Accept a payment

Securely accept payments online.

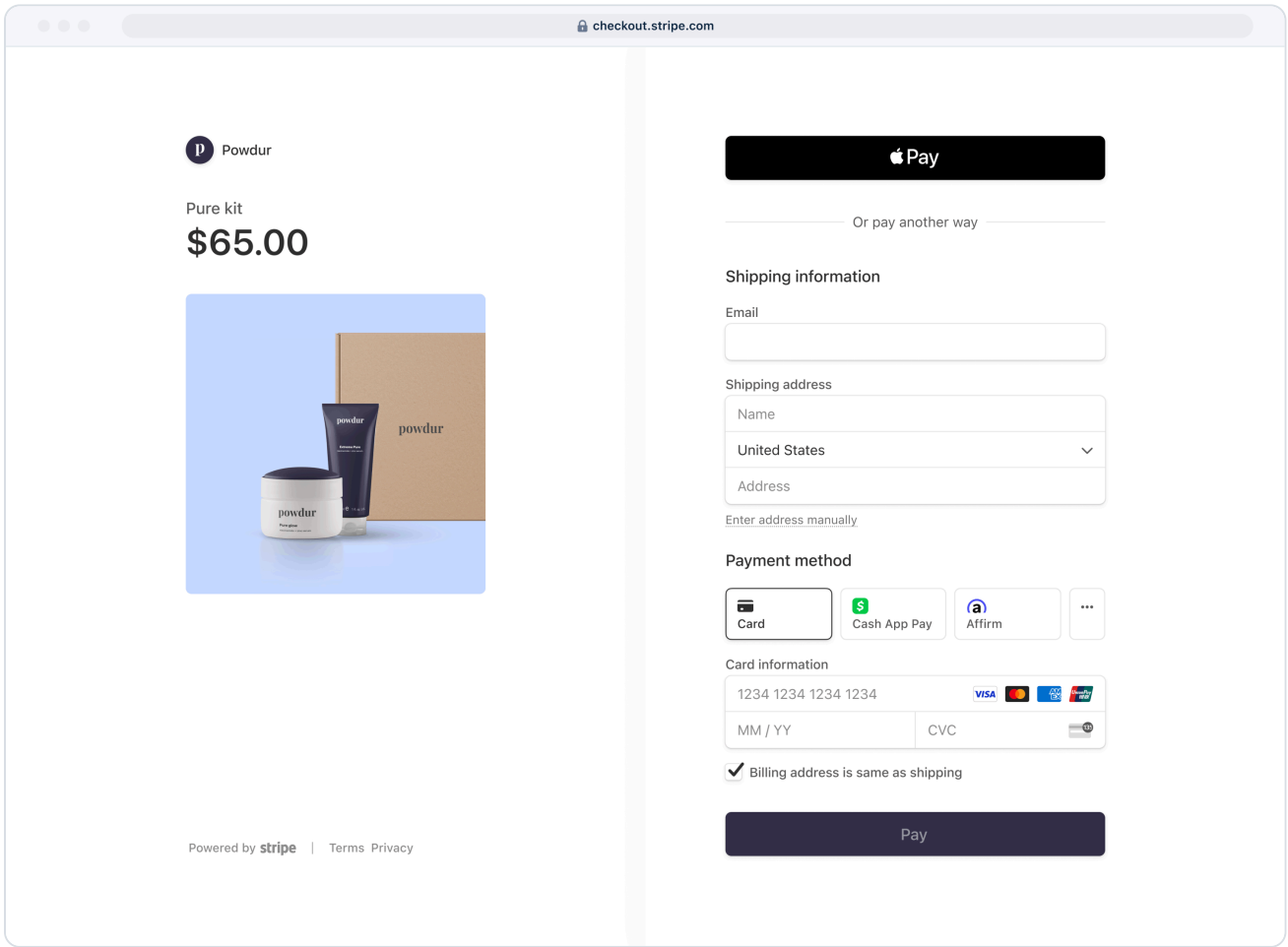
Copy page

Build a payment form or use a prebuilt checkout page to start accepting online payments.

Web | iOS | Android | React Native | Plugins

Stripe-hosted page | Embedded form | Embedded components | Advanced integration

Redirect to a Stripe-hosted payment page using **Stripe Checkout**. See how this integration **compares to Stripe's other integration types**.



