one more shell and start the development server from your project directory with this command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/` in your browser, add some products to the shopping cart, and fill in the checkout form. Click the **Place order** button. The order will be persisted to the database, the order ID will be saved in the current session, and you will be redirected to the payment process page.

The payment process page will look as follows:

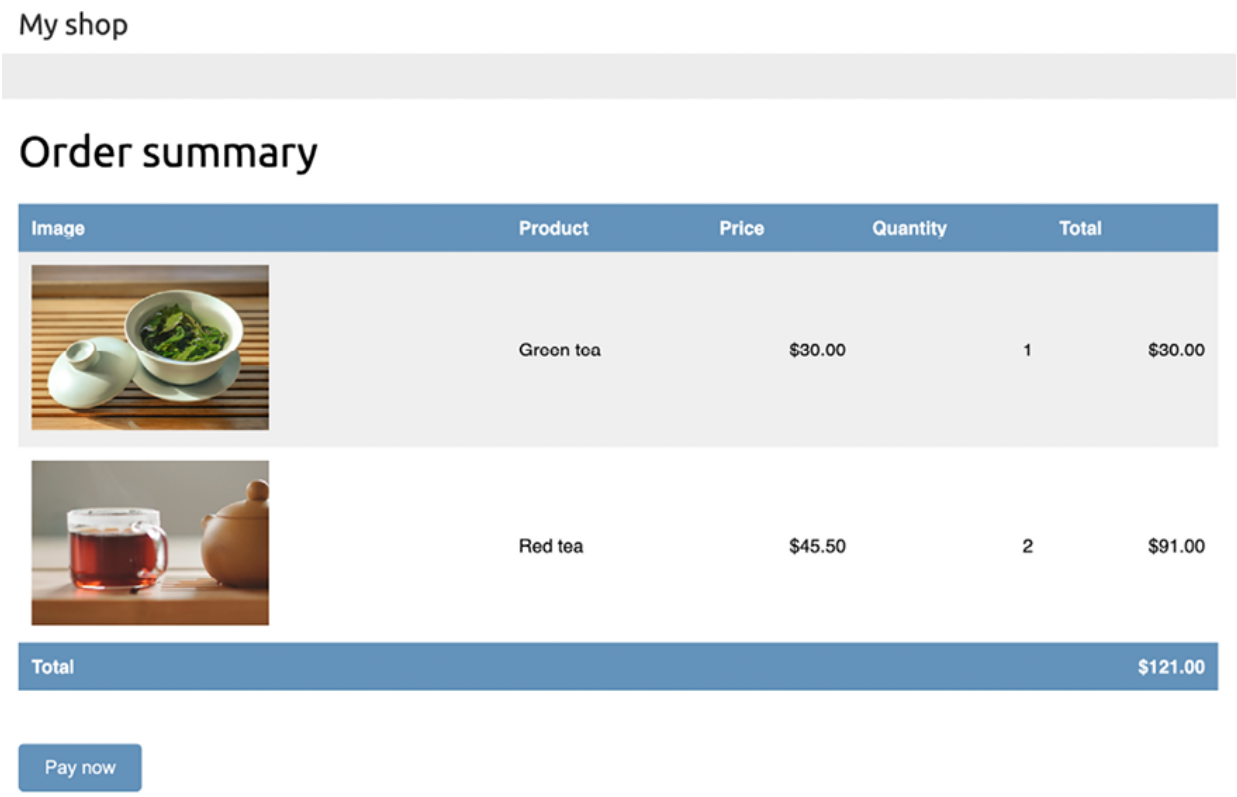



*Figure 9.9: The payment process page including an order summary*

Images in this chapter:

- *Green tea*: Photo by Jia Ye on Unsplash
- *Red tea*: Photo by Manki Kim on Unsplash

On this page, you can see an order summary and a **Pay now** button. Click on **Pay now**. The `payment_process` view will create a Stripe checkout session, and you will be redirected to the Stripe-hosted payment form.

You will see the following page:

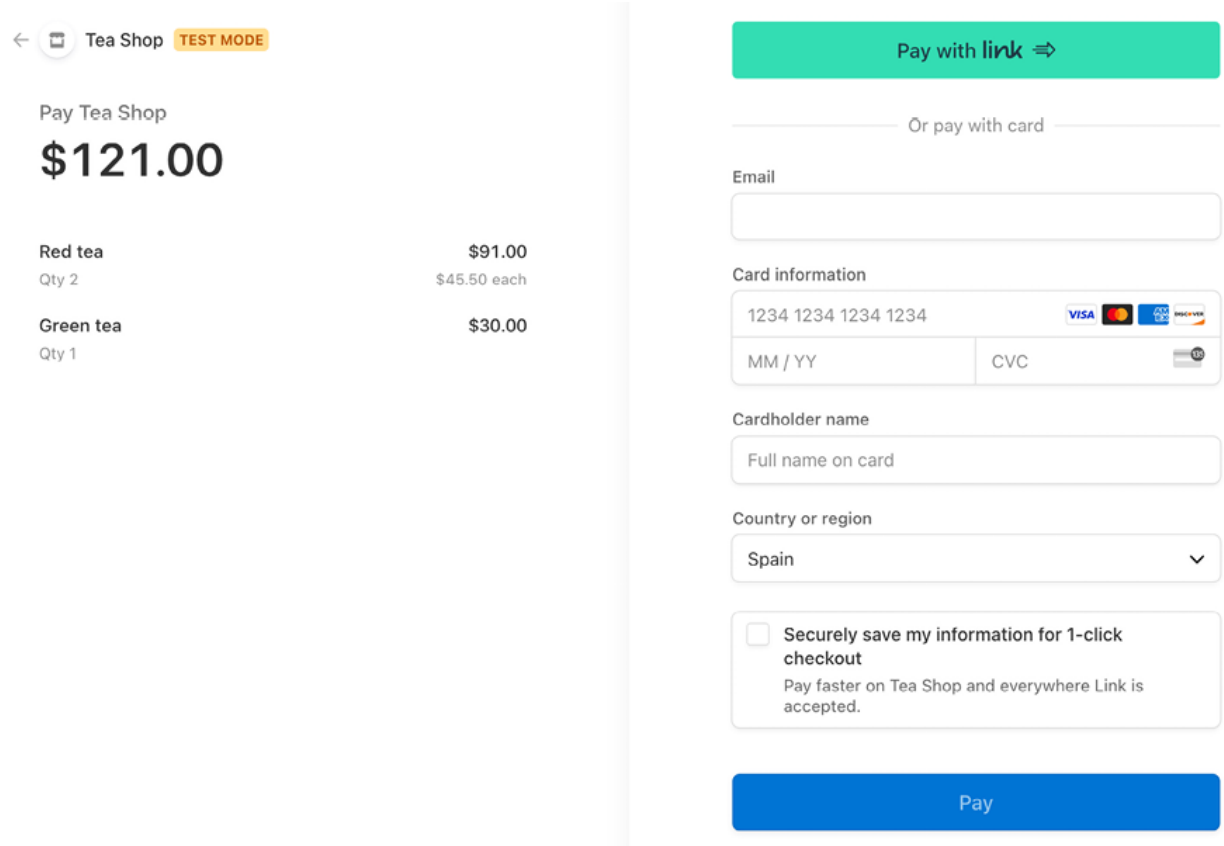



*Figure 9.10: The Stripe checkout payment from*

# Using test credit cards

Stripe provides different test credit cards from different card issuers and countries, which allows you to simulate payments to test all possible scenarios (successful payment, declined payment, etc.). The following table shows some of the cards you can test for different scenarios:

| **Result** | **Test Credit Card** | **CVC** | **Expiry date** |
|---|---|---|---|
| Successful payment | `4242 4242 4242 4242` | Any 3 digits | Any future date |
| Failed payment | `4000 0000 0000 0002` | Any 3 digits | Any future date |
| Requires 3D secure authentication | `4000 0025 0000 3155` | Any 3 digits | Any future date |

You can find the complete list of credit cards for testing at `https://stripe.com/docs/testing`.

We are going to use the test card `4242 4242 4242 4242`, which is a Visa card that returns a successful purchase. We will use the CVC `123` and any future expiration date, such as `12/29`. Enter the credit card details in the payment form as follows:

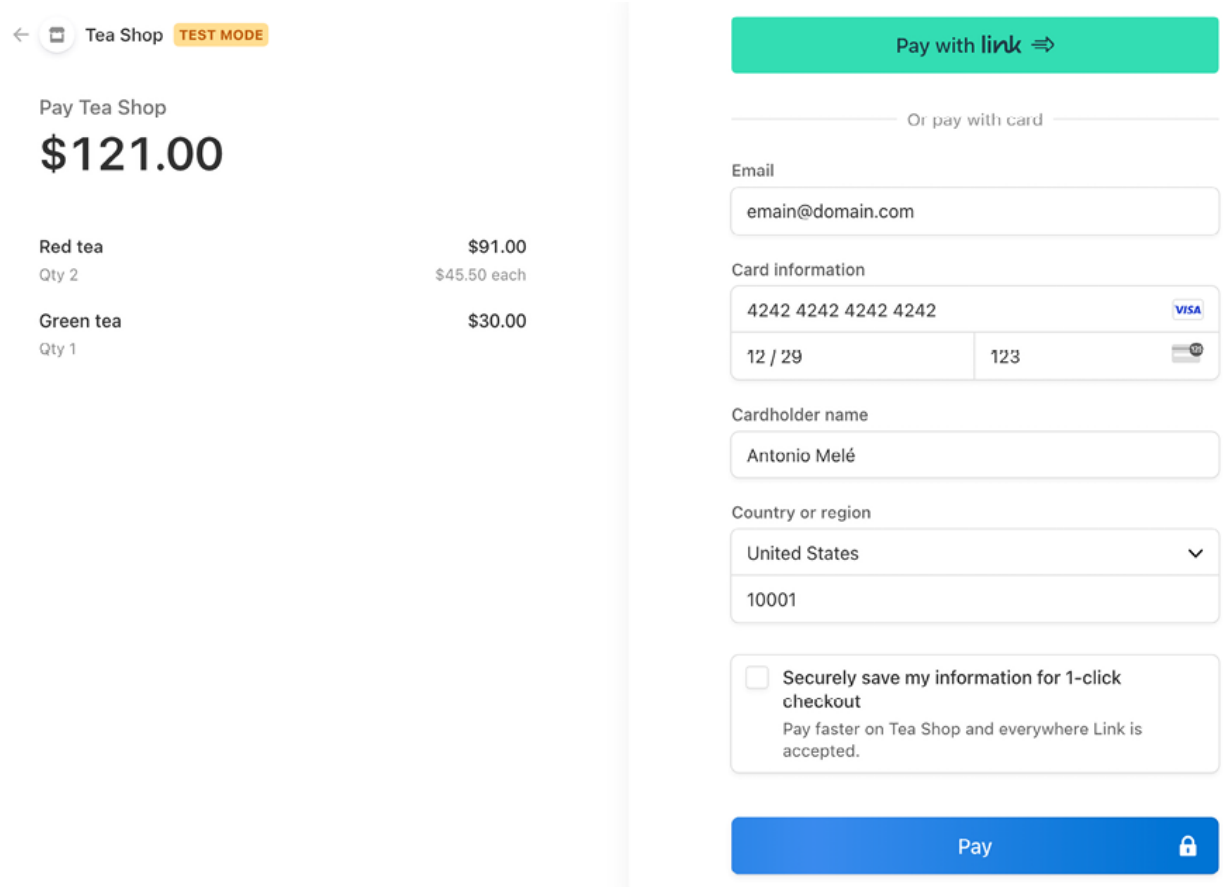



*Figure 9.11: The payment form with the valid test credit card details*

Click the **Pay** button. The button text will change to **Processing…**, as shown in *Figure 9.12*:

