Most web applications use email to manage crucial operations, such as resetting passwords, account activation, receiving customer feedback, sending newsletters and marketing campaigns. Most of these tasks require a dedicated service like [SendGrid](https://sendgrid.com/) or [Mailgun](https://www.mailgun.com/). But if you don’t expect your site to get a huge quantity of visitors, you can actually get a lot done through your personal email provider.

Sending emails with your personal email is a reasonable option for small or testing projects, so we’ll take that approach here to keep things simple. However, it’s not a good idea to use your personal email service for your production website. You can learn more about [Gmail sending restrictions](https://support.google.com/a/answer/166852), or refer to the limitations of your email provider.

*Note: the full code for this tutorial is available on [GitHub](https://github.com/sitepoint-editors/Email-Django-app).*

**Understanding SMTP**

[SMTP](https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol) (or the Simple Mail Transfer Protocol) is a set of rules for determining how emails are transferred from senders to recipients. SMTP servers use this protocol to send and relay outgoing emails. (Note that other protocols govern how emails are *recieved*.)

An SMTP server always has a unique address, and a specific port for sending messages, which in most cases is **587**. We’ll see how the port is relevant while sending emails with Django.

Since we’ll be using Gmail, the address we’ll be working with is smtp.gmail.com, and the port will be 587.

Now let’s see how we can send emails with Django.

**Creating a Django Project**

Every Django project should have a virtual environment, as we don’t want to mess up the project dependencies. To create one, run the following:

python -m venv .venv

*Note: if you’re unfamiliar with virtual environments, make sure to check our*[*Python virtual environments guide*](https://www.sitepoint.com/virtual-environments-python-made-easy/)*.*

The command above creates a virtual environment with the name .venv. To activate this virtual environment, you can use the following:

source .venv/bin/activate

Since Django is a third-party package, you have to install it with pip:

pip install django

This will install the latest version of Django, which you can check with pip freeze.

To create a Django project, you call the command line utility [django-admin](https://docs.djangoproject.com/en/dev/ref/contrib/admin/):

django-admin startproject EmailProject

With the command above, you’re creating a Django project with the name EmailProject, but you can create the project with whatever name you want.

Now, enter to the project directory and run the server:

cd EmailProject

python manage.py runserver

After running the Django server, visit [http://localhost:8000](http://localhost:8000/) in your browser. You’ll see an auto-generated page with the latest Django release notes.

## Modifying Settings

You’ll need to modify the settings file before sending emails, so let’s locate that file with the command tree:

Note: for simplicity’s sake, we’ll be using only UNIX (macOS or Linux) system commands.

tree

The tree command outputs the file structure of a directory. In this case, since we’re not giving it a specific directory path, we’ll get something similar to the following if we’re in the root folder of the project:

├── EmailProject

│ ├── asgi.py

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

└── manage.py

1 directory, 6 files

The file we’ll be constantly modifying through this tutorial is the [settings.py](https://docs.djangoproject.com/en/dev/topics/settings/) inside the EmailProject folder.

settings.py holds all the project configuration you’ll need, and allows you to set custom variables. As the Django docs say, “A settings file is just a Python module with module-level variables”.

Let’s look at the settings required for sending an email with Django. Open the EmailProject/settings.py file and paste the following settings at the bottom of the file:

# EmailProject/settings.py

# Bottom of the file

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = ''

EMAIL\_PORT = 587

EMAIL\_USE\_TLS = True

EMAIL\_HOST\_USER = ''

EMAIL\_HOST\_PASSWORD = ''

Let’s break down the code above by analyzing each one of these settings.

### **Email Backend**

The [EMAIL\_BACKEND](https://docs.djangoproject.com/en/3.2/ref/settings/#email-backend) setting declares the backend our Django project will use to connect with the SMTP server.

This variable is pointing to the smtp.EmailBackend class that receives all the parameters needed to send an email. I strongly suggest you take a look at the class constructor directly on the [Django source code](https://github.com/django/django/blob/8a7ac78b706797a03d26b88eddb9d1067ed35b66/django/core/mail/backends/smtp.py" \l "L12). You’ll be surprised by how readable this code is.

Note: although this class is the default *EMAIL\_BACKEND*, it’s considered a good practice to be explicit in the Django settings.

All the other email settings will be based on the constructor of this [EmailBackend class](https://docs.djangoproject.com/en/3.2/topics/email/" \l "smtp-backend).

### **Email host**

The EMAIL\_HOST setting refers to the SMTP server domain you’ll be using. This depends on your email provider. Below is a table with the SMTP server host corresponding to three common providers:

| **Email provider** | **SMTP server host** |
| --- | --- |
| Gmail | smtp.gmail.com |
| Outlook/Hotmail | smtp-mail.outlook.com |
| Yahoo | smtp.mail.yahoo.com |

We’re leaving this setting blank for now since we’ll use a .env file later to avoid hard-coded sensitive keys or per-site configurations. You should never set credentials directly into code.

We’ll be using [Django Environ](https://www.sitepoint.com/django-send-email/" \l "usingdjangoenvirontohidesensitivekeys) to solve this problem.