Integration effort

Low code

Integration type

Redirect to Stripe-hosted payment page

UI customization

Limited customization

Try it out

First, **register** for a Stripe account.

Use our official libraries to access the Stripe API from your application:

Command Line — Python

```
# Install through pip
pip3 install --upgrade stripe
```

PyPi — Python

```
# Or find the Stripe package on http://pypi.python.org/pypi/stripe/
```

requirements.txt — Python

```
# Find the version you want to pin:
# https://github.com/stripe/stripe-python/blob/master/CHANGELOG.md
# Specify that version in your requirements.txt file
stripe>=5.0.0
```

1 Redirect your customer to Stripe Checkout

Client-side Server-side

Add a checkout button to your website that calls a server-side endpoint to create a **Checkout Session** .

You can also create a Checkout Session for an **existing customer**, allowing you to prefill Checkout fields with known contact information and unify your purchase history for that customer.

checkout.html

```
<html>
  <head>
    <title>Buy cool new product</title>
  </head>
  <body>
    <!-- Use action="/create-checkout-session.php" if your server is PHP based. -->
    <form action="/create-checkout-session" method="POST">
      <button type="submit">Checkout</button>
    </form>
  </body>
</html>
```

A Checkout Session is the programmatic representation of what your customer sees when they're redirected to the payment form. You can configure it with options such as:

- **Line items** to charge
- Currencies to use

You must populate `success_url` with the URL value of a page on your website that Checkout returns your customer to after they complete the payment. You can optionally also provide a `cancel_url` value of a page on your website that Checkout returns your customer to if they terminate the payment process before completion.

> **Note**
> Checkout Sessions expire 24 hours after creation by default.

After creating a Checkout Session, redirect your customer to the **URL** returned in the response.

Python

```python
# This example sets up an endpoint using the Flask framework.
# Watch this video to get started: https://youtu.be/7Ul1vfmsDck.

import os
import stripe

from flask import Flask, redirect

app = Flask(__name__)

stripe.api_key = 'sk_test_tR3PYbcVNZZ796tH88S4VQ2u'

@app.route('/create-checkout-session', methods=['POST'])
def create_checkout_session():
  session = stripe.checkout.Session.create(
    line_items=[{
      'price_data': {
        'currency': 'usd',
        'product_data': {
          'name': 'T-shirt',
        },
        'unit_amount': 2000,
      },
      'quantity': 1,
    }],
    mode='payment',
    success_url='http://localhost:4242/success',
    cancel_url='http://localhost:4242/cancel',
  )

  return redirect(session.url, code=303)

if __name__== '__main__':
    app.run(port=4242)
```

Payment methods

By default, Stripe enables cards and other common payment methods. You can turn individual payment methods on or off in the **Stripe Dashboard**. In Checkout, Stripe evaluates the currency and any restrictions, then dynamically presents the supported payment methods to the customer.

To see how your payment methods appear to customers, enter a transaction ID or set an order amount and currency in the Dashboard.

You can enable Apple Pay and Google Pay in your **payment methods settings**. By default, Apple Pay is enabled and Google Pay is disabled. However, in some cases Stripe filters them out even when they're enabled. We filter Google Pay if you **enable automatic tax** without collecting a shipping address.

Checkout's Stripe-hosted pages don't need integration changes to enable Apple Pay or Google Pay. Stripe handles these payments the same way as other card payments.

Confirm your endpoint

Confirm your endpoint is accessible by starting your web server (for example, `localhost:4242`) and running the following command:

Command Line

```
$ curl -X POST -is "http://localhost:4242/create-checkout-session" -d ""
```

You should see a response in your terminal that looks like this:

Command Line

```
$ HTTP/1.1 303 See Other
> Location: https://checkout.stripe.com/c/pay/cs_test_...
> ...
```

Testing

You should now have a working checkout button that redirects your customer to Stripe Checkout.

1. Click the checkout button.
2. You're redirected to the Stripe Checkout payment form.

If your integration isn't working:

1. Open the Network tab in your browser's developer tools.
2. Click the checkout button and confirm it sent an XHR request to your server-side endpoint (`POST /create-checkout-session`).
3. Verify the request is returning a 200 status.
4. Use `console.log(session)` inside your button click listener to confirm the correct data returned.