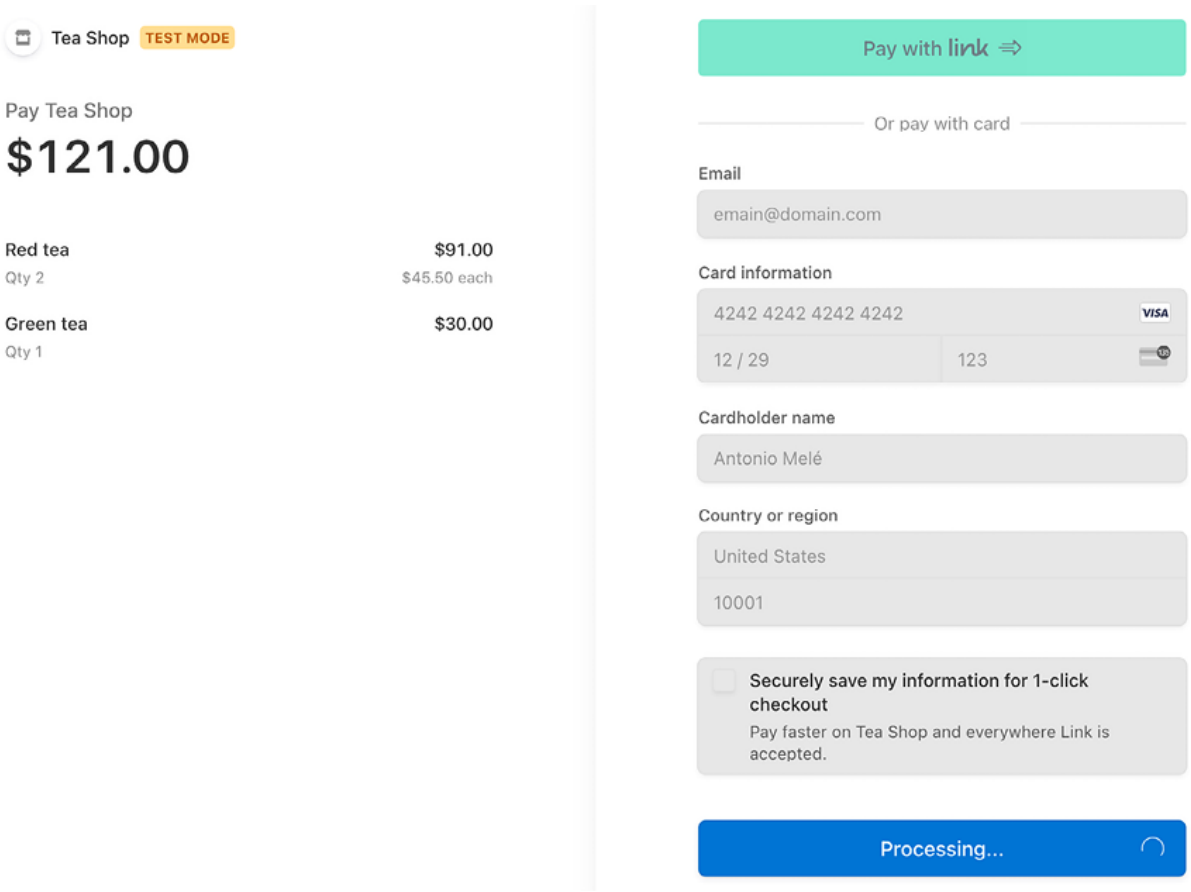



*Figure 9.12: The payment form being processed*

After a couple of seconds, you will see the button turn green, as in *Figure 9.13*:

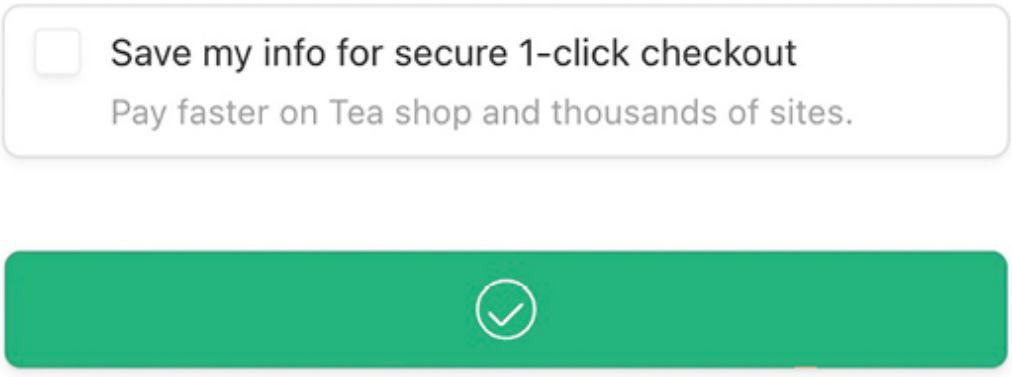



*Figure 9.13: The payment form after the payment is successful*

Then, Stripe redirects your browser to the payment completed URL you provided when creating the checkout session. You will see the following page:


*Figure 9.14: The successful payment page*

# Checking the payment information in the Stripe dashboard

Access the Stripe dashboard at https://dashboard.stripe.com/test/payments. Under **Payments**, you will be able to see the payment, as in *Figure 9.15*:

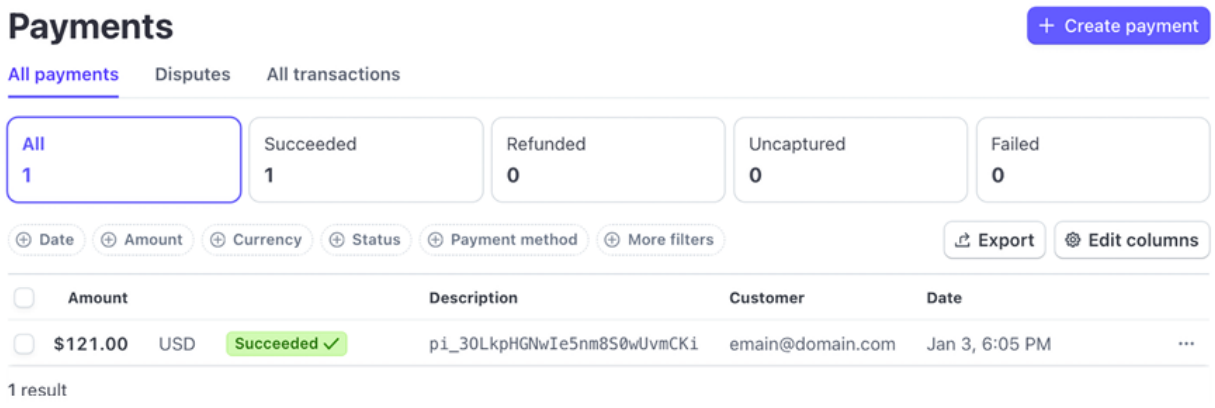



*Figure 9.15: The payment object with the status Succeeded in the Stripe dashboard*

The payment status is **Succeeded**. The payment description includes the **payment intent** ID that starts with `pi_`. When a checkout session is confirmed, Stripe creates a payment intent associated with the session. A payment intent is used to collect a payment from the user. Stripe records all attempted payments as payment intents. Each payment intent has a unique ID, and it encapsulates the details of the transaction, such as the supported payment methods, the amount to collect, and the desired currency. Click on the transaction to access the payment details.

You will see the following screen:

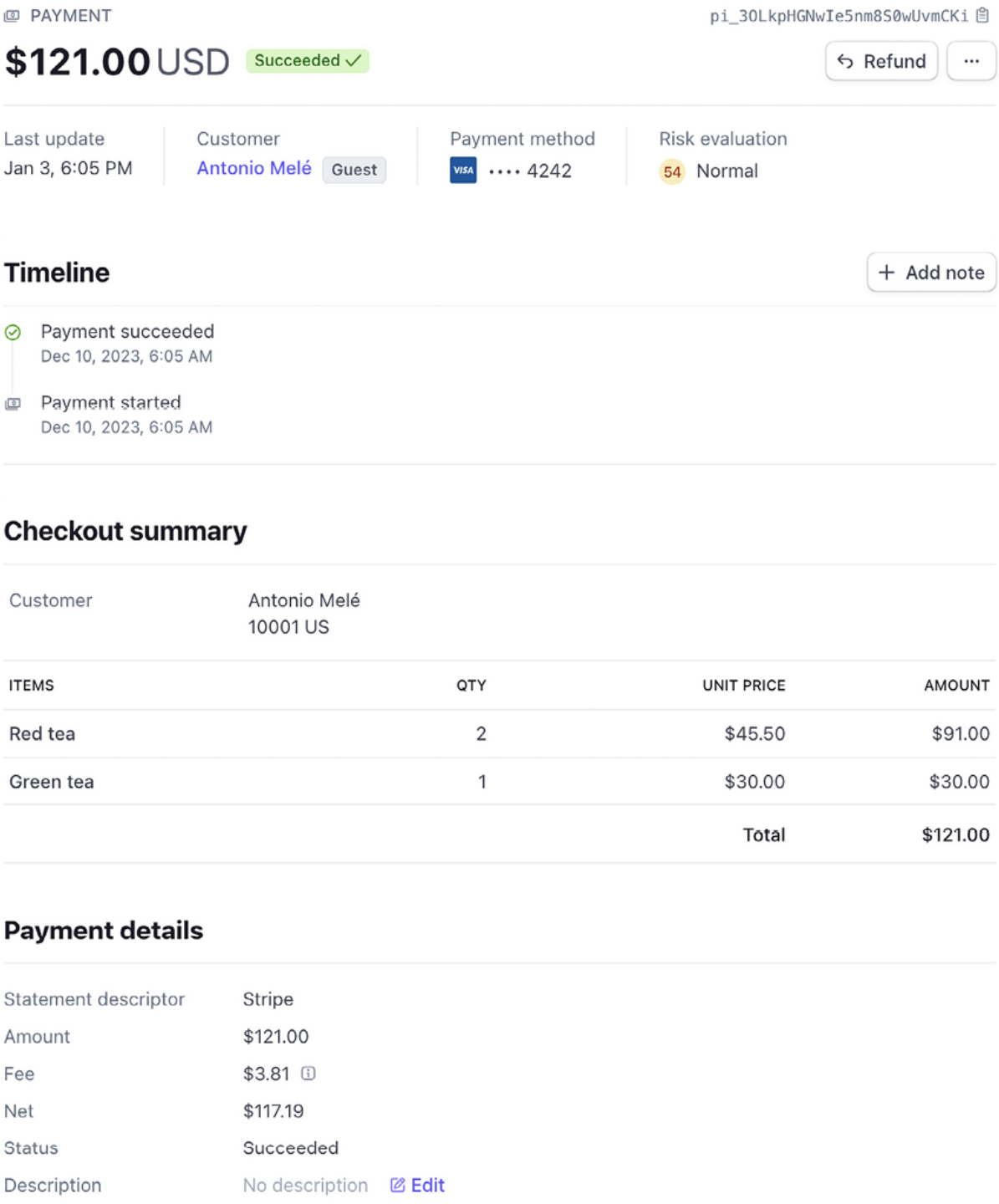



*Figure 9.16: Payment details for a Stripe transaction*

Here, you can see the payment information and the payment timeline, including payment changes. Under **Checkout summary**, you can find the

line items purchased, including the name, quantity, unit price, and amount.

