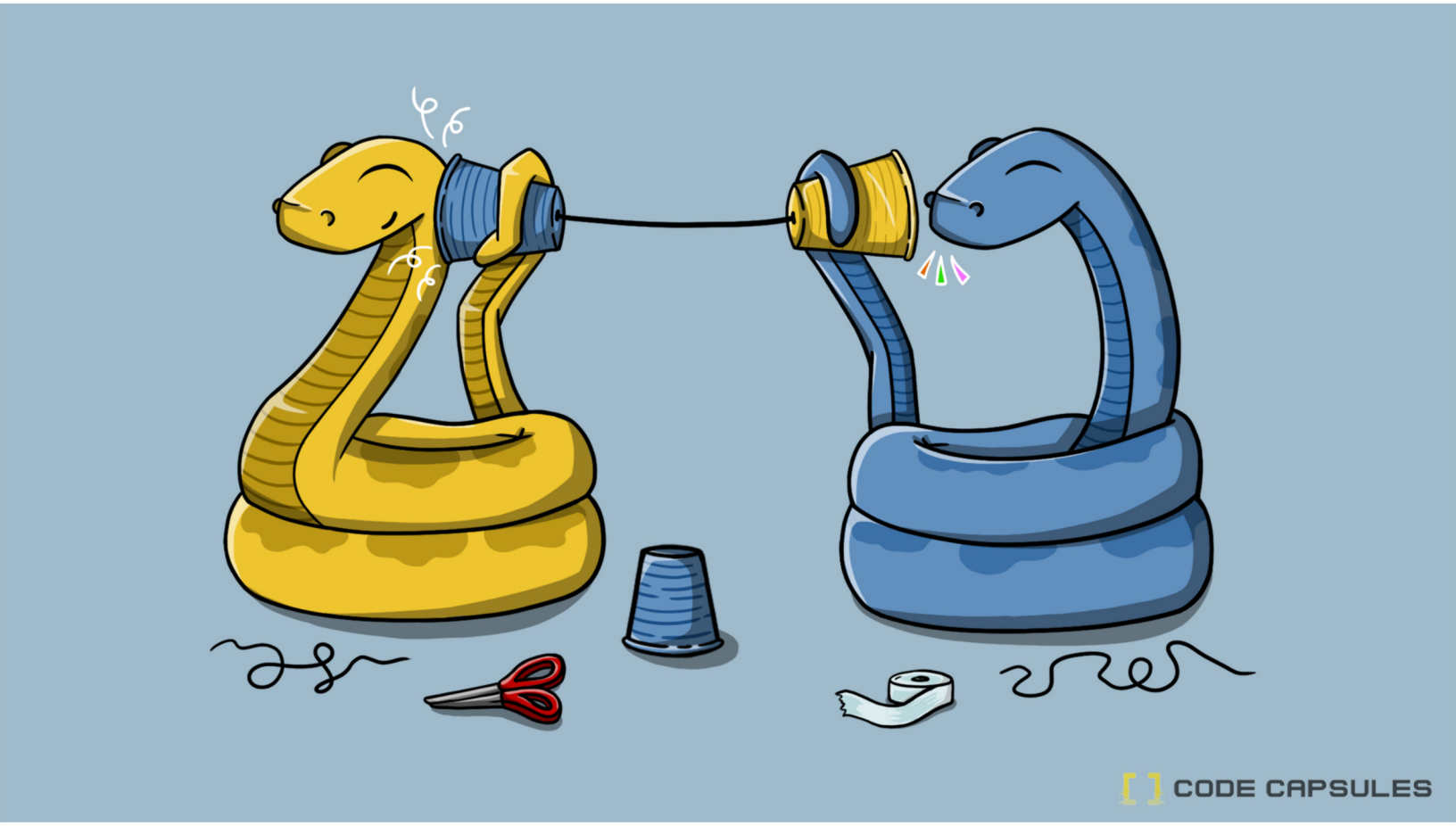
# Learn to Deploy Code Capsules




Build applications and git push to production with our simple and powerful DevOps platform.

# Learn to Deploy Code Capsules

Ritza

# Contents

# Hosting a Front-end: Building and Deploying Your Portfolio to Code Capsules

Publishing your portfolio online requires a solid technical background – managing servers can be challenging. You need to choose a server's operating system, maintain and update the server, and figure out where to host the server itself.

In this tutorial, we'll work with an alternative to the traditional method of hosting a front-end (content that visitors see when they load your website), called Code Capsules[1]. Code Capsules is a service that hosts front-end (and back-end) code online. Furthermore, Code Capsules:

- Manages all of the technical details – no server management required.
- Integrates with GitHub to deploy your code with a single `git push`.