2 Show a success page Client-side Server-side

It's important for your customer to see a success page after they successfully submit the payment form. Host this success page on your site.

Create a minimal success page:

success.html

```
<html>
  <head><title>Thanks for your order!</title></head>
  <body>
    <h1>Thanks for your order!</h1>
    <p>
      We appreciate your business!
      If you have any questions, please email
      <a href="mailto:orders@example.com">orders@example.com</a>.
    </p>
  </body>
</html>
```

Next, update the Checkout Session creation endpoint to use this new page:

server.py

Python Resources

```
# Set your secret key. Remember to switch to your live secret key in production.
# See your keys here: https://dashboard.stripe.com/apikeys
import stripe
stripe.api_key = "sk_test_tR3PYbcVNZZ796tH88S4VQ2u"

session =
stripe.checkout.Session.create(
  line_items=[
    {
      "price_data": {
        "currency": "usd",
        "product_data": {"name": "T-shirt"},
        "unit_amount": 2000,
      },
      "quantity": 1,
    },
  ],
  mode="payment",
  success_url="http://localhost:4242/success.html",
  cancel_url="http://localhost:4242/cancel.html",
)
```

> **Note**
> If you want to customize your success page, read the **custom success page** guide.

Testing

1. Click your checkout button.
2. Fill out the payment details with the test card information:

   Enter `4242 4242 4242 4242` as the card number.

   Enter any future date for card expiry.

   Enter any 3-digit number for CVC.

   Enter any billing postal code.
3. Click **Pay**.
4. You're redirected to your new success page.

Next, find the new payment in the Stripe Dashboard. Successful payments appear in the Dashboard's **list of payments**. When you click a payment, it takes you to the payment details page. The **Checkout summary** section contains billing information and the list of items purchased, which you can use to manually fulfill the order.

3 Handle post-payment events

Stripe sends a **checkout.session.completed** event when a customer completes a Checkout Session payment. Use the **Dashboard webhook tool** or follow the **webhook guide** to receive and handle these events, which might trigger you to:

Send an order confirmation email to your customer.

Log the sale in a database.

Start a shipping workflow.

Listen for these events rather than waiting for your customer to be redirected back to your website. Triggering fulfillment only from your Checkout landing page is unreliable. Setting up your integration to listen for asynchronous events allows you to accept **different types of payment methods** with a single integration.

Learn more in our **fulfillment guide for Checkout**.

Handle the following events when collecting payments with the Checkout:

EVENT	DESCRIPTION	ACTION
checkout.session.completed	Sent when a customer successfully	Send the customer an order

EVENT	DESCRIPTION	ACTION
	completes a Checkout Session.	confirmation and **fulfill** their order.
checkout.session.async_payment_succeeded	Sent when a payment made with a delayed payment method, such as ACH direct debt, succeeds.	Send the customer an order confirmation and **fulfill** their order.
checkout.session.async_payment_failed	Sent when a payment made with a delayed payment method, such as ACH direct debt, fails.	Notify the customer of the failure and bring them back on-session to attempt payment again.

4 Test your integration

To test your Stripe-hosted payment form integration:

1. Create a Checkout Session.
2. Fill out the payment details with a method from the following table.

   Enter any future date for card expiry.

   Enter any 3-digit number for CVC.

   Enter any billing postal code.
3. Click **Pay**. You're redirected to your `success_url`.
4. Go to the Dashboard and look for the payment on the **Transactions page**. If your payment succeeded, you'll see it in that list.
5. Click your payment to see more details, like a Checkout summary with billing information and the list of purchased items. You can use this information to fulfill the order.

Learn more about **testing your integration**.

Cards | Wallets | Bank redirects | Bank debits | Vouchers

CARD NUMBER	SCENARIO	HOW TO TEST
4242 4242 4242 4242	The card payment succeeds and doesn't require authentication.	Fill out the credit card form using the credit card number with any expiration, CVC, and postal code.
4000 0025 0000 3155	The card payment requires **authentication**.	Fill out the credit card form using the credit card number with any expiration, CVC, and postal code.
4000 0000 0000 9995	The card is declined with a decline code like `insufficient_funds`.	Fill out the credit card form using the credit card number with any expiration, CVC, and postal code.
6205 5000 0000 0000 004	The UnionPay card has a variable length of 13-19 digits.	Fill out the credit card form using the credit card number with any expiration, CVC, and postal code.