Under **Payment details**, you can see a breakdown of the amount paid and the Stripe fee for processing the payment.

Under this section, you will find a **Payment method** section, including details about the payment method and the credit card checks performed by Stripe, as in *Figure 9.17*:

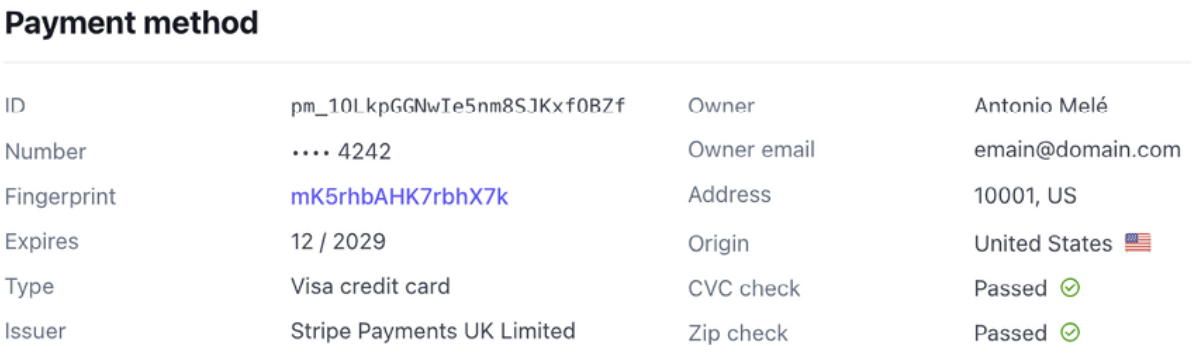



*Figure 9.17: Payment method used in the Stripe transaction*

Under this section, you will find another section named **Events and logs**, as in *Figure 9.18*:

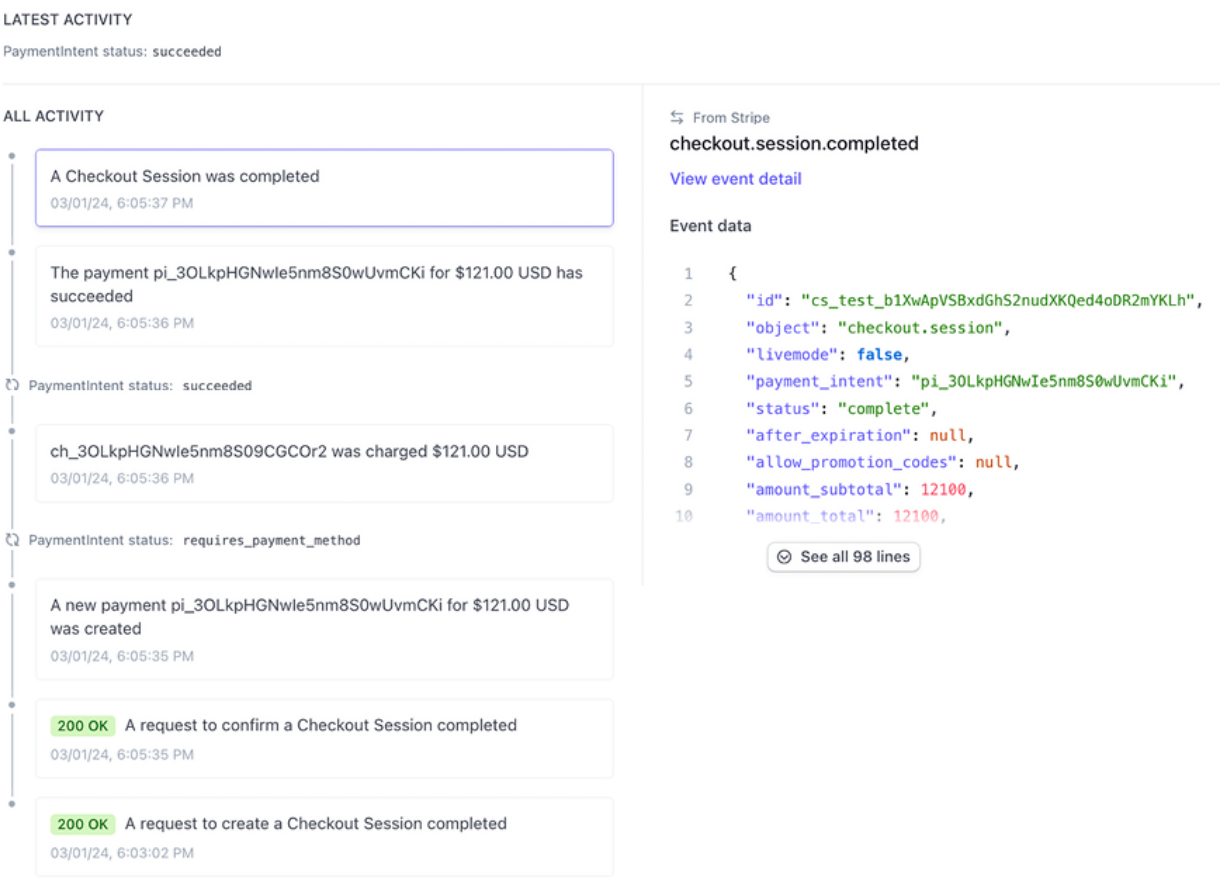



*Figure 9.18: Events and logs for a Stripe transaction*

This section contains all the activity related to the transaction, including requests to the Stripe API. You can click on any request to see the HTTP request to the Stripe API and the response in the JSON format.

Let's review the activity events in chronological order, from bottom to top:

1. First, a new checkout session is created by sending a `POST` request to the Stripe API endpoint `/v1/checkout/sessions`. The Stripe SDK method `stripe.checkout.Session.create()` that is used in the `payment_process` view builds and sends the request to the Stripe API, handling the response to return a `session` object.
2. The user is redirected to the checkout page where they submit the payment form. A request to confirm the checkout session is sent by the

Stripe checkout page.

3. A new payment intent is created.
4. A charge related to the payment intent is created.
5. The payment intent is now completed with a successful payment.
6. The checkout session is completed.

Congratulations! You have successfully integrated Stripe Checkout into your project. Next, you will learn how to receive payment notifications from Stripe and how to reference Stripe payments in your shop orders.

# Using webhooks to receive payment notifications

Stripe can push real-time events to our application by using webhooks. A **webhook**, also called a callback, can be thought of as an event-driven API instead of a request-driven API. Instead of polling the Stripe API frequently to know when a new payment is completed, Stripe can send an HTTP request to a URL of our application to notify us of successful payments in real time. The notification of these events will be asynchronous, when the event occurs, regardless of our synchronous calls to the Stripe API.

We will build a webhook endpoint to receive Stripe events. The webhook will consist of a view that will receive a JSON payload, with the event information to process it. We will use the event information to mark orders as paid when the checkout session is successfully completed.

## Creating a webhook endpoint

You can add webhook endpoint URLs to your Stripe account to receive events. Since we are using webhooks and we don’t have a hosted website

accessible through a public URL, we will use the Stripe **Command-Line Interface** (**CLI**) to listen to events and forward them to our local environment.

Open [https://dashboard.stripe.com/test/webhooks](https://dashboard.stripe.com/test/webhooks) in your browser. You will see the following screen:

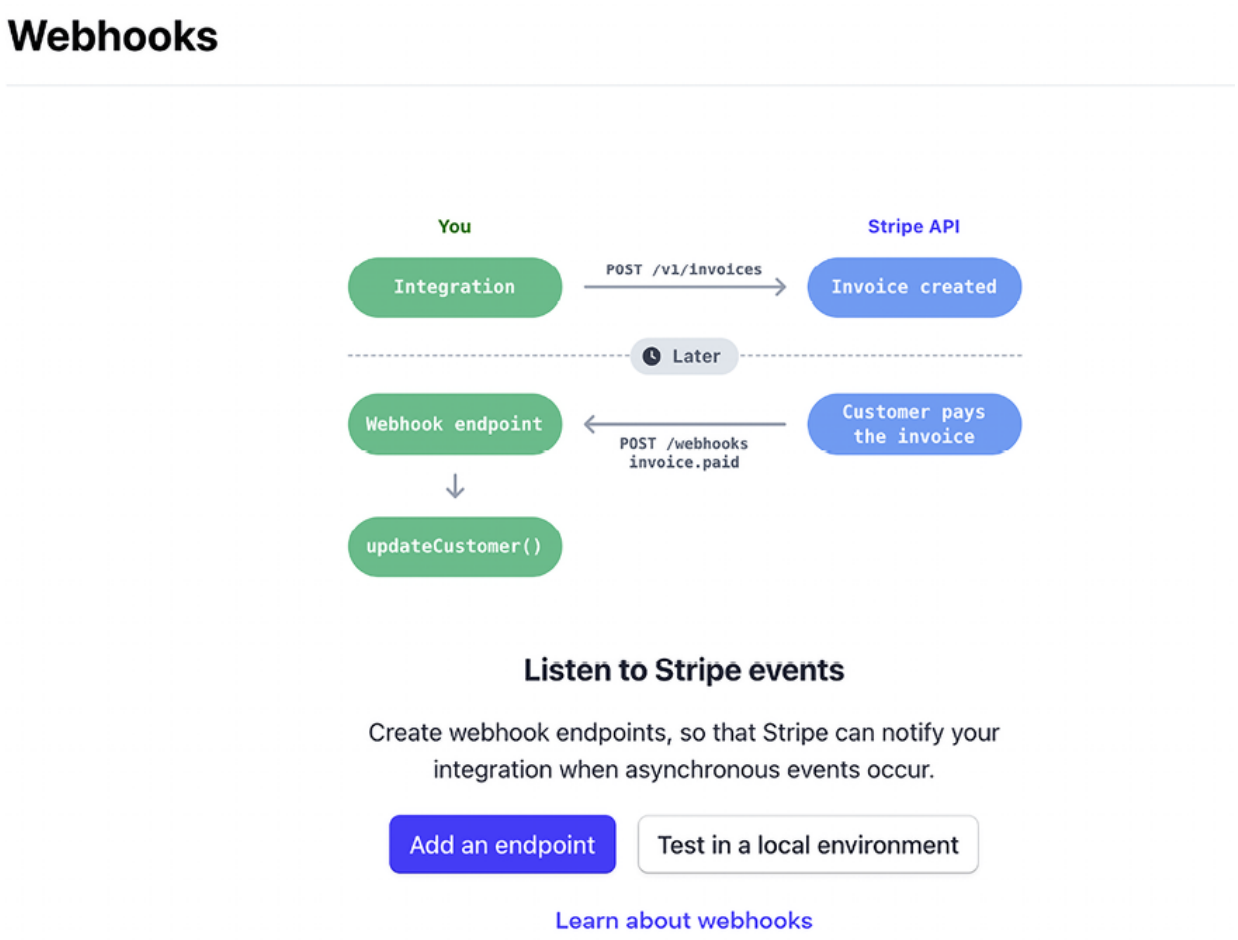



*Figure 9.19: The Stripe webhooks default screen*

Here, you can see a schema of how Stripe notifies your integration asynchronously. You will get Stripe notifications in real time whenever an event happens. Stripe sends different types of events, like checkout session created, payment intent created, payment intent updated, or checkout session

completed. You can find a list of all the types of events that Stripe sends at https://stripe.com/docs/api/events/types.