### **Email Port**

The EMAIL\_PORT setting must be set to 587 because it’s the default port for most SMTP servers. This remains true for personal email providers.

This port is used along with TLS encryption to ensure the security of email sending.

### **Email Use TLS**

[Transport Layer Security](https://en.wikipedia.org/wiki/Transport_Layer_Security) (TLS) is a security protocol used across the Web to encrypt the communication between web apps (Django) and servers (SMTP server).

Originally, we set the EMAIL\_USE\_TLS variable to True. This means Django will use Transport Layer Security to connect to the SMTP server and send emails. (It’s mandatory for personal email providers.)

### **Email Host User**

The EMAIL\_HOST\_USER setting is your personal email address. Leave it blank for now, since we’ll use django-environ to set up all of these credentials.

### **Email Host Password**

The EMAIL\_HOST\_PASSWORD setting is the app password you’ll get from your email account — the process we’ll be doing right after this section.

Same story: leave this setting blank, as we’ll use environmental variables later.

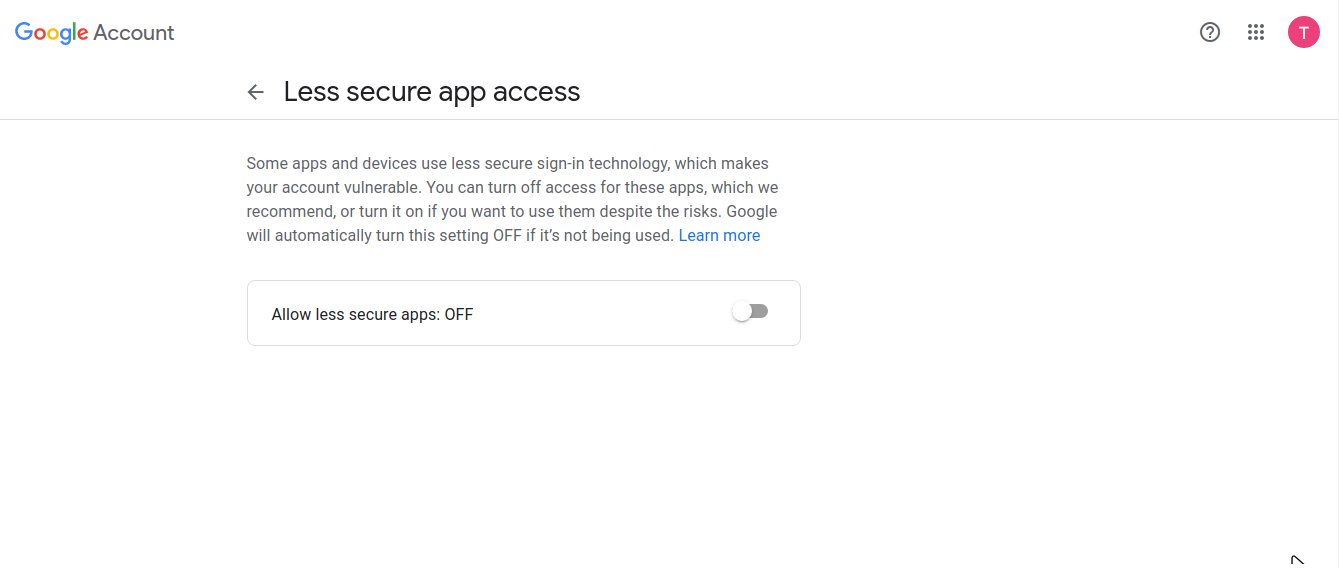
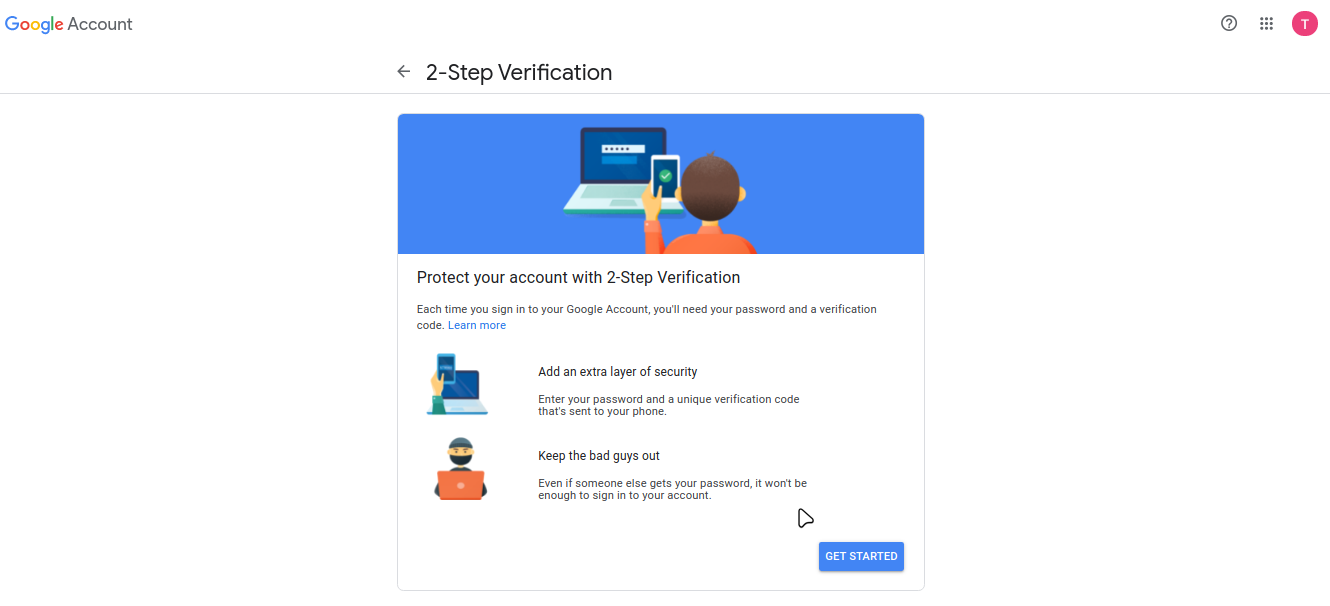
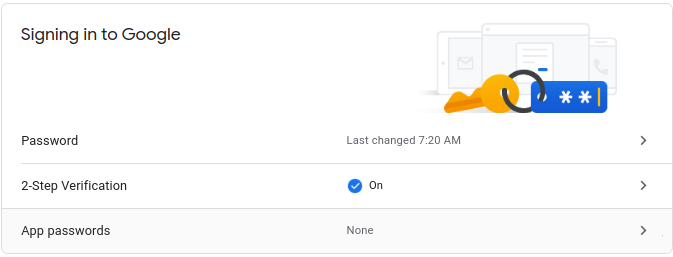
## Set Up an App Password in Gmail