See **Testing** for additional information to test your integration.

Test cards

NUMBER	DESCRIPTION
4242 4242 4242 4242	Succeeds and immediately processes the payment.
4000 0000 0000 3220	Requires 3D Secure 2 authentication for a successful payment.
4000 0000 0000 9995	Always fails with a decline code of `insufficient_funds`.

Build and test new features

Build and test new features using the Stripe developer tools.

Copy page

Use the Stripe developer tools to integrate new features without interrupting your business operations or compromising customer data. This guide helps you:

- Test changes without affecting your live system using **Sandboxes**.
- Build and manage your integration with **Workbench**.
- Listen to key activities in your Stripe account to automate business processes with **event destinations**.

Imagine you're a developer at Kavholm, a furniture company ready to introduce a **new payment method**. This payment method aims to position Kavholm as a market leader and improve customer satisfaction.

Test with Sandboxes

As you prepare to launch Kavholm's new payment method, **set up a sandbox** for thorough testing and monitoring. This isolated environment lets you test features with Stripe functionality without affecting your live system. You can create multiple sandboxes without affecting other users on your account, allowing you to manage isolated environments for development and continuous integration tests. Additionally, you can:

- **Simulate payment scenarios**: Experiment with payment processes related to the new payment method without making actual transactions. Any settings you configure in your sandbox remain isolated to the testing environment and don't impact your live account.
- **Choose a sandbox configuration**: Set up a pre-configured sandbox that mirrors some settings of your live account for realistic testing, or select a blank configuration to explore new settings. Learn more about **sandbox settings**.

Develop in a sandbox

To develop in a sandbox, consider using these features among other available options that might benefit your use case:

Team-based sandboxes: Assign dedicated sandboxes for development teams to focus on specific areas of the payment method integration without interference.

Test APIs and webhooks: Verify integration logic by testing API calls and webhook responses related to payment method events, such as `payment_intent.succeeded`.

Collaborate with external partners

To collaborate with external partners, such as vendors, assign them the **sandbox user role** to provide controlled access for testing, ensuring that live data security isn't compromised. This role is ideal for external partners and vendors, such as development agencies.

Debugging and validation

To debug and validate your integration, use the following features:

Troubleshoot and fix bugs: Debug payment method workflow issues by testing API calls or integration logic changes in the sandbox.

API key management: Configure API keys for secure requests in the sandbox, avoiding key-related errors.

Dedicated sandboxes

By organizing dedicated sandboxes for each testing scenario, you can simulate real-world conditions, monitor functionality, and debug integration issues without risking live operations.

For example, you can test all of the payment method functionality before it goes live, monitor integration performance continuously, and implement changes that you've vetted in an equivalent testing environment.

To get started, create a sandbox by navigating to the account picker menu in the Stripe Dashboard and selecting the **Create** button. Each sandbox must have a name and can copy settings from the live account to mirror actual conditions.

With dedicated sandboxes, you can test new features, such as the new payment method, through various stages such as development, integration testing, and user acceptance testing (UAT). This ensures that each stage receives thorough checks without impacting the production environment.

Build and manage your integration with Workbench

You can debug and manage your Stripe integration from your browser using Workbench directly in the **Dashboard**.

Use the following views to manage the payment method feature:

Overview:

- Make sure the new payment method uses the latest and most secure API version.
- Monitor API requests for payment method fee calculations to confirm they function as expected.
- Identify integration errors related to the payment method.

Errors:

- Identify and categorize errors in payment method calculations or payment processing.
- Track error frequency to understand impact and urgency.

Inspector:

- Analyze API object configurations and troubleshoot issues in real-time.

Logs:

- Filter API requests to make sure interactions align with expectations.

Events:

- Filter events to verify logistical triggers.
- Review event payloads to confirm correct back-end processing.

Webhooks:

- Set up webhook endpoints or other destination types, such as Amazon EventBridge, for real-time updates on payment method status and payment confirmations.

Shell and API Explorer:

- Simulate API requests and manage them using a command-line interface.