Click on **Test in a local environment**. You will see the following screen:

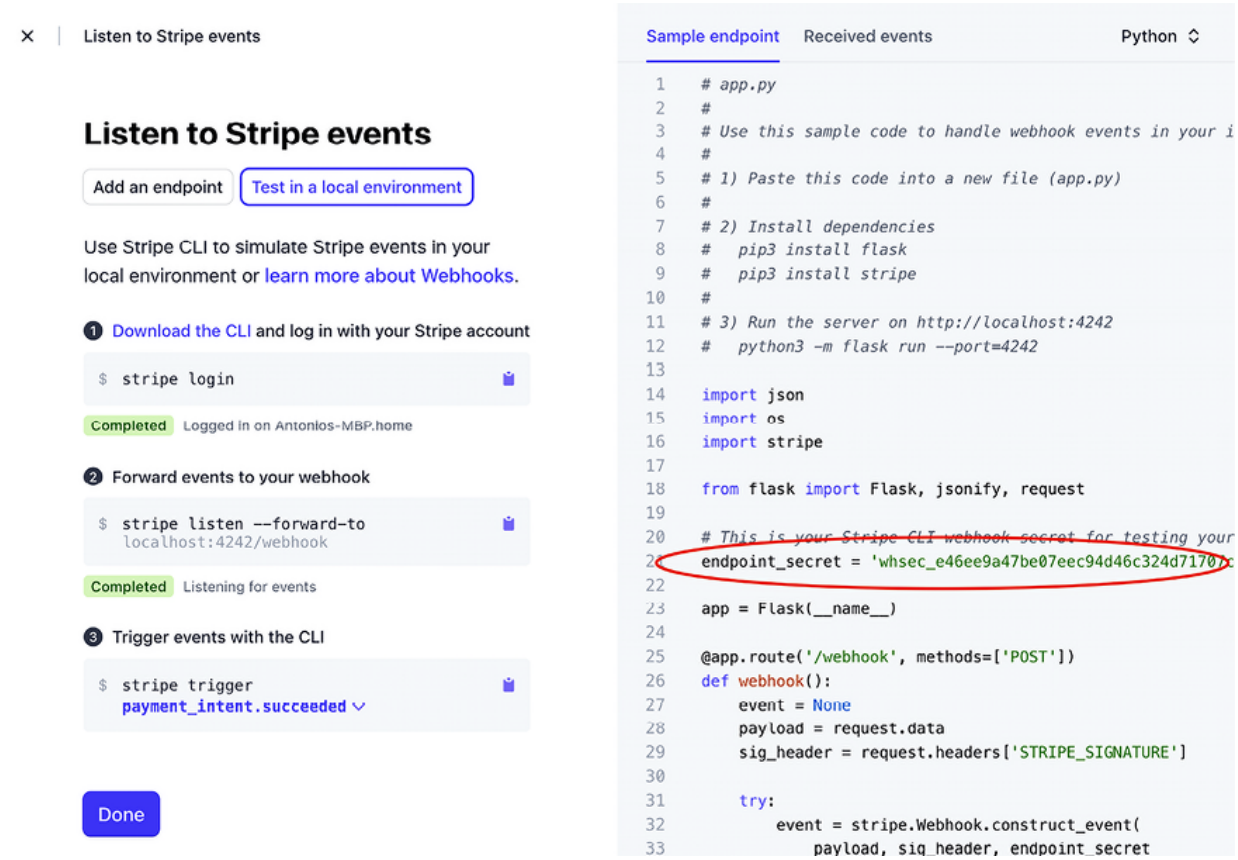



*Figure 9.20: The Stripe webhook setup screen*

This screen shows the steps to listen to Stripe events from your local environment. It also includes a sample Python webhook endpoint. Copy just the `endpoint_secret` value.

Edit the `.env` file of your project and add the following environment variable highlighted in bold:

```
STRIPE_PUBLISHABLE_KEY=pk_test_XXXX
STRIPE_SECRET_KEY=sk_test_XXXX
```

```
STRIPE_WEBHOOK_SECRET=whsec_XXXX
```

Replace the `STRIPE_WEBHOOK_SECRET` value with the `endpoint_secret` value provided by Stripe.

Edit the `settings.py` file of the `myshop` project and add the following setting to it:

```
# ...
STRIPE_PUBLISHABLE_KEY = config('STRIPE_PUBLISHABLE_KEY')
STRIPE_SECERT_KEY = config('STRIPE_SECRET_KEY')
STRIPE_API_VERSION = '2024-04-10'
STRIPE_WEBHOOK_SECRET = config('STRIPE_WEBHOOK_SECRET')
```

To build a webhook endpoint, we will create a view that receives a JSON payload with the event details. We will check the event details to identify when a checkout session is completed and mark the related order as paid.

Stripe signs the webhook events it sends to your endpoints by including a `Stripe-Signature` header, with a signature in each event. By checking the Stripe signature, you can verify that events were sent by Stripe and not by a third party. If you don't check the signature, an attacker could send fake events to your webhooks intentionally. The Stripe SDK provides a method to verify signatures. We will use it to create a webhook that verifies the signature.

Add a new file to the `payment/` application directory and name it `webhooks.py`. Add the following code to the new `webhooks.py` file:

```
import stripe
from django.conf import settings
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from orders.models import Order
```

```
@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None
try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, settings.STRIPE_WEBHOOK_SECRET
        )
    except ValueError as e:
        # Invalid payload
return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
return HttpResponse(status=400)
    return HttpResponse(status=200)
```

The `@csrf_exempt` decorator is used to prevent Django from performing the **cross-site request forgery** (**CSRF**) validation that is done by default for all `POST` requests. We use the method `stripe.Webhook.construct_event()` of the `stripe` library to verify the event's signature header. If the event's payload or the signature is invalid, we return an HTTP `400 Bad Request` response. Otherwise, we return an HTTP `200 OK` response.

This is the basic functionality required to verify the signature and construct the event from the JSON payload. Now, we can implement the actions of the webhook endpoint.

Add the following code highlighted in bold to the `stripe_webhook` view:

```
@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None
try:
        event = stripe.Webhook.construct_event(
```

```
                payload, sig_header, settings.STRIPE_WEBHOOK_SECRET
            )
        except ValueError as e:
            # Invalid payload
    return HttpResponse(status=400)
        except stripe.error.SignatureVerificationError as e:
            # Invalid signature
    return HttpResponse(status=400)
     if event.type == 'checkout.session.completed':
            session = event.data.object
     if (
                session.mode == 'payment'
     and session.payment_status == 'paid'
            ):
     try:
                    order = Order.objects.get(
     id=session.client_reference_id
                    )
     except Order.DoesNotExist:
     return HttpResponse(status=404)
     # mark order as paid
                order.paid = True
                order.save()
    return HttpResponse(status=200)
```

In the new code, we check whether the event received is `checkout.session.completed`. This event indicates that the checkout session has been successfully completed. If we receive this event, we retrieve the `session` object and check whether the session `mode` is `payment` because this is the expected mode for one-off payments.

Then, we get the `client_reference_id` attribute that we used when we created the checkout session and use the Django ORM to retrieve the `Order` object with the given `id`. If the order does not exist, we raise an HTTP `404` exception. Otherwise, we mark the order as paid with `order.paid = True`, and we save the order in the database.

Edit the `urls.py` file of the `payment` application and add the following code highlighted in bold:

```
from django.urls import path
from . import views, webhooks
app_name = 'payment'
urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('completed/', views.payment_completed, name='completed'
    path('canceled/', views.payment_canceled, name='canceled'),
    path('webhook/', webhooks.stripe_webhook, name='stripe-webho
]
```

We have imported the `webhooks` module and added the URL pattern for the Stripe webhook.

# Testing webhook notifications

To test webhooks, you need to install the Stripe CLI. The Stripe CLI is a developer tool that allows you to test and manage your integration with Stripe directly from your shell. You will find installation instructions at https://stripe.com/docs/stripe-cli#install.

If you are using macOS or Linux, you can install the Stripe CLI with Homebrew using the following command:

```
brew install stripe/stripe-cli/stripe
```

If you are using Windows, or you are using macOS or Linux without Homebrew, download the latest Stripe CLI release for macOS, Linux, or Windows from https://github.com/stripe/stripe-

cli/releases/latest and unzip the file. If you are using Windows, run the unzipped `.exe` file.

After installing the Stripe CLI, run the following command from a shell:

```
stripe login
```

You will see the following output:

```
Your pairing code is: xxxx-yyyy-zzzz-oooo This pairing code verif
```

Press *Enter* or open the URL in your browser. You will see the following screen:

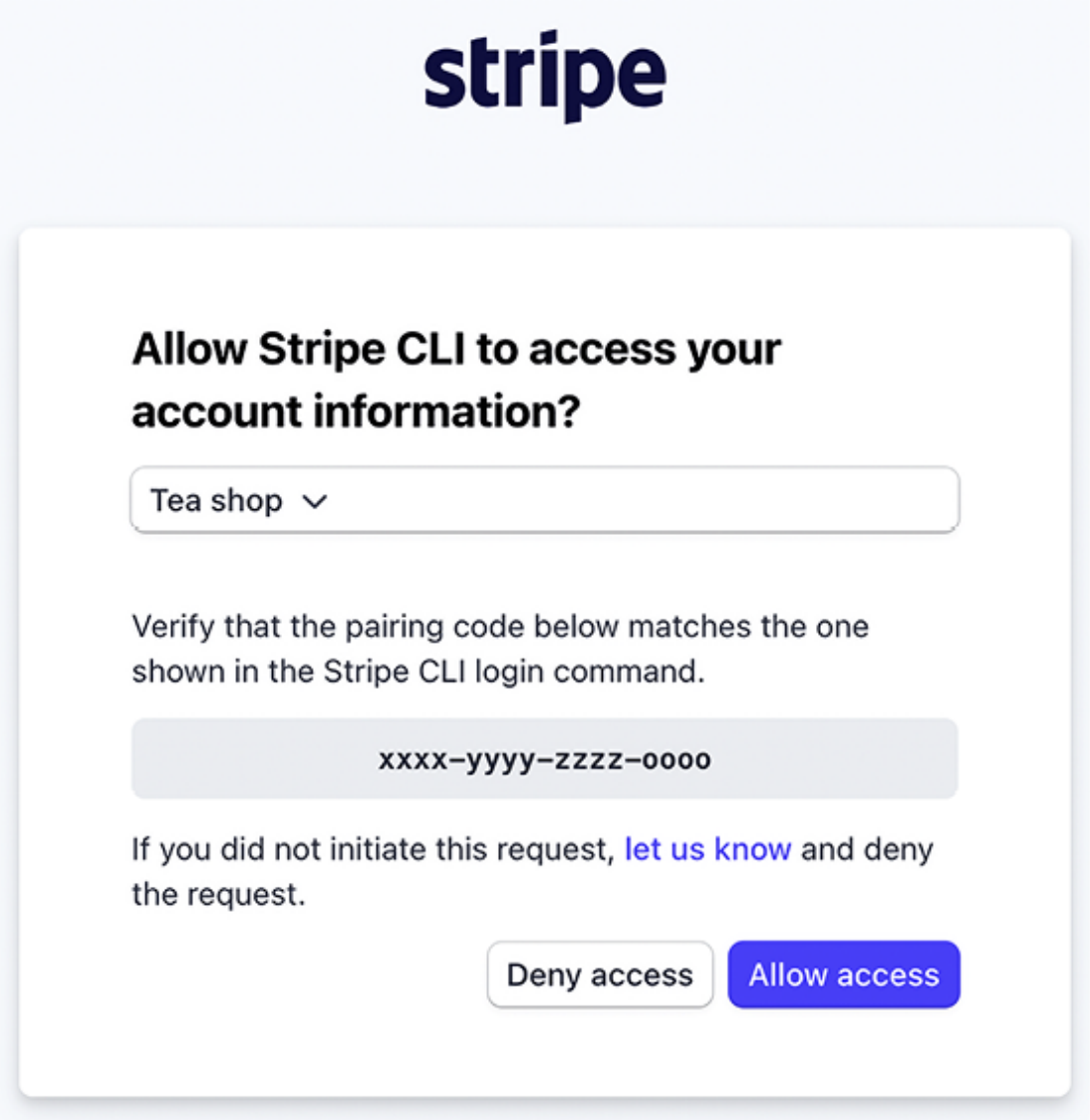



*Figure 9.21: The Stripe CLI pairing screen*

Verify that the pairing code in the Stripe CLI matches the one shown on the website and click on **Allow access**. You will see the following message:

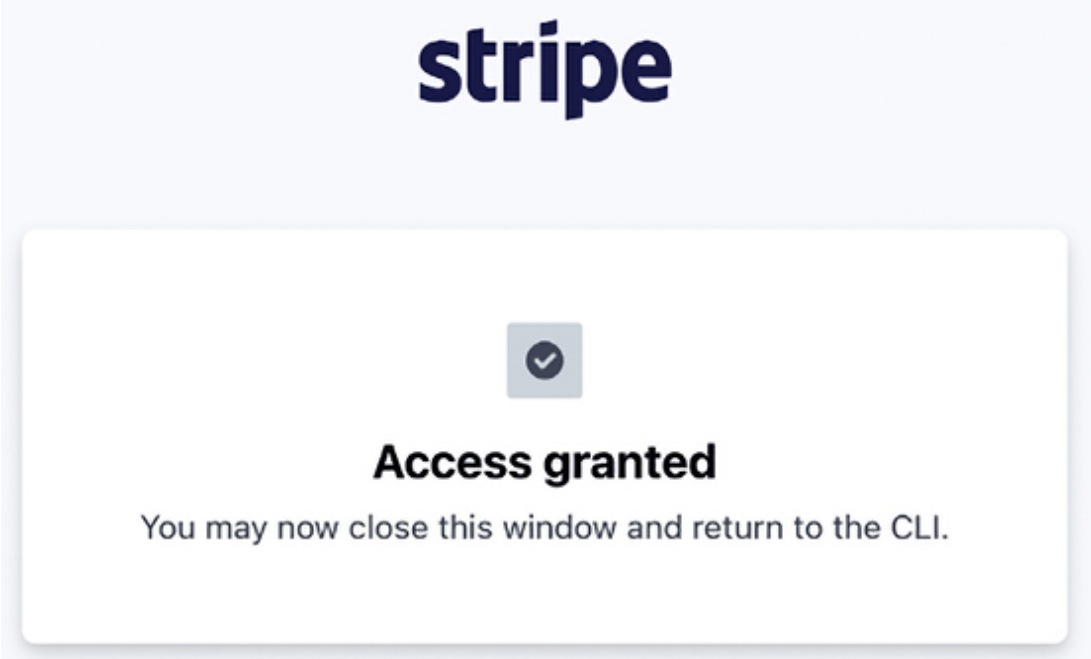



*Figure 9.22: The Stripe CLI pairing confirmation*

Now, run the following command from your shell:

```
stripe listen --forward-to 127.0.0.1:8000/payment/webhook/
```

We use this command to tell Stripe to listen to events and forward them to our localhost. We use port `8000`, where the Django development server is running, and the path `/payment/webhook/`, which matches the URL pattern of our webhook.

You will see the following output:

```
Getting ready... > Ready! You are using Stripe API Version [2024·
```

Here, you can see the webhook secret. Check that the webhook signing secret matches the `STRIPE_WEBHOOK_SECRET` setting in the `settings.py` file of your project.

Open https://dashboard.stripe.com/test/webhooks in your browser. You will see the following screen:

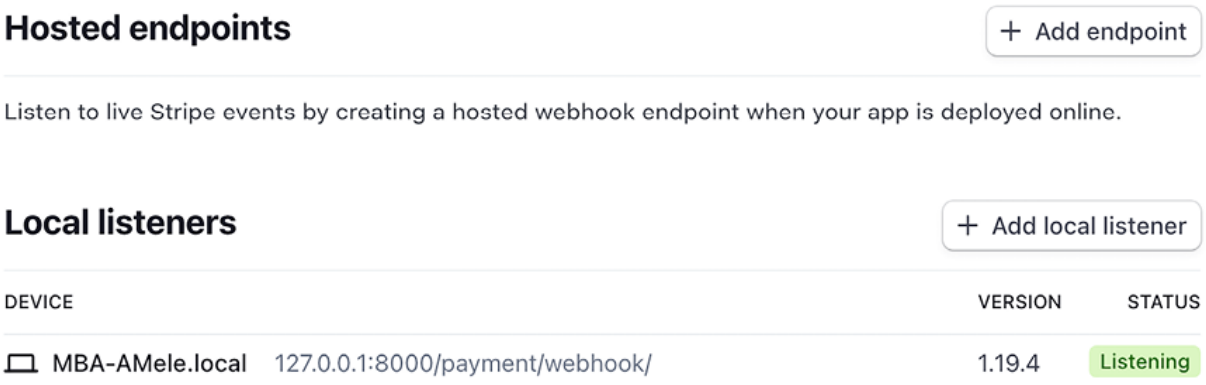



*Figure 9.23: The Stripe Webhooks page*

Under **Local listeners**, you will see the local listener that we created.

In a production environment, the Stripe CLI is not needed. Instead, you would need to add a hosted webhook endpoint using the URL of your hosted application.

Open `http://127.0.0.1:8000/` in your browser, add some products to the shopping cart, and complete the checkout process.

Check the shell where you are running the Stripe CLI:

```
2024-01-03 18:06:13   --> payment_intent.created [evt_...]
2024-01-03 18:06:13  <--  [200] POST http://127.0.0.1:8000/payme
2024-01-03 18:06:13   --> payment_intent.succeeded [evt_...]
2024-01-03 18:06:13  <--  [200] POST http://127.0.0.1:8000/payme
2024-01-03 18:06:13   --> charge.succeeded [evt_...]
2024-01-03 18:06:13  <--  [200] POST http://127.0.0.1:8000/payme
2024-01-03 18:06:14   --> checkout.session.completed [evt_...]
2024-01-03 18:06:14  <--  [200] POST http://127.0.0.1:8000/payme
```

You can see the different events that have been sent by Stripe to the local webhook endpoint. The events might be in a different order than above. Stripe doesn't guarantee the delivery of events in the order in which they are generated. Let's review the events:

- `payment_intent.created`: The payment intent has been created.
- `payment_intent.succeeded`: The payment intent succeeded.
- `charge.succeeded`: The charge associated with the payment intent succeeded.
- `checkout.session.completed`: The checkout session has been completed. This is the event that we use to mark the order as paid.

The `stripe_webhook` webhook returns an HTTP `200 OK` response to all of the requests sent by Stripe. However, we only process the event `checkout.session.completed` to mark the order related to the payment as paid.

Next, open `http://127.0.0.1:8000/admin/orders/order/` in your browser. The order should now be marked as paid:

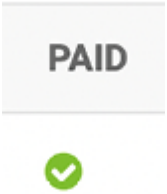



*Figure 9.24: An order marked as paid in the order list of the administration site*

Now, orders get automatically marked as paid with Stripe payment notifications. Next, you are going to learn how to reference Stripe payments in your shop orders.

# Referencing Stripe payments in orders

Each Stripe payment has a unique identifier. We can use the payment ID to associate each order with its corresponding Stripe payment. We will add a new field to the `Order` model of the `orders` application so that we can reference the related payment by its ID. This will allow us to link each order with the related Stripe transaction.

Edit the `models.py` file of the `orders` application and add the following field to the `Order` model. The new field is highlighted in bold:

```
class Order(models.Model):
    # ...
    stripe_id = models.CharField(max_length=250, blank=True)
```

Let's sync this field with the database. Use the following command to generate the database migrations for the project:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'orders':
  orders/migrations/0002_order_stripe_id.py
    - Add field stripe_id to order
```

Apply the migration to the database with the following command:

```
python manage.py migrate
```

You will see output that ends with the following line:

```
Applying orders.0002_order_stripe_id... OK
```

The model changes are now synced with the database. Now, you will be able to store the Stripe payment ID for each order.

Edit the `stripe_webhook` function in the `webhooks.py` file of the payment application and add the following lines highlighted in bold:

```
# ...
@csrf_exempt
def stripe_webhook(request):
    # ...
if event.type == 'checkout.session.completed':
        session = event.data.object
if (
            session.mode == 'payment'
and session.payment_status == 'paid'
        ):
            try:
                order = Order.objects.get(
                    id=session.client_reference_id
                )
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # mark order as paid
            order.paid = True
 # store Stripe payment ID
            order.stripe_id = session.payment_intent
            order.save()
    return HttpResponse(status=200)
```

With this change, when receiving a webhook notification for a completed checkout session, the payment intent ID is stored in the `stripe_id` field of the `Order` object.

Open `http://127.0.0.1:8000/` in your browser, add some products to the shopping cart, and complete the checkout process. Then, access `http://127.0.0.1:8000/admin/orders/order/` in your browser and click on the latest order ID to edit it. The `stripe_id` field should contain the payment intent ID, as shown in *Figure 9.25*:


*Figure 9.25: The Stripe id field with the payment intent ID*

Great! We have successfully referenced Stripe payments in orders. Now, we can add Stripe payment IDs to the order list on the administration site. We can also include a link to each payment ID to see the payment details in the Stripe dashboard.

Edit the `models.py` file of the `orders` application and add the following code highlighted in bold:

```
from django.conf import settings
from django.db import models
class Order(models.Model):
    # ...
class Meta:
        # ...
def __str__(self):
        return f'Order {self.id}'
def get_total_cost(self):
        return sum(item.get_cost() for item in self.items.all())
 def get_stripe_url(self):
 if not self.stripe_id:
 # no payment associated
 return ''
 if '_test_' in settings.STRIPE_SECRET_KEY:
 # Stripe path for test payments
```