First, we'll take a look at choosing a portfolio template and personalising it. After, we'll push the portfolio to a GitHub repository and see how Code Capsules connects to GitHub and makes your portfolio visible to the world.

## Requirements & Prerequisite Knowledge

Hosting a portfolio on Code Capsules requires no previous knowledge about servers or front-end development. To personalise a portfolio template and deploy it to Code Capsules, we'll need:

- A text editor, such as Sublime Text[2], or VSCode[3].
- A registered GitHub[4] account.
- The Git command-line interface[5] installed.

[1] https://codecapsules.io
[2] https://www.sublimetext.com/
[3] https://code.visualstudio.com/
[4] https://www.github.com
[5] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

# Creating a Portfolio

HTML5 UP[6] provides HTML site templates for free. We'll use the Massively[7] template – an easy to modify and elegant HTML template.

Follow these instructions carefully:

1. Download the Massively template.
2. Create a directory somewhere on your computer, then enter it.
3. **Within this directory, create another directory, and extract the Massively template files into it**.

This last step is necessary for hosting a web-page on Code Capsules. The file structure should look something like this:

```
myPortfolio
    portFolder
        + assets
        + images
        + generic.html
        + elements.html
        + index.html
```

The `index.html` file contains all of the HTML code for our portfolio – any changes to this code will result in a change to the portfolio. To view changes you make as we begin to modify the template, double click the `index.html` file to open the portfolio in a web-browser.

# Personalizing the Template

This tutorial will following the creation of a portfolio for Abraham Lincoln – the 16th president of the USA. We'll take a closer look at some things Abraham Lincoln wouldn't want in a portfolio – and maybe you too. The next few sections will cover how to modify the following elements of the portfolio template:

- Any not personalized text.
- The "Generic Page" and "Elements Reference" tabs.
- Pagination.
- The email contact form.
- Personal information (address, social media account).

Let's start with the title and subheading of the portfolio. Open the index.html file in your text editor. You'll see the following block of HTML near the top of the file:

[6] https://www.html5up.net
[7] https://html5up.net/massively

```
<head>
    <title>Massively by HTML5 UP</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1, user-scalabl\
e=no" />
    <link rel="stylesheet" href="assets/css/main.css" />
    <noscript><link rel="stylesheet" href="assets/css/noscript.css" /></noscript>
</head>
```

Change the text within the `<title>` tags to whatever you'd like, such as: "Abraham Lincoln". This is what will appear in search engines and browser tabs.

Now we'll change the text that displays at the top of the portfolio. Scroll down in your text editor, until you see the following code:

```
<!-- Intro -->
    <div id="intro">
        <h1>This is<br />
        Massively</h1>
        <p>A free, fully responsive HTML5 + CSS3 site template designed by <a href="\
https://twitter.com/ajlkn">@ajlkn</a> for <a href="https://html5up.net">HTML5 UP</a>\
<br />
        and released for free under the <a href="https://html5up.net/license">Creati\
ve Commons license</a>.</p>
        <ul class="actions">
            <li><a href="#header" class="button icon solid solo fa-arrow-down scroll\
y">Continue</a></li>
        </ul>
    </div>

<!-- Header -->
    <header id="header">
        <a href="index.html" class="logo">Massively</a>
    </header>
```

1. Customise the words wrapped in the `<h1>...</h1>` tags – this is the large text that displays at the top of the portfolio.
2. Change the text within the `<p>...</p>` tags to edit the subheading of the portfolio – the `<br />` tags and the `<a>...</a>` tags are safe to delete.
3. Delete the "Massively" button that appears as you scroll down the portfolio by deleting the three lines under the `<!-- Header -->` text wrapped in `<header>...</header>` tags.

Save the file and open it in a web browser. Our portfolio should now look something like this:


April 25, 2017

**image2**

Next, we'll take a look at deleting the date entries above each portfolio piece, removing the "Generic Page" and "Elements Reference" tabs, and modifying the social media links.

## Removing the tabs, dates, and links

The default layout for Massively is designed for a blog or news website containing articles. To look like a portfolio, we should delete the dates above each article entry. Do so by locating and deleting all lines beginning with `<span class="date"...>`.

To make this a single-page portfolio, we can delete the "Generic Page" and "Elements Reference" tabs by finding:

```
<li class="active"><a href="index.html">This is Massively</a></li>
<li><a href="generic.html">Generic Page</a></li>
<li><a href="elements.html">Elements Reference</a></li>
```

and deleting the last two lines. While we're at it, alter the title of the main tab by changing the "This is Massively" text.