Listen to real-time updates with event destinations

At Kavholm, use event destinations to make sure all payment method features function as intended without affecting your live systems. Use event destinations to track real-time

activities in your Stripe account, and to respond to critical events such as payment confirmations or subscription updates. You can:

- Send events to AWS through **Amazon EventBridge** or to an **HTTPS endpoint through webhooks**.
- Access real-time data using **thin or snapshot events**.

Testing

To simulate real-world conditions and evaluate the payment method's performance before it goes live:

- Test the payment method feature in a sandbox by simulating Stripe-generated events.
- Set up test webhooks to observe event processing without impacting live customers.

React to real-time updates

To track key events and maintain oversight of the payment method feature's ongoing functionality:

- Configure event destinations to **aggregate and alert you on events** such as `payment_intent.succeeded`. Event destinations support receiving alerts at a webhook endpoint or Amazon EventBridge.
- View payment event history to troubleshoot payment method processes and **track event deliveries**. Use logs for debugging, especially when your focus is on event logs rather than API request logs.

Debugging

To identify and resolve issues through real-time event analysis and ensure uninterrupted service for Kavholm's customers:

- Quickly identify billing or payment method issues to alert your customers with specific event notifications.
- Use **thin events for real-time analysis** of payment method processing.

With event destinations, Kavholm tests, monitors, and debugs its new payment method while minimizing disruptions.

See also

- Sandboxes

Go-live checklist

Use this checklist to ensure a smooth transition when taking your integration live.

Copy page

> Note
> **Become a Stripe Partner** to access additional best practices and receive relevant news and updates from Stripe.

CHECKLIST PROGRESS

As you complete each item and check it off, the state of each checkbox is stored within your browser's cache. You can refer back to this page at any time to see what you've completed so far.

You can **log in** to see some of your current settings.

Stripe has designed its live and **sandbox** environments to function as **similarly as possible**. Switching between them is mostly a matter of swapping your **API keys**.

If you are a developer, or had a developer perform an integration for you, you should also consider the following items before going live. If you're using Stripe through a connected website or a plug-in, most won't apply.

Set the API version

> Warning
> All requests use your account API settings, unless you override the API version. The **changelog** lists every available version. Note that by default webhook events are structured according to your account API version, unless you set an API version during **endpoint creation** .

If you're using a strongly typed language (Go, Java, TypeScript, .NET), the server-side library pins the API version based on the library version being used. If you're not familiar with how Stripe manages versioning, please see the **versioning docs**.

To make sure everything is in sync:

- Upgrade to the latest API version in **Workbench** within the Dashboard

For dynamic languages (Node.js, PHP, Python, Ruby): **set the API version** in the server-side library

For strongly typed languages (Go, Java, TypeScript, .NET): **upgrade to the latest version** of your chosen library

Handle edge cases

We've created several **test values** you can use to replicate various states and responses. Beyond these options, perform your due diligence, testing your integration with:

Incomplete data

Invalid data

Duplicate data (for example, retry the same request to see what happens) We also recommend you have someone else test your integration, especially if that other person isn't a developer themselves.

Review your API error handling

Once you've gone live is an unfortunate time to discover you've not properly written your code to handle every possible **error type** , including those that should "never" happen. Be certain your code is defensive, handling not just the common errors, but all possibilities.

When testing your error handling, pay close attention to what information is shown to your users. A card being declined (that is, a `card_error`) is a different concern than an error on your backend (for example, an `invalid_request_error`).

Review your logging

Stripe logs every request made with your API keys, with these records being viewable in the **Dashboard**. We recommend that you log all important data on your end, too, despite the apparent redundancy. Your own logs will be a life-saver if your server has a problem contacting Stripe or there's an issue with your API keys—both cases would prevent us from logging your request.

Send your first Stripe API request

Get started with the Stripe API.

Copy page

Every call to a Stripe API must include an API secret key. After you create a Stripe account, we generate two pairs of **API keys** for you—a publishable client-side key and a secret server-side key—for both **test** and **live** modes. To start moving real money with your live-mode keys, you need to **activate your account**.

1 Before you begin

This guide walks you through a simple interaction with the Stripe API—creating a customer. For a better understanding of Stripe API objects and how they fit together, take a **tour of the API** or visit the **API reference** . If you're ready to start accepting payments, see our **quickstart**.