To use the EMAIL\_HOST\_PASSWORD setting you’ll need to activate the **less secure app access** and have an app password from your personal email address.

If you don’t activate the **less secure app access**, you’ll probably get a SMTPAuthenticationError, because Django has no way to comply with Google security protocols.

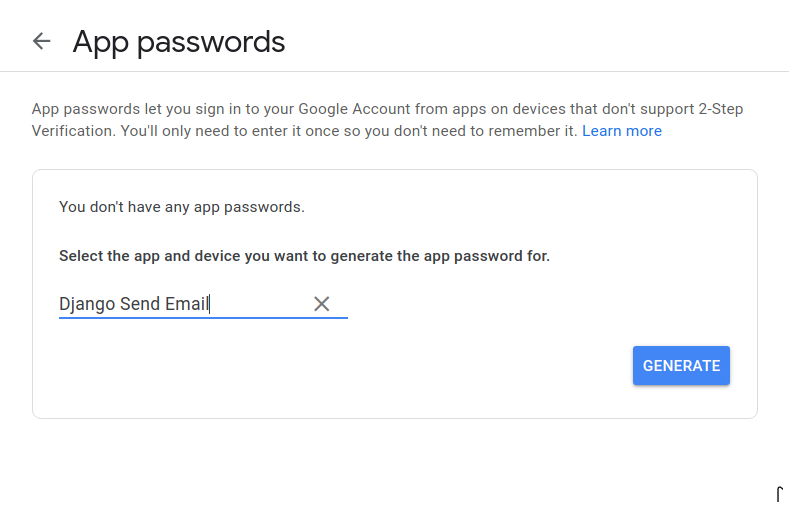
You may opt to use your normal password, but it would be even more risky than using an app password. My advice is to create a new Gmail account or to use a “testing” email address.

Taking this into account, you can get a [Gmail app password](https://support.google.com/accounts/answer/185833?hl=en) with the steps below. Note that if you’re using an existing account and have enabled 2-step verification, you can skip steps 2 and 3:

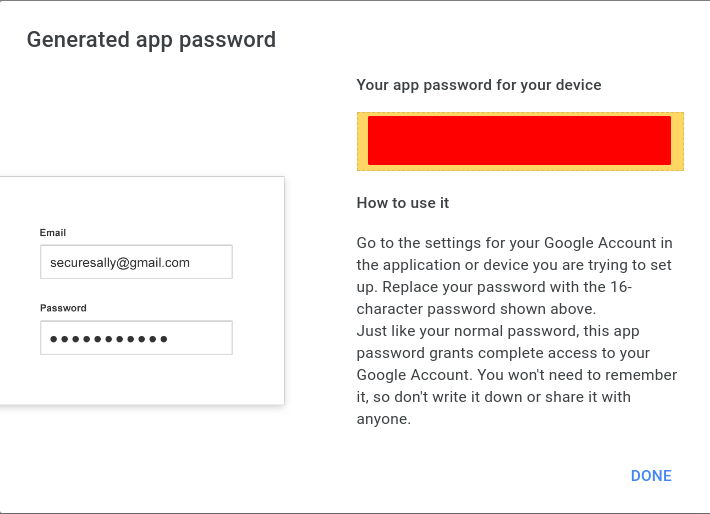
1. Create or Login into a Gmail account
2. Go to [myaccount.google.com/lesssecureapps](https://myaccount.google.com/lesssecureapps) and turn on the less secure apps option.  
   