The code for social media accounts is located at the top and bottom of the index.html file. Starting at the top, find this block of code:

```
<ul class="icons">
    <li><a href="#" class="icon brands fa-twitter"><span class="label">Twitter</span\
></a></li>
    <li><a href="#" class="icon brands fa-facebook-f"><span class="label">Facebook</\
span></a></li>
    <li><a href="#" class="icon brands fa-instagram"><span class="label">Instagram</\
span></a></li>
    <li><a href="#" class="icon brands fa-github"><span class="label">GitHub</span><\
/a></li>
</ul>
```

Delete any social media account you don't need – Abraham Lincoln doesn't have Instagram, so he would delete the following line to remove the Instagram link:

```
<li><a href="#" class="icon brands fa-instagram"><span class="label">Instagram</span\
></a></li>
```

If you'd like to link your social media accounts, enter the account link in place of the `#` in `href="#"`. For example, to link Abraham Lincoln's Twitter account, you'd edit the Twitter line like so:

```
<li><a href="https://twitter.com/Abe_Lincoln" class="icon brands fa-twitter"><span c\
lass="label">Twitter</span></a></li>
```

The social media code at the bottom of the `index.html` is nearly identical – follow this same process to edit the code at the bottom.

## Removing unnecessary content and further personalisation

In this section we'll:

- Remove the contact form
- Remove pagination

pagination


**pagination**

- Update or remove contact information

1. Remove the contact form found at the bottom of the portfolio by deleting the following code:

```
  <section>
    <form method="post" action="#">
        <div class="fields">
            <div class="field">
                <label for="name">Name</label>
                <input type="text" name="name" id="name" />
            </div>
            <div class="field">
                <label for="email">Email</label>
                <input type="text" name="email" id="email" />
            </div>
            <div class="field">
                <label for="message">Message</label>
                <textarea name="message" id="message" rows="3"></textarea>
            </div>
        </div>
        <ul class="actions">
            <li><input type="submit" value="Send Message" /></li>
        </ul>
    </form>
</section>
```

2. Remove pagination by deleting:

```
  <footer>
    <div class="pagination">
        <!--<a href="#" class="previous">Prev</a>-->
        <a href="#" class="page active">1</a>
        <a href="#" class="page">2</a>
        <a href="#" class="page">3</a>
        <span class="extra">&hellip;</span>
        <a href="#" class="page">8</a>
        <a href="#" class="page">9</a>
        <a href="#" class="page">10</a>
        <a href="#" class="next">Next</a>
    </div>
  </footer>
```

3. To delete specific contact information sections (the address section is shown below), delete the `<section>` tag, the information you want to delete, and the corresponding `</section>` tag.

```
<section class="alt">
    <h3>Address</h3>
    <p>1234 Somewhere Road #87257<br />
    Nashville, TN 00000-0000</p>
</section>
```

If you'd like to personalise your contact information instead, edit the text within the `<h3>...</h3>` tags and the `<p>...</p>` tags.

## Personalising portfolio pieces

Our portfolio is almost complete – we just need to personalise the actual portfolio pieces – customising the images, button links, and other text. Let's start with the images.

Gather any images you'd like to replace with the default images. Then, place your images in the `images` directory, located in the same directory as the `index.html` file.

You can swap images by finding lines wrapped in `<img.../>` tags, like the below line.

```
<img src="images/pic01.jpg" alt="" />
```

`images` is the name of the directory where you placed your image. To change the image, replace the text "pic01.jpg" with the **name and file extension** of your desired image.

Once we've replaced the images, we should change the text for each portfolio entry. Find code blocks wrapped in `<article>...</article>` tags, such as:

```
<article>
    <header>
        <h2><a href="#">Sed magna<br />
        ipsum faucibus</a></h2>
    </header>
    <a href="#" class="image fit"><img src="images/pic02.jpg" alt="" /></a>
    <p>Donec eget ex magna. Interdum et malesuada fames ac ante ipsum primis in fauc\
ibus. Pellentesque venenatis dolor imperdiet dolor mattis sagittis magna etiam.</p>
    <ul class="actions special">
        <li><a href="#" class="button">Full Story</a></li>
    </ul>
</article>
```

You can:

- Change the entry's title by editing the text within the `<h2>...</h2>` tags.
- Personalise the entry text by editing the Latin wrapped in the `<p>...</p>` tags.