```
            path = '/test/'
    else:
    # Stripe path for real payments
            path = '/'
    return f'https://dashboard.stripe.com{path}payments/{self.strip
```

We have added the new `get_stripe_url()` method to the `Order` model. This method is used to return the Stripe dashboard's URL for the payment associated with the order. If no payment ID is stored in the `stripe_id` field of the `Order` object, an empty string is returned. Otherwise, the URL for the payment in the Stripe dashboard is returned. We check if the string `_test_` is present in the `STRIPE_SECRET_KEY` setting to discriminate the production environment from the test environment. Payments in the production environment follow the pattern `https://dashboard.stripe.com/payments/{id}`, whereas test payments follow the pattern `https://dashboard.stripe.com/payments/test/{id}`.

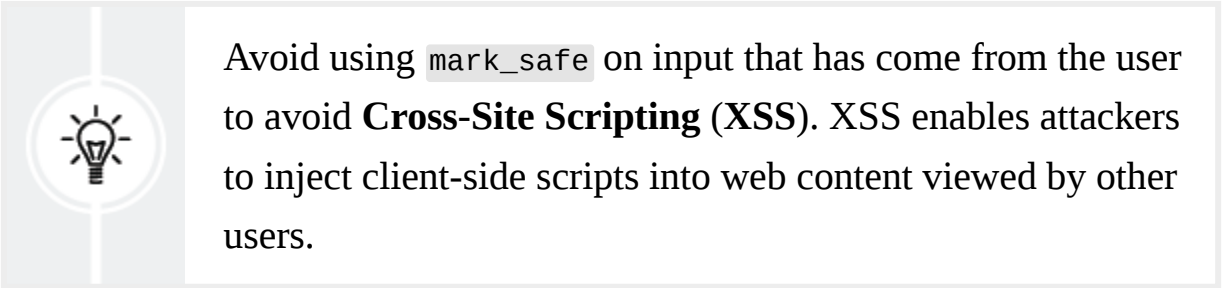
Let's add a link to each `Order` object on the list display page of the administration site.

Edit the `admin.py` file of the `orders` application and add the following code highlighted in bold:

```
# ...
from django.utils.safestring import mark_safe
def order_payment(obj):
    url = obj.get_stripe_url()
    if obj.stripe_id:
        html = f'<a href="{url}" target="_blank">{obj.stripe_id}
    return mark_safe(html)
    return ''
order_payment.short_description = 'Stripe payment'
@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = [
```

```
            'id',
            'first_name',
            'last_name',
            'email',
            'address',
            'postal_code',
            'city',
            'paid',
            order_payment,
    'created',
            'updated'
        ]
        # ...
```

The `order_stripe_payment()` function takes an `Order` object as an argument and returns an HTML link with the payment URL in Stripe. Django escapes HTML output by default. We use the `mark_safe` function to avoid auto-escaping.



Avoid using `mark_safe` on input that has come from the user to avoid **Cross-Site Scripting** (**XSS**). XSS enables attackers to inject client-side scripts into web content viewed by other users.

Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. You will see a new column named **STRIPE PAYMENT**. You will see the related Stripe payment ID for the latest order. If you click on the payment ID, you will be taken to the payment URL in Stripe, where you can find the additional payment details.

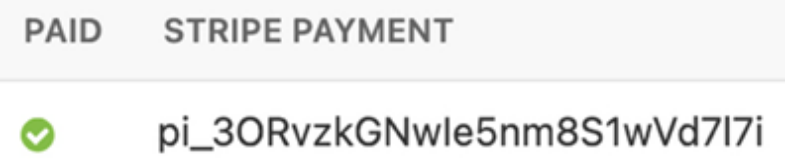



*Figure 9.26: The Stripe payment ID for an Order object in the administration site*

Now, you automatically store Stripe payment IDs in orders when receiving payment notifications. You have successfully integrated Stripe into your project.

# Going live

Once you have tested your integration, you can apply for a production Stripe account. When you are ready to move into production, remember to replace your test Stripe credentials with the live ones in the `settings.py` file. You will also need to add a webhook endpoint for your hosted website at https://dashboard.stripe.com/webhooks instead of using the Stripe CLI. *Chapter 17*, *Going Live*, will teach you how to configure project settings for multiple environments.

# Exporting orders to CSV files

Sometimes, you might want to export the information contained in a model to a file so that you can import it into another system. One of the most widely used formats to export/import data is the **Comma-Separated Values** (**CSV**) format. A CSV file is a plain text file consisting of a number of records. There is usually one record per line and some delimiter character, usually a literal comma, separating the record fields. We are going to customize the administration site to be able to export orders to CSV files.

# Adding custom actions to the administration site

Django offers a wide range of options to customize the administration site. You are going to modify the object list view to include a custom administration action. You can implement custom administration actions to allow staff users to apply actions to multiple elements at once in the change list view.

An administration action works as follows: a user selects objects from the administration object list page with checkboxes, selects an action to perform on all of the selected items, and then executes the actions. *Figure 9.27* shows where the actions are located on the administration site:

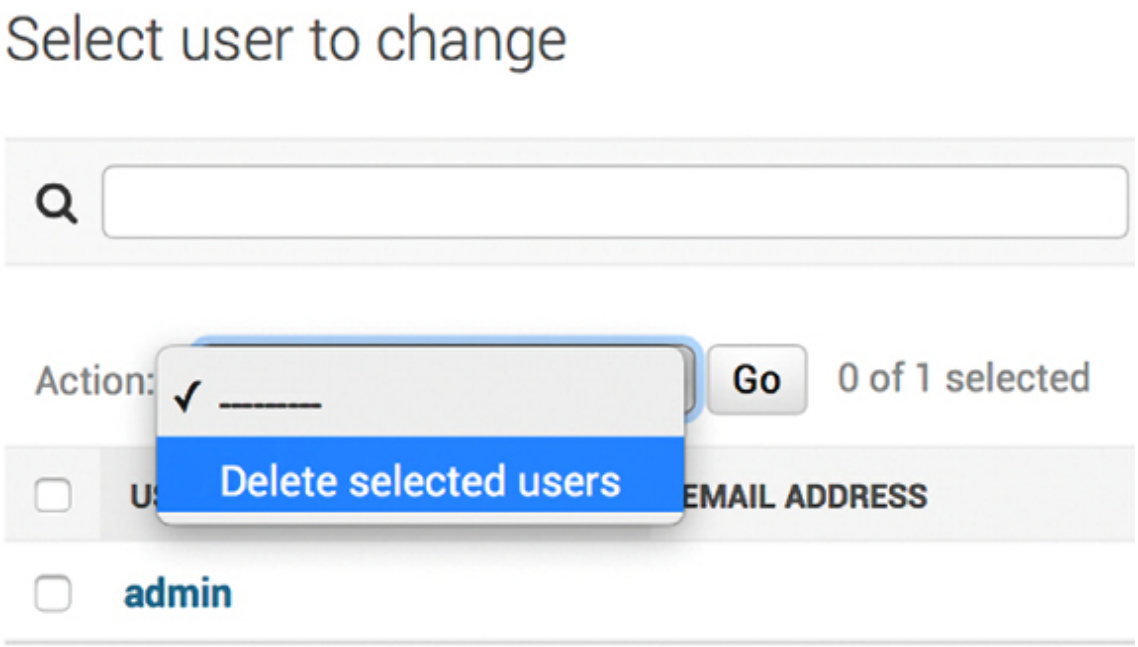



*Figure 9.27: The drop-down menu for Django administration actions*

You can create a custom action by writing a regular function that receives the following parameters:

- The current `ModelAdmin` being displayed
- The current request object as an `HttpRequest` instance
- A QuerySet for the objects selected by the user

This function will be executed when the action is triggered from the administration site.

You are going to create a custom administration action to download a list of orders as a CSV file.

Edit the `admin.py` file of the `orders` application and add the following code before the `OrderAdmin` class:

```
import csv
import datetime
from django.http import HttpResponse
def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    content_disposition = (
        f'attachment; filename={opts.verbose_name}.csv'
 )
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = content_disposition
    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'
```

In this code, you perform the following tasks:

1. You create an instance of `HttpResponse`, specifying the `text/csv` content type, to tell the browser that the response has to be treated as a CSV file. You also add a `Content-Disposition` header to indicate that the HTTP response contains an attached file.
2. You create a CSV `writer` object that will write to the `response` object.
3. You get the `model` fields dynamically using the `get_fields()` method of the model's `_meta` options. You exclude many-to-many and one-to-many relationships.
4. You write a header row, including the field names.
5. You iterate over the given QuerySet and write a row for each object returned by the QuerySet. You take care of formatting `datetime` objects because the output value for CSV has to be a string.
6. You customize the display name for the action in the action's drop-down element of the administration site by setting a `short_description` attribute on the function.

You have created a generic administration action that can be added to any `ModelAdmin` class.

Finally, add the new `export_to_csv` administration action to the `OrderAdmin` class, as follows. The new code is highlighted in bold:

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

Start the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. The resulting administration action should look like this:

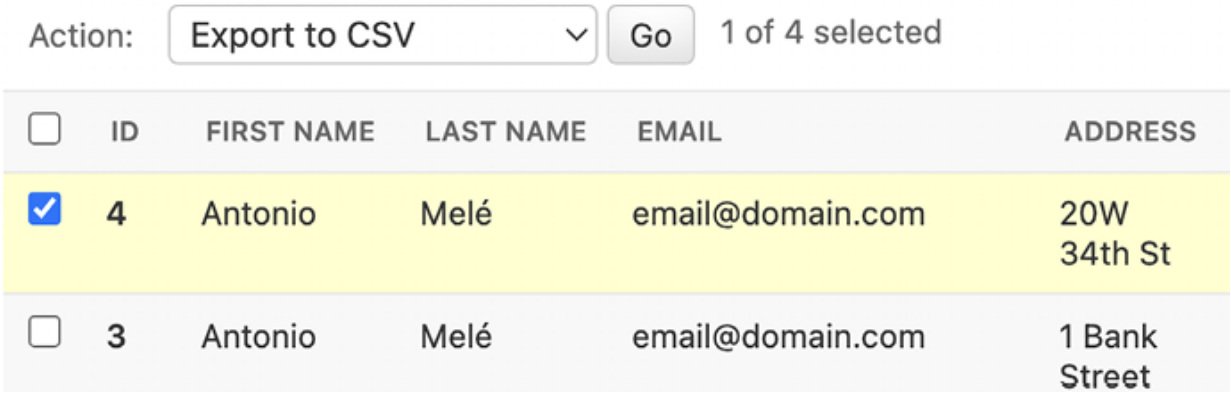



*Figure 9.28: Using the custom Export to CSV administration action*

Select some orders, choose the **Export to CSV** action from the select box, and then click the **Go** button. Your browser will download the generated CSV file named `order.csv`. Open the downloaded file using a text editor. You should see content with the following format, including a header row and a row for each `Order` object you selected:

```
ID,first name,last name,email,address,postal code,city,created,u
4,Antonio,Melé,email@domain.com,20 W 34th St,10001,New York,03/0
...
```

As you can see, creating administration actions is pretty straightforward. You can learn more about generating CSV files with Django at

https://docs.djangoproject.com/en/5.0/howto/outputting-csv/.

If you want to add more advanced import/export functionalities to your administration site, you can use the third-party application `django-import-export`. You can find its documentation at https://django-import-export.readthedocs.io/en/latest/.

The example we have implemented works well for small to medium datasets. Given that the export occurs within an HTTP request, very large datasets could lead to server timeouts if the server closes the connection before the export process concludes. To circumvent this, you can generate exports asynchronously using Celery, with the `django-import-export-celery` application. This project is available at https://github.com/auto-mat/django-import-export-celery.

Next, you are going to customize the administration site further by creating a custom administration view.

# Extending the administration site with custom views

Sometimes, you may want to customize the administration site beyond what is possible by configuring `ModelAdmin`, creating administration actions, and overriding administration templates. You might want to implement additional functionalities that are not available in existing administration views or templates. If this is the case, you need to create a custom administration view. With a custom view, you can build any functionality you want; you just have to make sure that only staff users can access your

view and that you maintain the administration look and feel by making your template extend an administration template.

Let's create a custom view to display information about an order. Edit the `views.py` file of the `orders` application and add the following code highlighted in bold:

```
from django.contrib.admin.views.decorators import staff_member_r
from django.shortcuts import get_object_or_404, redirect, render
from cart.cart import Cart
from .forms import OrderCreateForm
from .models import Order, OrderItem
from .tasks import order_created
def order_create(request):
    # ...
@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': ord
    )
```

The `staff_member_required` decorator checks that both the `is_active` and `is_staff` fields of the user requesting the page are set to `True`. In this view, you get the `Order` object with the given ID and render a template to display the order.

Next, edit the `urls.py` file of the `orders` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('create/', views.order_create, name='order_create'),
    path(
 'admin/order/<int:order_id>/',
        views.admin_order_detail,
        name='admin_order_detail'
```

```
    ),
]
```

Create the following file structure inside the `templates/` directory of the `orders` application:

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

Edit the `detail.html` template and add the following content to it:

```
{% extends "admin/base_site.html" %}
{% 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 {
&rsaquo; Detail
  </div>
{% endblock %}
{% block content %}
<div class="module">
<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 %}<
</tr>
<tr>
<th>Stripe payment</th>
<td>
        {% if order.stripe_id %}
          <a href="{{ order.get_stripe_url }}" target="_blank">
            {{ order.stripe_id }}
          </a>
        {% endif %}
      </td>
</tr>
</table>
</div>
<div class="module">
<h2>Items bought</h2>
<table style="width:100%">
<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>
{% endblock %}
```

Make sure that no template tag is split across multiple lines.