3. Enable [two-factor authentication](https://myaccount.google.com/signinoptions/two-step-verification/enroll-welcome), as it’s required to get an app password.  
   
4. Now you have two-factor authentication enabled, it’s time to get an app password. You can do this by going to the [security section](https://myaccount.google.com/security) of your google account, scrolling down to the Signing in to Google section, and clicking on **App passwords**.  
   

You’ll need to re-prompt your password (account password), before being redirected to the App passwords page.

Once you’re in, click on **select app**, where you’ll choose a custom name for that app password — such as “Django Send Email” — then click on **GENERATE**.



A new window will show up with a sixteen-character password. Copy it, because we’ll need it to configure our Django project.



If you’re using other email providers, make sure to read the following guides:

* [Yahoo app password](https://help.yahoo.com/kb/generate-separate-password-sln15241.html)
* [Outlook or Hotmail app password](https://docs.microsoft.com/en-us/azure/active-directory/user-help/multi-factor-authentication-end-user-app-passwords)

## Using Django Environ to Hide Sensitive Keys

Even if you’re just sending emails in development, you shouldn’t write passwords directly into source code. This becomes even more important when using a [version control system](https://www.sitepoint.com/git-for-beginners/) along with GitHub to host your project. You don’t want people to access your data.

Let’s see how we can prevent this by using [Django-environ](https://pypi.org/project/django-environ/).

Create a .env file inside the EmailProject directory (where the settings.py file is located) with the command below:

cd EmailProject/

ls

settings.py # The settings file must be here

touch .env

Now, open that .env file and enter the following key–value pairs:

EMAIL\_HOST=smtp.gmail.com

EMAIL\_HOST\_USER=YourEmail@address

EMAIL\_HOST\_PASSWORD=YourAppPassword

RECIPIENT\_ADDRESS=TheRecieverOfTheMails

Breaking down the contents of this file:

* EMAIL\_HOST: your email provider SMTP server address. See the email host table above for quick guidance. In this case, I’m using smtp.gmail.com, the Gmail SMTP address.
* EMAIL\_HOST\_USER: your email address.
* EMAIL\_HOST\_PASSWORD: the app password you just generated. Have in mind it doesn’t include any spaces.
* RECIPIENT\_ADDRESS: the email address in which you’ll receive the messages. This is a custom setting that we’ll create later to send all the emails to the same recipient.

To make use of these environmental variables, we’ll need to install Django-environ:

pip install django-environ

Note: make sure your virtual environment is activated.

Now, open the settings.py located at the EmailProject directory and use the code below:

# EmailProject/settings.py

# This should be at the start of the file

import environ

env = environ.Env()

environ.Env.read\_env()

# Previous settings ...

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = env('EMAIL\_HOST')

EMAIL\_PORT = 587

EMAIL\_USE\_TLS = True

EMAIL\_HOST\_USER = env('EMAIL\_HOST\_USER')

EMAIL\_HOST\_PASSWORD = env('EMAIL\_HOST\_PASSWORD')

# Custom setting. To email

RECIPIENT\_ADDRESS = env('RECIPIENT\_ADDRESS')

First, we’re importing the environ package at the top of the settings file. Remember that all imports should be at the start.

Then we create an env variable which will contain all the key–value pairs available on the .env.

The env('KEY') statement means we’re looking up the value of that key. Make sure you have set up your .env file before proceeding, because you’ll get a Django ImproperlyConfigured error in case some environmental variable wasn’t set.

Note that RECIPIENT\_ADDRESS is a custom setting that we’ll use to send the emails to an address we can access.

**Don’t forget** to include the .env file in your [.gitignore](https://git-scm.com/docs/gitignore) in case you’re using Git and GitHub. You can do this just by opening it and adding the following line:

.env

## 1. Sending Emails with the Django Shell

Finally, we get to the juicy part of the article! It’s time to send your first email to Django.

Open up a terminal, activate the virtual environment, and run:

python manage.py shell

This will create a shell with all the Django settings already configured for us. Inside that brand new shell, paste the following code:

>>> from django.core.mail import send\_mail

>>> from django.conf import settings

>>> send\_mail(

... subject='A cool subject',

... message='A stunning message',

... from\_email=settings.EMAIL\_HOST\_USER,

... recipient\_list=[settings.RECIPIENT\_ADDRESS])

1

We can also make a one-liner without specifying the arguments:

>>> send\_mail('A cool subject', 'A stunning message', settings.EMAIL\_HOST\_USER, [settings.RECIPIENT\_ADDRESS])