2 Send your first API request

You can begin exploring Stripe APIs using the **Stripe Shell**. The Stripe Shell allows you to execute Stripe CLI commands directly within the Stripe docs site. As it operates in a **sandbox** environment only, you don't have to worry about initiating any real money-moving transactions.

1 To **create a customer** using the Stripe Shell, enter the following command:

Command Line

```
$ stripe customers create --email=jane.smith@email.com --name="Jane Smith" --des
```

If everything worked, the command line displays the following response:

```
{
  "id": "cus_LfctGLAICpokzr",
  "object": "customer",
```

… See all 31 lines

2 (Optional) Run the same command by passing in your API secret key in a sandbox:

Command Line

```
$ stripe customers create --email=jane.smith@email.com --name="Jane Smith" --des
```

If everything worked, the command line displays the following response:

```
{
  "id": "cus_LfdZgLFhah76qf",
  "object": "customer",
```

… See all 32 lines

3 View logs and events

Whenever you make a call to Stripe APIs, Stripe creates and stores API and **Events** objects for your Stripe **user account**. The API key you specify for the request determines whether the objects are stored in a sandbox environment or in live mode. For example, the last request used your API secret key, so Stripe stored the objects in a sandbox.

To view the API request log:

Open the **Logs** page.

Click **200 OK POST /v1 customers**.

To view the Event log:

Open the **Events** page.

Click **jane.smith@email.com is a new customer**.

4 Store your API keys

By default, all accounts have a total of four API keys, two in a **sandbox** and two in live mode.

Sandbox secret key: Use this key to authenticate requests on your server when you're testing in a sandbox. By default, you can use this key to perform any API request without restriction. Reserve this key for testing and development to make sure that you don't accidentally modify your live customers or charges.

Sandbox publishable key: Use this key for testing purposes in your web or mobile app's client-side code. Reserve this key for testing and development to make sure that you don't accidentally modify your live customers or charges.

Live mode secret key: Use this key to authenticate requests on your server when in live mode. By default, you can use this key to perform any API request without restriction.

Live mode publishable key: Use this key, when you're ready to launch your app, in your web or mobile app's client-side code.

Your secret and publishable keys are in the Dashboard in the **API keys** tab. If you can't view your API keys, ask the owner of your Stripe account to add you to their **team** with the proper permissions.

> **Restricted API keys**
> You can generate **restricted API keys** in the Dashboard to enable customizable and limited access to the API. However, Stripe doesn't offer any restricted keys by default.

When you're logged in to Stripe, our documentation automatically populates code examples with your test API keys. Only you can see these values. If you're not logged in, our code examples include randomly generated API keys. You can replace them with your own test keys or **log in** to see the code examples populated with your test API keys.

The following table shows randomly generated examples of secret and publishable test API keys:

TYPE	VALUE	WHEN TO USE
Secret	`sk_test_tR3PYbcVNZZ796tH88S4VQ2u`	**On the server side**: Must be secret and stored securely in your web or mobile app's server-side code (such as in an environment variable or credential management system) to call Stripe APIs. Don't expose this key on a website or embed it in a mobile application.

TYPE	VALUE	WHEN TO USE
Publishable	`pk_test_51BTUDGJAJfZb9HEBwDg86TN1KNprHjkfipXmEDMb0gSCassK5T3ZfxsAbcgKVmAIXF7oZ6ItlZZbXO6idTHE67IM007EwQ4uN3`	**On the client side**: Can be publicly accessible in your web or mobile app's client-side code (such as checkout.js) to securely collect payment information, such as with **Stripe Elements**. By default, **Stripe Checkout** securely collects payment information.
Restricted	A string that starts with `rk_test_`	**In microservices**: Must be secret and stored securely in your microservice code to call Stripe APIs. Don't expose this key on a website or embed it in a mobile application.

See also

Set up your development environment

Get familiar with the Stripe CLI and our server-side SDKs.

Copy page

NOT A DEVELOPER?

Check out our **no-code docs**, use a **prebuilt solution** from our partner directory, or hire a **Stripe-certified expert**.

Stripe's server-side SDKs and command-line interface (CLI) allow you to interact with Stripe's REST APIs. Start with the Stripe CLI to streamline your development environment and make API calls.