- Customise buttons by finding the `<li><a href="#" class="button">Full Story</a></li>` lines and replacing the "#" with a link to your portfolio piece.
- Replace "Full Story" text with something more appropriate.

If you would like to remove a portfolio piece, delete the `<article>...</article>` tags and all the text wrapped in the tags.

Once finished with the portfolio, we need to push it to GitHub. After, the portfolio can be deployed to Code Capsules, making the portfolio publicly viewable.

# Uploading to GitHub

If you already know how to push code from a local repository to a remote repository on GitHub, push the **sub-directory** containing the portfolio to GitHub and skip to the next section.

Otherwise, we'll push (or send) our portfolio code to a GitHub remote repository (a place where your code stores on GitHub). Once complete, Code Capsules can connect to the repository and automatically "deploy" the portfolio online. Let's create the remote repository now.

## Creating the remote repository

Follow the steps below to create a remote repository on GitHub:

- Go to www.github.com and log in.
- Find the "Create new repository" button and click it.
- Name your repository anything (in this picture it was named "myPortfolio").
- Copy the URL given to you under "Quick setup".

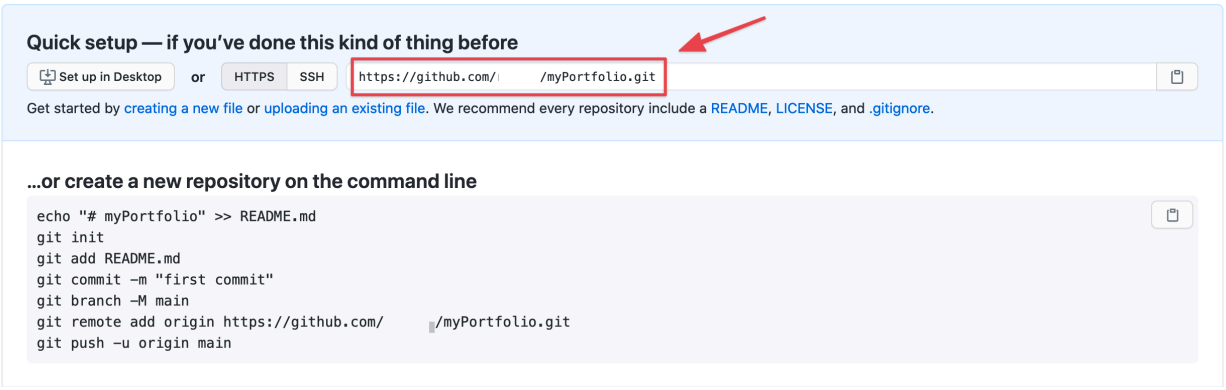



**image10**

Locate the link to your repository under "Quick Setup"

## Sending files to the GitHub repository

We've created the remote repository – now we need to push the portfolio to GitHub.

Open a terminal and navigate to the top-level directory containing the portfolio. This directory should contain the sub-directory that has all the portfolio files.

If your file structure looked like:

```
myPortfolio
    portFolder
        + assets
        + images
        + generic.html
        + elements.html
        + index.html
```

You would open the terminal in the `myPortfolio` directory. Not the `portFolder` directory.

Enter each command in order:

```
git init
git add .
git commit -m "First commit!"
git branch -M main
git remote add origin https://github.com/yourusername/yourrepositoryname.git
git push -u origin main
```

Replace the URL above with the URL to your remote repository (copied in the previous section).

Now you can see the portfolio code in your GitHub repository. Your repository should look similar to the below, where all of your portfolio code is contained in a sub-directory (in this image, the sub-directory is "portFolder"):