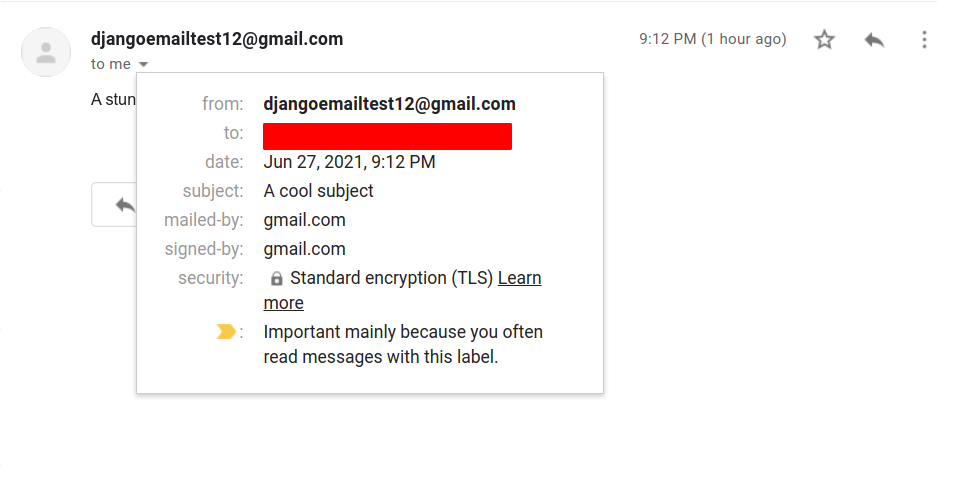
1

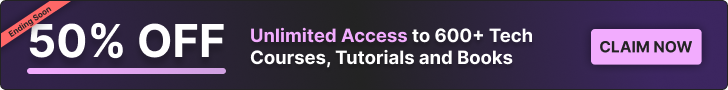
Let’s break down the code above:

* We import the Django [send\_mail](https://docs.djangoproject.com/en/3.2/topics/email/" \l "send-mail) function.
* Then we import the settings object which contains all the [global settings](https://github.com/django/django/blob/main/django/conf/global_settings.py) and the per-site settings (those inside the settings.py file).
* Finally, we pass all the needed arguments to the send\_mail function. This function returns the number of emails sent, in this case, **1**.

Note how we’re using the settings object to get the from\_email (the email you’re sending emails with) and the recipient\_list (the RECIPIENT\_ADDRESS custom setting we defined in the .env).

Now, if I check my inbox — as I set the RECIPIENT\_ADDRESS environmental variable to my email address — I’ll get the message sent by Django.



[](https://featured.sitepoint.com/50-off-sitepoint-premium)

## 2. Build an Automated Contact Form with Django

In this section, we’ll build an automated contact form with Django forms and the built-in send\_mail function. Also, we’ll be creating a custom function, send(), inside the contact form so it’s easier to implement it in the views.

Let’s start by creating the contact app. Enter the project root directory — where manage.py is located — and run:

python manage.py startapp contact

Then, install it in your INSTALLED\_APPS variable inside the EmailProject/settings.py file:

# EmailProject/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

...

# Custom

'contact',

]

Before advancing with the contact app, let’s configure the urlpatterns inside of the EmailProject/urls.py file. To do this, import the django.urls.include function and include the contact URLs in the overall project. Don’t worry; we’ll configure the contact URLs later:

# EmailProject/urls.py

from django.contrib import admin

from django.urls import path, include # New import

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('contact.urls')) # Include Contact URLs

]

### **Contact Form**

Enter the contact app folder and create a forms.py file. It’s a good practice to define all of your forms inside of a forms.py file, but it isn’t mandatory. That’s why Django doesn’t include this file by default.

You can create the forms file with the following commands:

cd ../contact/

# You were inside the EmailProject folder

touch forms.py

Open the file you just created and make the following imports:

# contact/forms.py

from django import forms

from django.conf import settings

from django.core.mail import send\_mail

The Django [form module](https://docs.djangoproject.com/en/3.2/topics/forms/#the-django-form-class) gives us all the needed classes and fields to create our contact form. Once again we’re importing the settings object and the send\_mail function to send the emails.

Our contact form will contain several fields and use two custom methods: get\_info(), which formats the user-provided information, and send(), which will send the message.

Let’s see this implemented in code:

# contact/forms.py

class ContactForm(forms.Form):

name = forms.CharField(max\_length=120)

email = forms.EmailField()

inquiry = forms.CharField(max\_length=70)

message = forms.CharField(widget=forms.Textarea)

def get\_info(self):

"""

Method that returns formatted information

:return: subject, msg

"""

# Cleaned data

cl\_data = super().clean()

name = cl\_data.get('name').strip()

from\_email = cl\_data.get('email')

subject = cl\_data.get('inquiry')

msg = f'{name} with email {from\_email} said:'

msg += f'\n"{subject}"\n\n'

msg += cl\_data.get('message')

return subject, msg

def send(self):

subject, msg = self.get\_info()

send\_mail(

subject=subject,

message=msg,

from\_email=settings.EMAIL\_HOST\_USER,

recipient\_list=[settings.RECIPIENT\_ADDRESS]

)

This is a huge class, so let’s break down what we’re doing in each part. Firstly, we define four fields that will be required to send the message:

* name and enquiry are [CharFields](https://docs.djangoproject.com/en/3.2/ref/forms/fields/" \l "charfield) that represent the name and reason of the contact message.
* email is an [EmailField](https://docs.djangoproject.com/en/3.2/ref/forms/fields/" \l "emailfield) that represents the email of the person who’s trying to contact you. Take into account that the email won’t be sent by the email address of the user, but by the email address you set to send the emails in the Django project.
* message is another CharField with the exception we’re using the [Textarea](https://docs.djangoproject.com/en/3.2/ref/forms/widgets/" \l "textarea) widget. This means that, when displaying the form, it’ll render a <textarea> tag instead of a simple <input>.