Use the SDKs to avoid writing boilerplate code. To start sending requests from your environment, choose a language to follow a quickstart guide.

> **Chrome extensions**
> We recommend you build your payment integration with Stripe (such as **Elements** or **Checkout**) on your own website. Then, set up your Chrome extension to send users to this payment page when they're ready to complete a purchase.
>
> This method is more secure and easier to maintain than trying to handle payments directly within the extension.

Ruby | Python | Go | Java | Node.js | PHP | .NET

In this quickstart, you install the **Stripe CLI**—an essential tool that gets you command line access to your Stripe integration. You also install the **Stripe Python server-side SDK** to get access to Stripe APIs from applications written in Python.

What you learn

In this quickstart, you'll learn:

- How to call Stripe APIs without writing a line of code

How to manage third-party dependencies using a virtual environment and the pip package manager

How to install the latest Stripe Python SDK v12.0.0

How to send your first SDK request

Initial setup

First, **create a Stripe account** or **sign in**.

1 Set up the Stripe CLI

Install

From the command-line, use an install script or download and extract a versioned archive file for your operating system to install the CLI.

homebrew | apt | yum | Scoop | macOS | Linux | Windows | Docker

To install the Stripe CLI with **homebrew**, run:

Command Line

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

This command fails if you run it on the Linux version of homebrew, but you can use this alternative or follow the instructions on the Linux tab.

Command Line

```
brew install stripe-cli
```

Authenticate

Log in and authenticate your Stripe user **Account** to generate a set of *restricted keys*. To learn more, see **Stripe CLI keys and permissions**.

Command Line

```
$ stripe login
```

Press the **Enter** key on your keyboard to complete the authentication process in your browser.

Output

```
$  Your pairing code is: enjoy-enough-outwit-win
>  This pairing code verifies your authentication with Stripe.
>  Press Enter to open the browser or visit https://dashboard.stripe.com/stripecli/c
```

Confirm setup

Now that you've installed the CLI, you can make a single API request to Create a product .

Command Line

```
$  stripe products create \
>  --name="My First Product" \
>  --description="Created with the Stripe CLI"
```

Look for the product identifier (in `id`) in the response object. Save it for the next step.

If everything worked, the command-line displays the following response.

```
1  {
2    "id": "prod_LTenIrmp8Q67sa",
3    "object": "product",
...  See all 25 lines
```

Next, call Create a price to attach a price of 30 USD. Swap the placeholder in `product` with your product identifier (for example, `prod_LTenIrmp8Q67sa`).

Command Line

```
$  stripe prices create \
>    --unit-amount=3000 \
>    --currency=usd \
>    --product={{PRODUCT_ID}}
```

If everything worked, the command-line displays the following response.

```
{
  "id": "price_1KzlAMJJDeE9fu01WMJJr79o",
  "object": "price",
```

… See all 20 lines

2 Manage third-party dependencies

We recommend managing third-party dependencies using the **venv** module, which allows you to add new libraries and include them in your Python 3 projects.

On Windows (cmd.exe):

Command Line

```
$ python3 -m venv env
> .\env\Scripts\activate.bat
```

On GNU/Linux or MacOS (bash):

Command Line

```
$ python3 -m venv env
> source env/bin/activate
```

3 Install the Python server-side SDK

The latest version of the Stripe Python server-side SDK is v12.0.0. It supports Python versions 3.6+.

Check your Python version:

Command Line

```
$ python3 --version
```

Install the library

Install the library from **PyPi**, a package manager for Python:

Command Line

```
$ pip3 install --upgrade stripe
```

Next, specify the following version in your requirements.txt file:

requirements.txt

```
1 stripe>=12.0.0
```

Installation alternatives

4 Run your first SDK request

Now that you have the Python SDK installed, you can create a subscription **Product** and attach a **Price** with a couple API requests. We're using the product identifier returned in the response to create the price in this example.

> **Note**
> This sample uses the default keys of your Stripe user **account** for your **sandbox** environment. Only you can see these values.

create_price.py

```
1 import stripe
2 stripe.api_key = "sk_test_tR3PYbcVNZZ796tH88S4VQ2u"
3
4 starter_subscription = stripe.Product.create(
5   name="Starter Subscription",
6   description="$12/Month subscription",
```