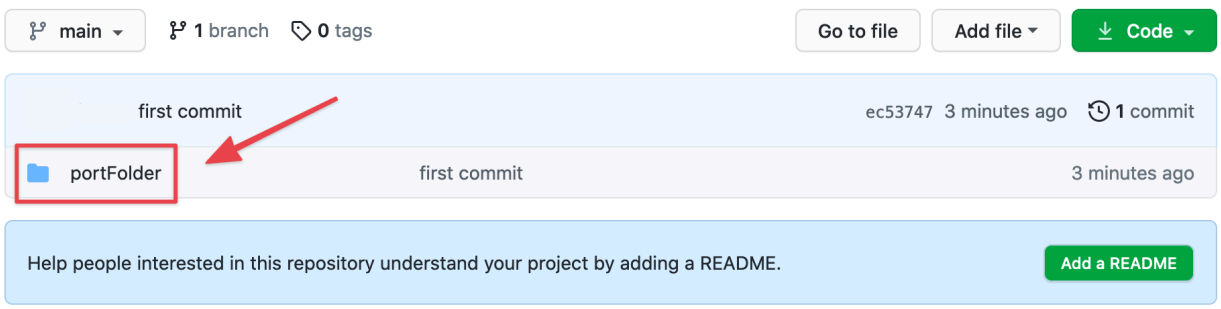


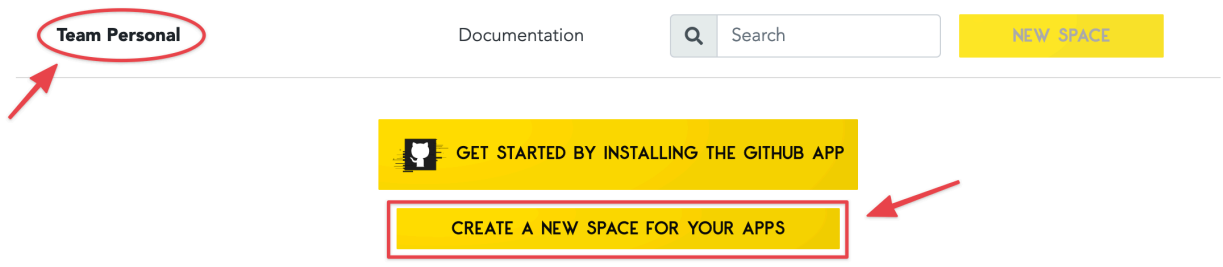

**image11**

Now Code Capsules can host the portfolio.

# Deploying to Code Capsules

To deploy the portfolio to Code Capsules, navigate to https://codecapsules.io/, create an account, and log in.

After logging in, you'll be greeted with a page that looks like the below.



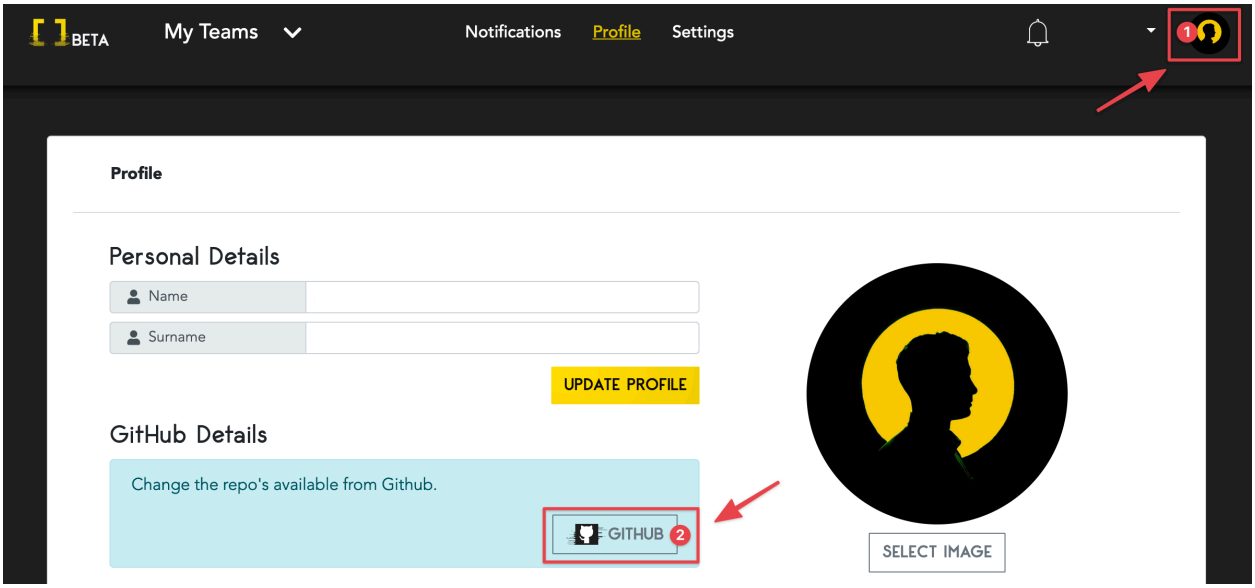

**team**

Notice the "Team Personal" at the top left. Every new account starts with a "Team Personal". Code Capsules provides Teams for collaborative development – you can invite other people to your Team and Team members can view and edit your web-applications. You can create other teams - but the default "Personal" team is fine for now.

At the center, you'll see a clickable box, labeled "Personal". This is called a Space. Spaces act as a further layer of organisation. Spaces can contain one or many Capsules (more on Capsules shortly) and can help organise large projects. We'll take a look at this space soon, but we first need to link Code Capsules to Github.