Heading into the custom methods, we’re only using the get\_info method to format the information provided by the user and return two variables: subject, which is nothing but the inquiry field, and the message, which will be the actual message sent by Django.

On the other hand, the send() method only gets the formatted info from get\_info and sends the message with the send\_mail function.

Although this section was pretty large, you’ll see how we simplified the contact views by implementing all the sending logic to the ContactForm itself.

### **Contact Views**

Open the contact/views.py file and add the following imports:

# contact/views.py

from django.views.generic import FormView, TemplateView

from .forms import ContactForm

from django.urls import reverse\_lazy

As you can see, we’re going to use [Django generic views](https://docs.djangoproject.com/en/3.2/topics/class-based-views/), which saves us a ton of time when making simple tasks — for example, when setting up a form with FormView or creating a view that only renders a template with TemplateView.

Also, we’re importing the ContactForm that we built in the previous section and the [reverse\_lazy](https://docs.djangoproject.com/en/3.2/ref/urlresolvers/" \l "reverse-lazy) function used when working with class-based views.

Continuing with the views, let’s write the ContactView:

# contact/views.py

class ContactView(FormView):

template\_name = 'contact/contact.html'

form\_class = ContactForm

success\_url = reverse\_lazy('contact:success')

def form\_valid(self, form):

# Calls the custom send method

form.send()

return super().form\_valid(form)

As you can see, we’re building a simple [FormView](https://docs.djangoproject.com/en/3.2/ref/class-based-views/generic-editing/" \l "formview) using the ContactForm we created. We’re also setting up the template\_name and the success\_url. We’ll write the HTML template and set up the URLs later.

The [form valid](https://docs.djangoproject.com/en/3.2/ref/class-based-views/mixins-editing/#django.views.generic.edit.FormMixin.form_valid) method let us send the email using the ContactForm.send() method only if all the fields of the form are valid. This implies that if the user enters invalid input — such as an unformatted email address — the message won’t be sent.

The above form\_valid method implementation would be equivalent to the following in a function-based view:

# Previous function based contact\_view ...

if request.method == 'POST':

form = ContactForm(request.POST)

if form.is\_valid():

form.send()

return redirect('contact:success')

else:

form = ContactForm())

Ending this section, we’re going to write a ContactSucessView, which will show a success message to the user. Since we’ve already imported the TemplateView class, we only need to inherit from it and define the template\_name attribute:

# contact/views.py

class ContactSuccessView(TemplateView):

template\_name = 'contact/success.html'

You can check out the views.py file on the [GitHub repository](https://github.com/sitepoint-editors/Email-Django-app/blob/main/EmailProject/contact/views.py) in case you have any concerns.

### **Contact URLs**

It’s time to create the URL patterns of the contact app. Since Django doesn’t give us the urls.py file by default, we’ll need to create it with the following command (make sure to be inside the contact app folder):

pwd

# /path/to/project/EmailProject/contact

touch urls.py

Open that file and set up the app\_name and urlpatterns variables:

from django.urls import path

from .views import ContactView, ContactSuccessView

app\_name = 'contact'

urlpatterns = [

path('', ContactView.as\_view(), name="contact"),

path('success/', ContactSuccessView.as\_view(), name="success"),

]

We use [path](https://docs.djangoproject.com/en/3.2/ref/urls/#path) to include the route and its correspondent view to the URL configuration of the app. When we set the app\_name variable to 'contact', it means the URL namespacing of the app will look like this:

contact:name\_of\_path

# For ContactView

contact:contact

# For ContactSuccessView

contact:success

Note: a namespace is what we call URLs dynamically in Django templates and Views.

You can learn more about the Django URL dispatcher in the [official documentation](https://docs.djangoproject.com/en/3.2/topics/http/urls/).

### **Writing templates**

Django templates are the preferred way to display data dynamically, using [HTML](https://www.sitepoint.com/html-css/html/) and special tags that Django Template Language gives us.

For this specific app, we’ll be using three templates:

* base.html: all the other templates will inherit from it. It’ll contain all the HTML skeleton that all of the templates must have, as well as links to Bootstrap.
* contact.html: displays the contact form.
* success.html: displays a success message.

Let’s start by creating the contact’s app template structure (make sure you’re inside the contact app folder):

mkdir -p templates/contact/

cd templates/contact

touch base.html contact.html success.html

The commands above create the typical template structure of a reusable Django app — appname/templates/appname — and the tree template files I mentioned before.

Here’s what the app file structure should now look like:

.