This is the template to display the details of an order on the administration site. This template extends the `admin/base_site.html` template of Django's administration site, which contains the main HTML structure and CSS styles. You use the blocks defined in the parent template to include your own content. You display information about the order and the items bought.

When you want to extend an administration template, you need to know its structure and identify existing blocks. You can find all administration templates at https://github.com/django/django/tree/5.0/django/contrib/admin/templates/admin.

You can also override an administration template if you need to. To do so, copy a template into your `templates/` directory, keeping the same relative path and filename. Django's administration site will use your custom template instead of the default one.

Finally, let's add a link to each `Order` object on the list display page of the administration site. Edit the `admin.py` file of the `orders` application and add the following code to it, above the `OrderAdmin` class:

```
from django.urls import reverse
def order_detail(obj):
    url = reverse('orders:admin_order_detail', args=[obj.id])
    return mark_safe(f'<a href="{url}">View</a>')
```

This is a function that takes an `Order` object as an argument and returns an HTML link for the `admin_order_detail` URL. Django escapes HTML output by default. You have to use the `mark_safe` function to avoid auto-escaping.

Then, edit the `OrderAdmin` class to display the link, as follows. The new code is highlighted in bold:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = [
        'id',
        'first_name',
        'last_name',
        'email',
        'address',
        'postal_code',
        'city',
        'paid',
        order_payment,
        'created',
        'updated',
```

```
        order_detail,
    ]
    # ...
```

Start the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. Each row includes a **View** link, as follows:


*Figure 9.29: The View link included in each order row*

Click on the **View** link for any order to load the custom order detail page. You should see a page like the following one:

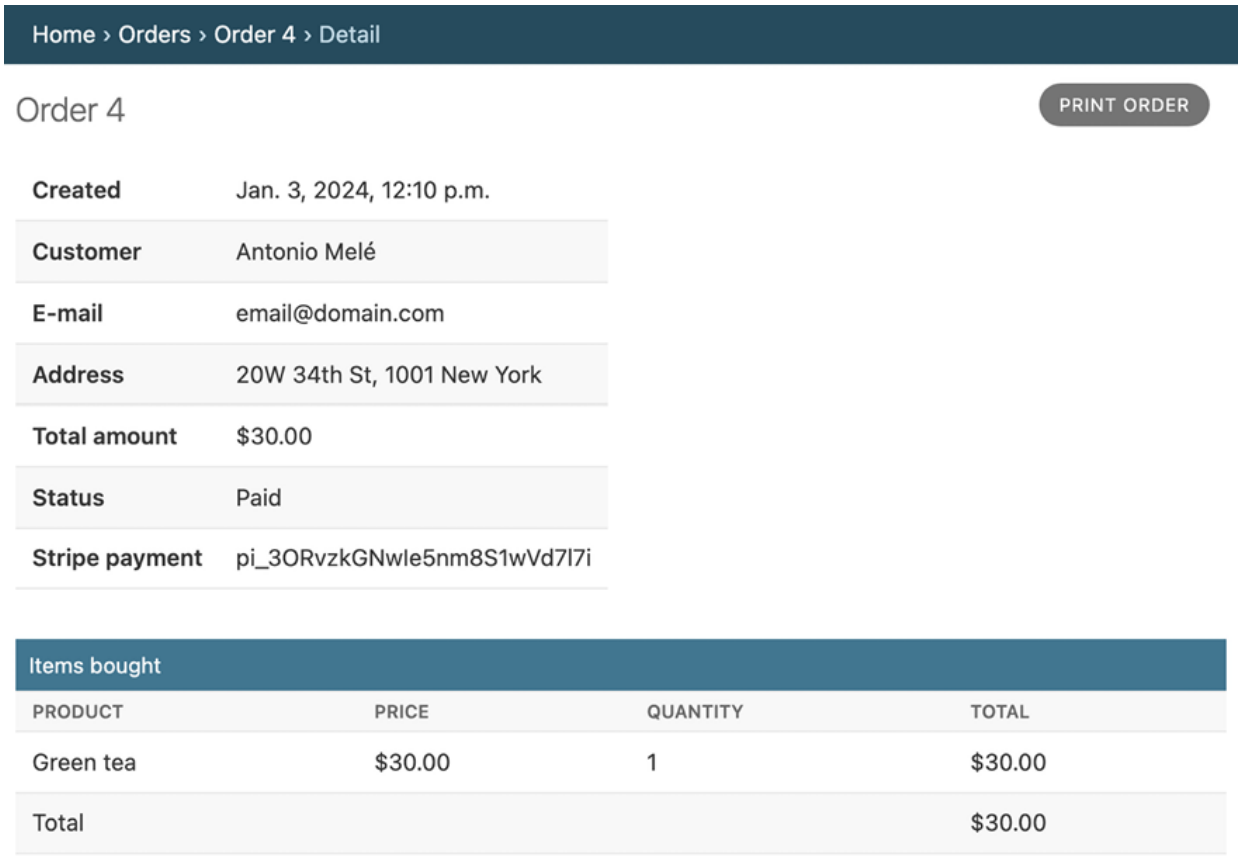



*Figure 9.30: The custom order detail page on the administration site*

Now that you have created the product detail page, you will learn how to generate order invoices in the PDF format dynamically.

# Generating PDF invoices dynamically

Now that you have a complete checkout and payment system, you can generate a PDF invoice for each order. There are several Python libraries to generate PDF files. One popular library to generate PDFs with Python code is ReportLab. You can find information about how to output PDF files with ReportLab at

https://docs.djangoproject.com/en/5.0/howto/outputting-pdf/.

In most cases, you will have to add custom styles and formatting to your PDF files. You will find it more convenient to render an HTML template and convert it into a PDF file, keeping Python away from the presentation layer. You are going to follow this approach and use a module to generate PDF files with Django. You will use WeasyPrint, which is a Python library that can generate PDF files from HTML templates.

# Installing WeasyPrint

First, install WeasyPrint's dependencies for your operating system from https://doc.courtbouillon.org/weasyprint/stable/first_steps.html. Then, install WeasyPrint via `pip` using the following command:

```
python -m pip install WeasyPrint==61.2
```

# Creating a PDF template

You need an HTML document as input for WeasyPrint. You are going to create an HTML template, render it using Django, and pass it to WeasyPrint to generate the PDF file.

Create a new template file inside the `templates/orders/order/` directory of the `orders` application and name it `pdf.html`. Add the following code to it:

```
<html>
<body>
<h1>My Shop</h1>
```

```
<p>
    Invoice no. {{ order.id }}<br>
<span class="secondary">
      {{ order.created|date:"M d, Y" }}
    </span>
</p>
<h3>Bill to</h3>
<p>
    {{ order.first_name }} {{ order.last_name }}<br>
    {{ order.email }}<br>
    {{ order.address }}<br>
    {{ order.postal_code }}, {{ order.city }}
  </p>
<h3>Items bought</h3>
<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
<th>Cost</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>
<span class="{% if order.paid %}paid{% else %}pending{% endif %}
    {% if order.paid %}Paid{% else %}Pending payment{% endif %}
  </span>
```

```
</body>
</html>
```

This is the template for the PDF invoice. In this template, you display all order details and an HTML `<table>` element, including the products. You also include a message to display whether the order has been paid.

# Rendering PDF files

You are going to create a view to generate PDF invoices for existing orders using the administration site. Edit the `views.py` file inside the `orders` application directory and add the following code to it:

```
import weasyprint
from django.contrib.staticfiles import finders
from django.http import HttpResponse
from django.template.loader import render_to_string
@staff_member_required
def admin_order_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    html = render_to_string('orders/order/pdf.html', {'order': o
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'filename=order_{order.id
    weasyprint.HTML(string=html).write_pdf(
        response,
        stylesheets=[weasyprint.CSS(finders.find('css/pdf.css'))
    )
    return response
```

This is the view to generate a PDF invoice for an order. You use the `staff_member_required` decorator to make sure only staff users can access this view.

You get the `Order` object with the given ID and use the `render_to_string()` function provided by Django to render `orders/order/pdf.html`. The rendered HTML is saved in the `html` variable.

Then, you generate a new `HttpResponse` object, specifying the `application/pdf` content type and including the `Content-Disposition` header to specify the filename. You use WeasyPrint to generate a PDF file from the rendered HTML code and write the file to the `HttpResponse` object.

You use the static file `css/pdf.css` to add CSS styles to the generated PDF file. To locate the file, you use the `finders()` function of the `staticfiles` module. Finally, you return the generated response.

If you are missing the CSS styles, remember to copy the static files located in the `static/` directory of the `shop` application to the same location of your project.

You can find the contents of the directory at https://github.com/PacktPublishing/Django-5-by-Example/tree/main/Chapter09/myshop/shop/static.

Since you need to use the `STATIC_ROOT` setting, you have to add it to your project. This is the project's path where static files reside. Edit the `settings.py` file of the `myshop` project and add the following setting:

```
STATIC_ROOT = BASE_DIR / 'static'
```

Then, run the following command:

```
python manage.py collectstatic
```

You should see output that ends like this:

```
131 static files copied to 'code/myshop/static'.
```

The `collectstatic` command copies all static files from your applications into the directory defined in the `STATIC_ROOT` setting. This allows each application to provide its own static files using a `static/` directory containing them. You can also provide additional static file sources in the `STATICFILES_DIRS` setting. All of the directories specified in the `STATICFILES_DIRS` list will also be copied to the `STATIC_ROOT` directory when `collectstatic` is executed. Whenever you execute `collectstatic` again, you will be asked if you want to override the existing static files.

Edit the `urls.py` file inside the `orders` application directory and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    # ...
    path('admin/order/<int:order_id>/pdf/',
         views.admin_order_pdf,
         name='admin_order_pdf'
 ),
]
```

Now, you can edit the administration list display page for the `Order` model to add a link to the PDF file for each result. Edit the `admin.py` file inside the `orders` application and add the following code above the `OrderAdmin` class:

```
def order_pdf(obj):
    url = reverse('orders:admin_order_pdf', args=[obj.id])
    return mark_safe(f'<a href="{url}">PDF</a>')
order_pdf.short_description = 'Invoice'
```

If you specify a `short_description` attribute for your callable, Django will use it for the name of the column.