## Linking the repository

We need to give Code Capsules access to our portfolio. Click on your profile image at the top right of the screen, then find the "GitHub" button – click on it. Code Capsules will redirect you to GitHub.

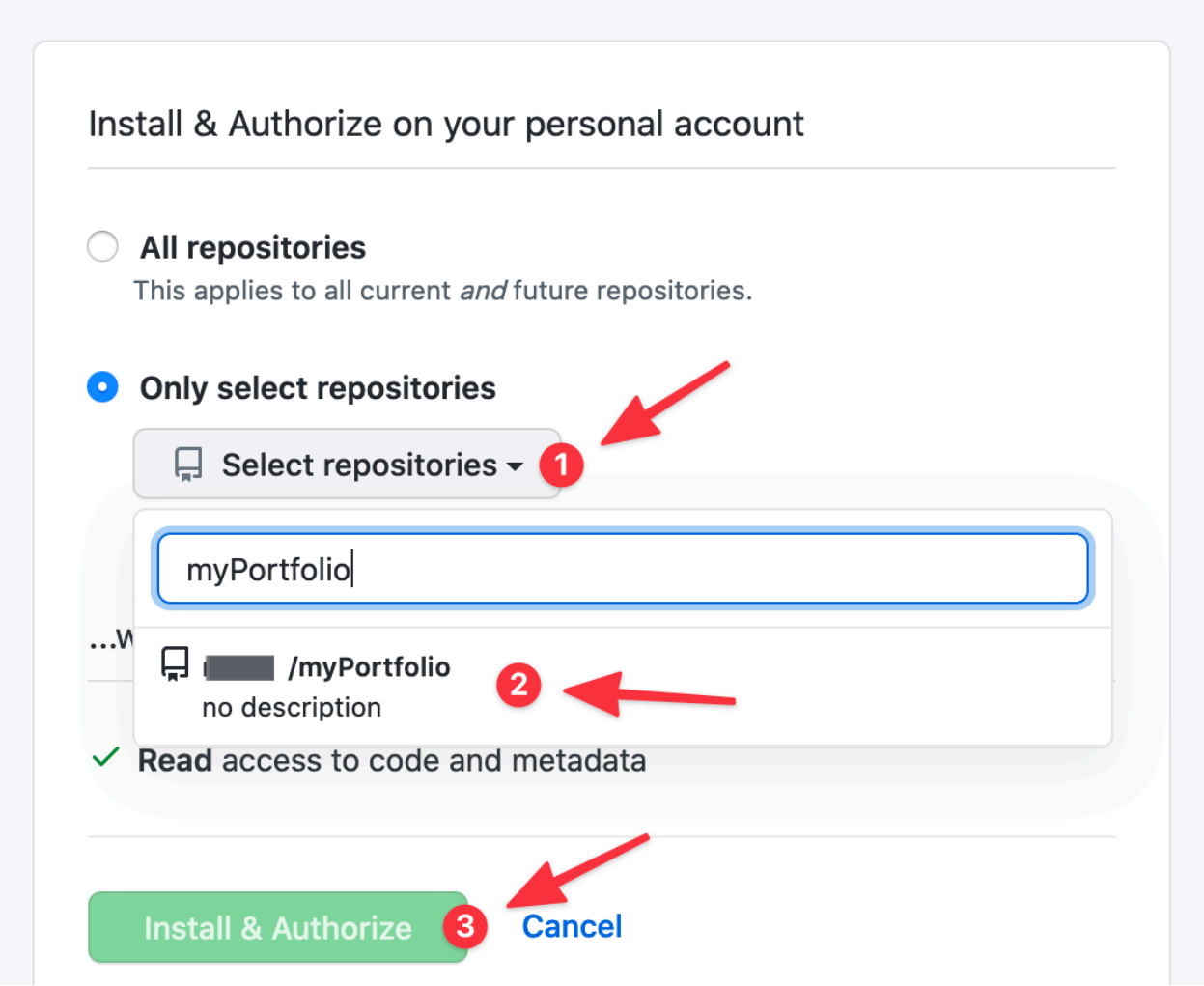



**GitHubButton**

Then:

1. Log in to GitHub.
2. Click your username.
3. Press “Only select repositories”.
4. From the drop-down menu, type the repository’s name containing your portfolio and select it.
5. Press “Install & Authorize”.

permissions_git




**permissions_git**

Now we can deploy our portfolio. Return back to your Team, and enter the space labeled “Personal”

## Creating Capsule, and viewing the portfolio

The last step to deploying the Portfolio is creating a Capsule. Capsules provide the server for your application or code – in our case, we’ll create a Capsule that’ll host our portfolio. Click “Create a new Capsule for your Space”.