├── admin.py

├── apps.py

├── forms.py

├── \_\_init\_\_.py

├── migrations

│ └── \_\_init\_\_.py

├── models.py

├── templates

│ └── contact

│ ├── base.html

│ ├── contact.html

│ └── success.html

├── tests.py

├── urls.py

└── views.py

Let’s jump into the content of the base.html template:

<!-- contact/templates/contact/base.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>Django Email Send</title>

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/css/bootstrap.min.css" rel="stylesheet"

integrity="sha384-wEmeIV1mKuiNpC+IOBjI7aAzPcEZeedi5yW5f2yOq55WWLwNGmvvx4Um1vskeMj0" crossorigin="anonymous" />

</head>

<body>

{% block body %}

{% endblock %}

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/js/bootstrap.bundle.min.js"

integrity="sha384-p34f1UUtsS3wqzfto5wAAmdvj+osOnFyQFpp4Ua3gs/ZVWx6oOypYoCJhGGScy+8" crossorigin="anonymous">

</script>

</body>

</html>

As you can see, it’s the simple skeleton of an HTML file that includes links to [Bootstrap 5](https://getbootstrap.com/docs/5.0/getting-started/introduction/). This allows us to stylize our contact app without using CSS files.

The {% block name-of-block %} tag allows us to set up a placeholder that “child templates” will utilize. The use of this tag makes [template inheritance](https://docs.djangoproject.com/en/3.2/ref/templates/language/#template-inheritance) an easy task.

Before heading into the forms, you’ll need to install the [Django crispy forms](https://django-crispy-forms.readthedocs.io/en/latest/) package, which allows us to stylize them easily:

pip install django-crispy-forms

Once again, crispy\_forms is a Django app, and we need to include it on the INSTALLED\_APPS list:

# config/settings.py

INSTALLED\_APPS = [

...

# 3rd party apps

'crispy\_forms',

# Custom apps

'contact',

]

# Indicates the frontend framework django crispy forms use

CRISPY\_TEMPLATE\_PACK = 'bootstrap4'

We use the template pack of Bootstrap 4, because the Bootstrap form classes are compatible between the 4th and 5th version (at the time of writing).

Now, let’s work on the contact.html template:

<!-- contact/templates/contact/contact.html -->

{% extends 'contact/base.html' %}

{% load crispy\_forms\_tags %}

{% block body %}

<div class="mx-auto my-4 text-center">

<h1>Contact Us</h1>

</div>

<div class="container">

<form action="" method="post">

{% csrf\_token %}

{{ form | crispy }}

<button class="btn btn-success my-3" type="submit">Send message</button>

</form>

</div>

{% endblock %}

Note how we extended the base template and make use of the block placeholder. This is what makes Django Template Language so efficient, as it lets us save a lot of HTML copying and pasting.

Talking about the form, we’re using the method "post", which means that our ContactView will process the data given by the user and send the email if the form is valid.

The {% csrf\_token %} is mandatory in every form because of security reasons. Django’s documentation has a dedicated page on [CSRF tokens](https://docs.djangoproject.com/en/3.2/ref/csrf/) and the reasons to use them when working with forms.

We’ll be rendering the form with the crispy template tag, which is why we loaded the crispy tags with {% load crispy\_forms\_tags %}.

Finally, let’s write the success.html template:

{% extends 'contact/base.html' %}

{% block body %}

<div class="mx-auto my-4 text-center">

<h1 class="fw-bolder text-success">We sent your message</h1>

<p class="my-5">You can send another in the <a href="{% url 'contact:contact' %}">contact page</a></p>