Add `order_pdf` to the `list_display` attribute of the `OrderAdmin` class, as follows:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = [
        'id',
        'first_name',
        'last_name',
        'email',
        'address',
        'postal_code',
        'city',
        'paid',
        order_payment,
        'created',
        'updated',
        order_detail,
        order_pdf,
    ]
```

Make sure the development server is running. Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. Each row should now include a **PDF** link, like this:

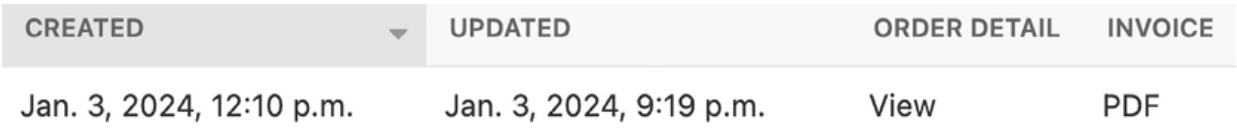



*Figure 9.31: The PDF link included in each order row*

Click on the **PDF** link for any order. You should see a generated PDF file like the following one for orders that have not been paid yet:

# My Shop

Invoice no. 5
Jan 3, 2024

## Bill to

Antonio Melé
email@domain.com
20W 34th St
1001, New York

## Items bought

| Product | Price | Quantity | Cost |
|---|---|---|---|
| Green tea | $30.00 | 1 | $30.00 |
| Red tea | $45.50 | 2 | $91.00 |
| **Total** | | | **$121.00** |

PENDING PAYMENT

*Figure 9.32: The PDF invoice for an unpaid order*

For paid orders, you will see the following PDF file:

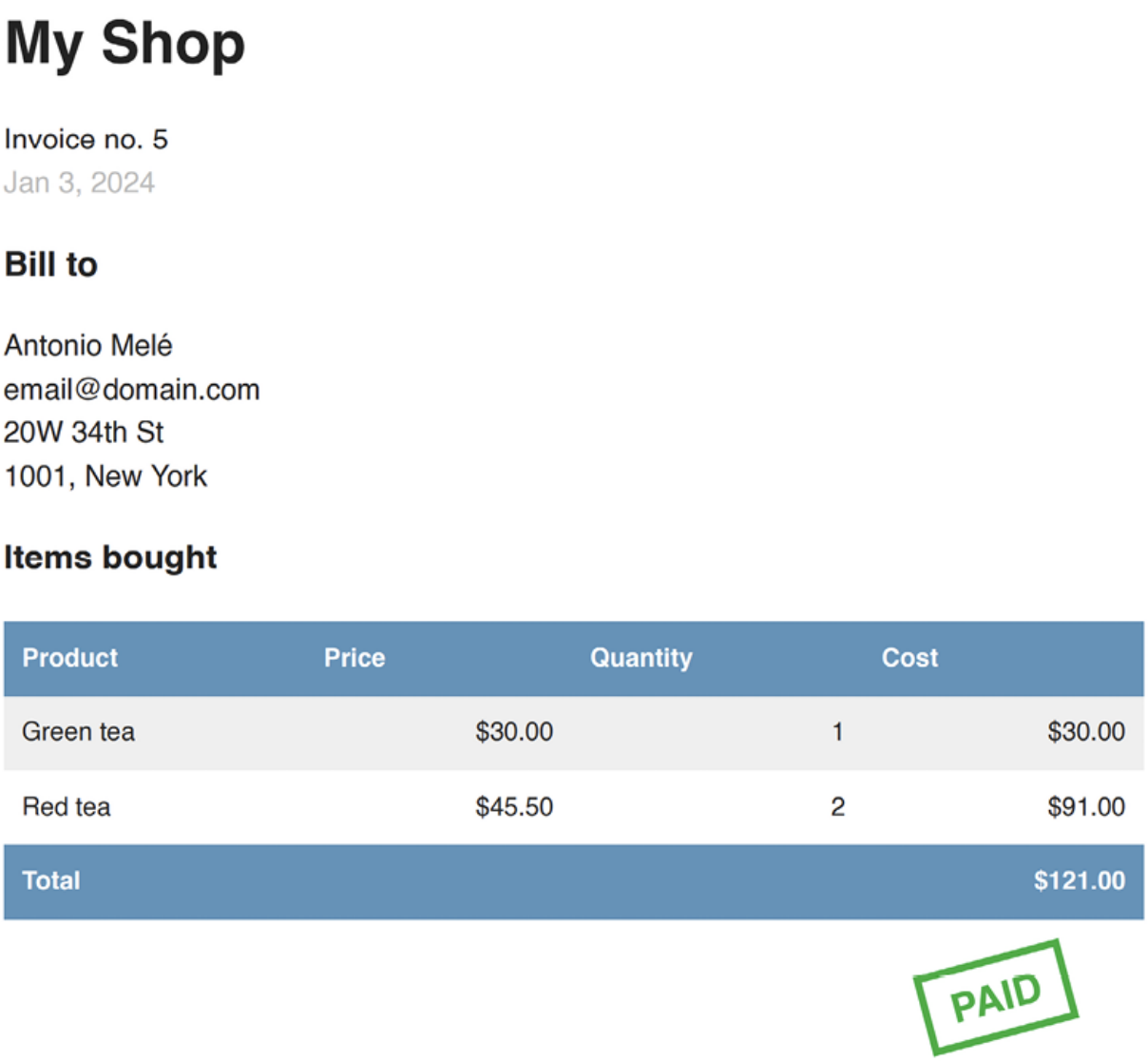



*Figure 9.33: The PDF invoice for a paid order*

# Sending PDF files by email

When a payment is successful, you will send an automatic email to your customer including the generated PDF invoice. You will create an asynchronous task to perform this action.

Create a new file inside the `payment` application directory and name it `tasks.py`. Add the following code to it:

```
from io import BytesIO
import weasyprint
from celery import shared_task
from django.contrib.staticfiles import finders
from django.core.mail import EmailMessage
from django.template.loader import render_to_string
from orders.models import Order
@shared_task
def payment_completed(order_id):
    """
    Task to send an e-mail notification when an order is
    successfully paid.
    """
    order = Order.objects.get(id=order_id)
    # create invoice e-mail
    subject = f'My Shop - Invoice no. {order.id}'
    message = (
        'Please, find attached the invoice for your recent purch
    )
    email = EmailMessage(
        subject, message, 'admin@myshop.com', [order.email]
    )
    # generate PDF
    html = render_to_string('orders/order/pdf.html', {'order': o
    out = BytesIO()
    stylesheets=[weasyprint.CSS(finders.find('css/pdf.css'))]
    weasyprint.HTML(string=html).write_pdf(out, stylesheets=styl
    # attach PDF file
    email.attach(
        f'order_{order.id}.pdf', out.getvalue(), 'application/pd
 )
    # send e-mail
    email.send()
```

You define the `payment_completed` task by using the `@shared_task` decorator. In this task, you use the `EmailMessage` class provided by Django to create an `email` object. Then, you render the template in the `html` variable. You generate the PDF file from the rendered template and output it

to a `BytesIO` instance, which is an in-memory bytes buffer. Then, you attach the generated PDF file to the `EmailMessage` object using the `attach()` method, including the contents of the `out` buffer. Finally, you send the email.

Remember to set up your **Simple Mail Transfer Protocol** (**SMTP**) settings in the `settings.py` file of the project to send emails. You can refer to *Chapter 2, Enhancing Your Blog with Advanced Features*, to see a working example of an SMTP configuration. If you don't want to set up email settings, you can tell Django to write emails to the console by adding the following setting to the `settings.py` file:

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

Let's add the `payment_completed` task to the webhook endpoint that handles payment completion events.

Edit the `webhooks.py` file of the `payment` application and modify it to make it look like this:

```
import stripe
from django.conf import settings
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from orders.models import Order
from .tasks import payment_completed
@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None
try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, settings.STRIPE_WEBHOOK_SECRET
        )
```

```
    except ValueError as e:
        # Invalid payload
return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
return HttpResponse(status=400)
    if event.type == 'checkout.session.completed':
        session = event.data.object
if (
            session.mode == 'payment'
and session.payment_status == 'paid'
        ):
            try:
                order = Order.objects.get(
                    id=session.client_reference_id
                )
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # mark order as paid
            order.paid = True
# store Stripe payment ID
            order.stripe_id = session.payment_intent
            order.save()
            # launch asynchronous task
            payment_completed.delay(order.id)
return HttpResponse(status=200)
```

The `payment_completed` task is queued by calling its `delay()` method. The task will be added to the queue and executed asynchronously by a Celery worker as soon as possible.

Now, you can complete a new checkout process in order to receive the PDF invoice in your email. If you are using the `console.EmailBackend` for your email backend, in the shell where you are running Celery, you will be able to see the following output:

```
MIME-Version: 1.0
Subject: My Shop - Invoice no. 7
```

```
From: admin@myshop.com
To: email@domain.com
Date: Wed, 3 Jan 2024 20:15:24 -0000
Message-ID: <164841212458.94972.10344068999595916799@amele-mbp.ho
--===============8908668108717577350==
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Please, find attached the invoice for your recent purchase.
--===============8908668108717577350==
Content-Type: application/pdf
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="order_7.pdf"
JVBERi0xLjcKJfCflqQKMSAwIG9iago8PAovVHlwZSA...
```

This output shows that the email contains an attachment. You have learned how to attach files to emails and send them programmatically.

Congratulations! You have completed the Stripe integration and have added valuable functionality to your shop.

# Summary

In this chapter, you integrated the Stripe payment gateway into your project and created a webhook endpoint to receive payment notifications. You built a custom administration action to export orders to CSV. You also customized the Django administration site using custom views and templates. Finally, you learned how to generate PDF files with WeasyPrint and how to attach them to emails.

The next chapter will teach you how to create a coupon system using Django sessions, and you will build a product recommendation engine with Redis.

# Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter: https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter09
- Stripe website: https://www.stripe.com/
- Stripe Checkout documentation: https://stripe.com/docs/payments/checkout
- Creating a Stripe account: https://dashboard.stripe.com/register
- Stripe account settings: https://dashboard.stripe.com/settings/account
- Stripe Python library: https://github.com/stripe/stripe-python
- Stripe test API keys: https://dashboard.stripe.com/test/apikeys
- Stripe API keys documentation: https://stripe.com/docs/keys
- Stripe API version 2024-04-10 release: https://stripe.com/docs/upgrades#2024-04-10
- Stripe checkout session modes: https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-mode
- Building absolute URIs with Django: https://docs.djangoproject.com/en/5.0/ref/reques

t-response/#django.http.HttpRequest.build_absolute_uri

- Creating Stripe sessions: https://stripe.com/docs/api/checkout/sessions/create
- Stripe-supported currencies: https://stripe.com/docs/currencies
- Stripe Payments dashboard: https://dashboard.stripe.com/test/payments
- Credit cards for testing payments with Stripe: https://stripe.com/docs/testing
- Stripe webhooks: https://dashboard.stripe.com/test/webhooks
- Types of events sent by Stripe: https://stripe.com/docs/api/events/types
- Installing the Stripe CLI: https://stripe.com/docs/stripe-cli#install
- Latest Stripe CLI release: https://github.com/stripe/stripe-cli/releases/latest
- Generating CSV files with Django: https://docs.djangoproject.com/en/5.0/howto/outputting-csv/
- `django-import-export` application: https://django-import-export.readthedocs.io/en/latest/