You’ll be prompted to choose a Capsule type – our portfolio contains only front-end code, so choose a “Frontend” Capsule and:

1. Select the “Trial” product type.
2. Click the repository containing the portfolio.
3. Press “Next”.

4. Leave the build command blank and enter the name of the sub-directory containing the portfolio files in the “Static Content Folder Path” entry box.

written by Mozilla.

For any help with GitHub, take a look at their documentation[8].

[8] https://docs.github.com/en

# Customising Your Domain on Code Capsules

In this tutorial, we'll set up a custom domain name for your website or application hosted on Code Capsules.

## Why Custom Domains and How Do They Work?

Custom domains garner name recognition for your web-application or website. Consider the Google search-engine: without a domain, you would need to type in the IP[9] address for it. This would be far more difficult to remember than the URL www.google.com – instantly recognizable.

Web-addresses like www.google.com act as placeholders for an IP address and help us remember the website. When you type a URL in your search bar, your computer sends a request with the URL to the Domain Name System (DNS) – a cluster of servers worldwide containing domain names and corresponding IP addresses. The DNS then returns the URL's corresponding IP address, and you connect to the website you were trying to reach.

Following this guide, we'll learn how to buy a domain and route it to a Code Capsules hosted web-application. Along the way, we'll learn more about the DNS and related topics.

## Prerequisites

To complete this tutorial, we'll need:

- A web-application hosted on Code Capsules[10].
- A valid payment method (credit card, PayPal, cryptocurrency, bank transfer) to purchase a custom domain.

## Where to Buy a Domain

Domain Registrars are businesses accredited to sell domains. We'll purchase a domain from the registrar www.gandi.net. Some things to keep in mind when choosing a domain name:

[9] https://www.popularmechanics.com/technology/a32729384/how-to-find-ip-address/
[10] https://codecapsules.io

- Domains that don't contain highly sought after words[11] are usually inexpensive.
- You can save on domains by using less popular Top-level domains[12](TLD's) – for example: rather than register a ".com" website, register a ".info" website.

Keeping these tips in mind, let's purchase a domain.

## Purchasing a domain from Gandi

To purchase a domain on Gandi:

1. Navigate to www.gandi.net.
2. Enter the domain you want in the domain search box (ex: https://www.lincolnsportfolio.co.za)
3. Add the domain to the shopping cart.
4. Checkout by clicking the shopping cart at the top right of the screen.
5. Decide how many years you'd like to host the domain, and press the Checkout button.

Follow the prompts to create an account and purchase your domain. Then log in to Gandi.net with your new account and click the "Domain" button on the dashboard.

[11] https://webhostingcompare.co.za/most-expensive-domain-names-sold-south-africa/
[12] https://en.wikipedia.org/wiki/Top-level_domain