</div>

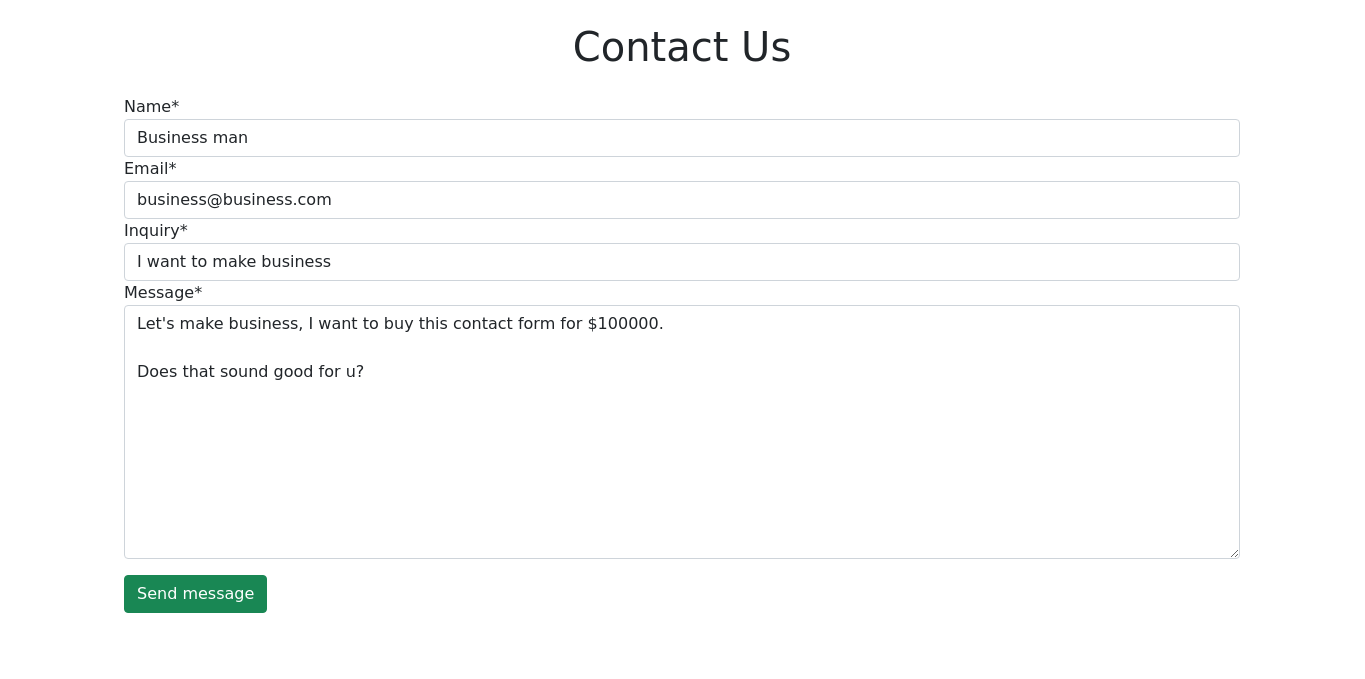
{% endblock %}

As you can see, it’s a simple success announcement that includes a link to the contact form in case the user wanted to send another message.

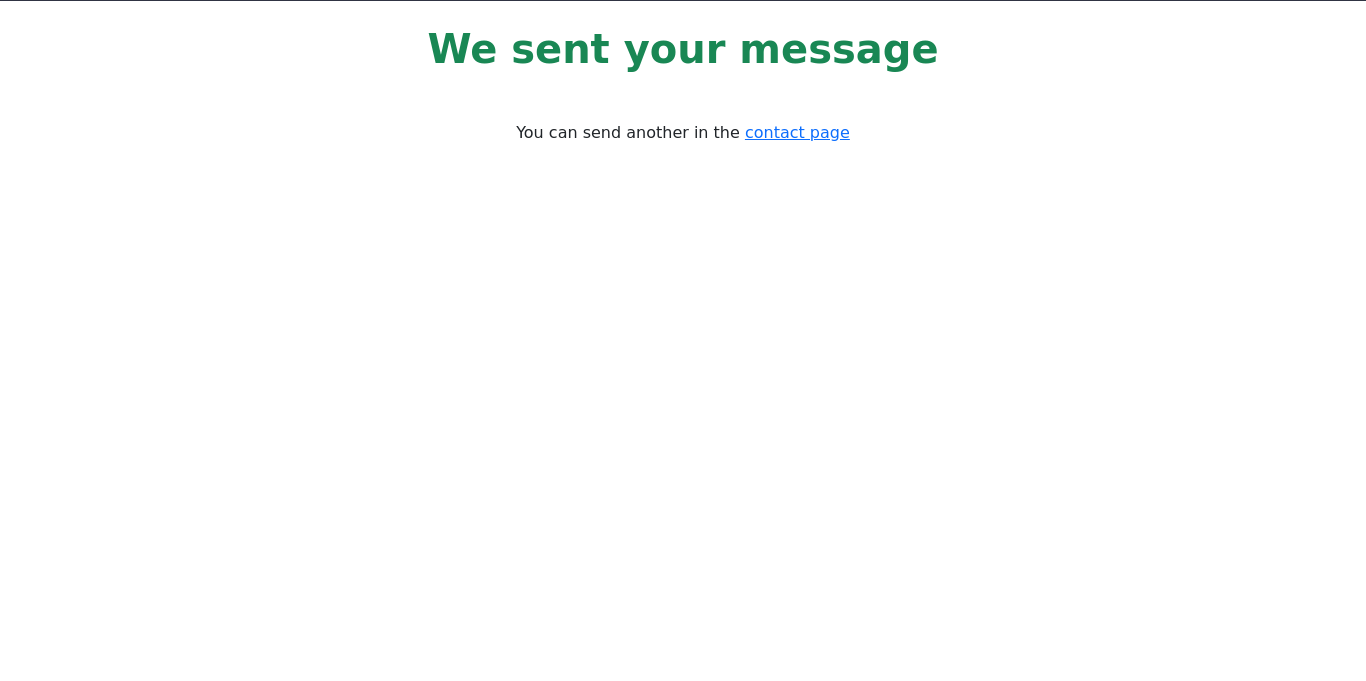
Let’s run the server again and visit [http://localhost:8000](http://localhost:8000/) (make sure you have the .venv activated and you’re inside the project root folder):

python manage.py runserver

The image below shows what the final contact form looks like.



And this is an image of the success message.



And here’s an image of the email in the inbox.