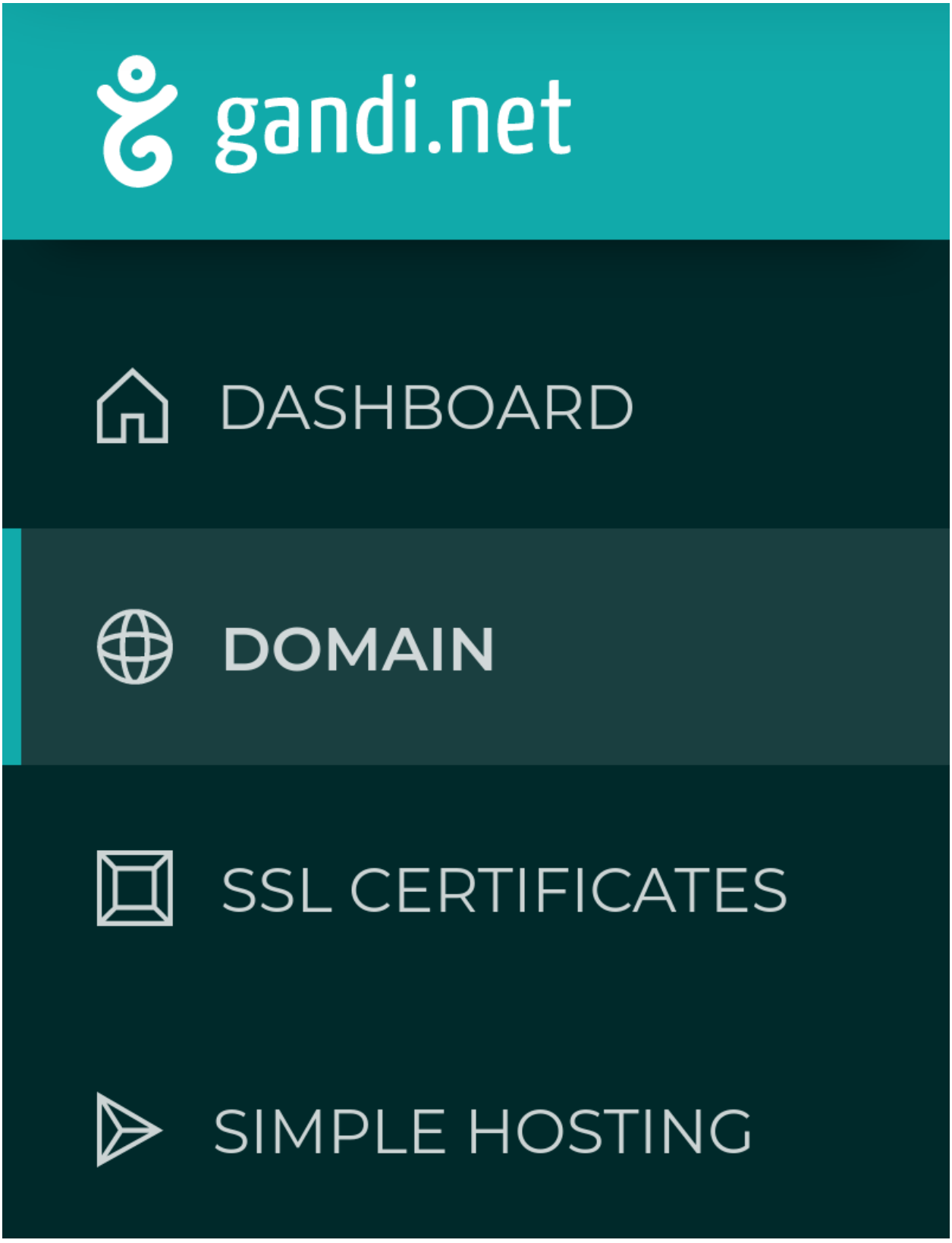

gandi.net
DASHBOARD
DOMAIN
SSL CERTIFICATES
SIMPLE HOSTING

**image3**

If Gandi has processed the domain, find it under the "Active" tab – if it's still processing, view it under the "Pending" tab. Processing a domain can take some time.

Before routing the new domain to a web application, we have something left to consider – the security of your web-application.

## HTTP vs. HTTPS

Domain names are one portion of a URL (Uniform Resource Locator) – Google's domain name is google.com[13], and the URL is http://www.google.com. Similarly, example.com[14] is a domain name, and http://www.example.com is the URL associated with it.

HTTP stands for Hypertext Transfer Protocol. When you see HTTP beginning a URL such as http://www.google.com, you know that the information retrieved by entering this address returns in clear text[15]. This means data is vulnerable when interacting with this website, presenting a problem for any website dealing with sensitive information. The alternative is HTTPS – Hypertext Transfer Protocol Secure.

HTTPS encrypts data sent between you and the server that you're connected to. Because of the security risks associated with HTTP[16], many websites "force" an HTTPS connection. Try entering http://www.google.com in your web browser. You'll notice the `http` portion automatically becomes `https`.

Like the Google example, we'll make sure that if a user attempts to connect via `http://www.yourwebsitehere.com`, they'll redirect to `https:/www.yourwebitehere.com`.

Setting up HTTPS is a quick process with Gandi – let's do that for your domain.

## Setting Up HTTPS for Your Domain

To set up HTTPS with the domain, we need to register a free SSL (Secure Sockets Layer) certificate. In short, an SSL certificate helps encrypt the data sent when connected via HTTPS.

To register an SSL certificate for our domain we must:

1. Click on the domain under the **active** tab.
2. Navigate to the **Web Forwarding** tab.
3. Click **Create** at the top right.
4. From the **Address drop-down menu**, choose "HTTP:// + HTTPS://"
5. Type "www" in the textbox to the right.
6. From the **Address to forward to** drop-down menu, choose "HTTPS://"

[13] https://google.com
[14] https://example.com
[15] https://www.pcmag.com/encyclopedia/term/cleartext
[16] https://www.w3.org/Protocols/rfc2616/rfc2616-sec15.html

7. Type in the name of the domain.
8. Choose “Permanent” under **Type of web forwarding**

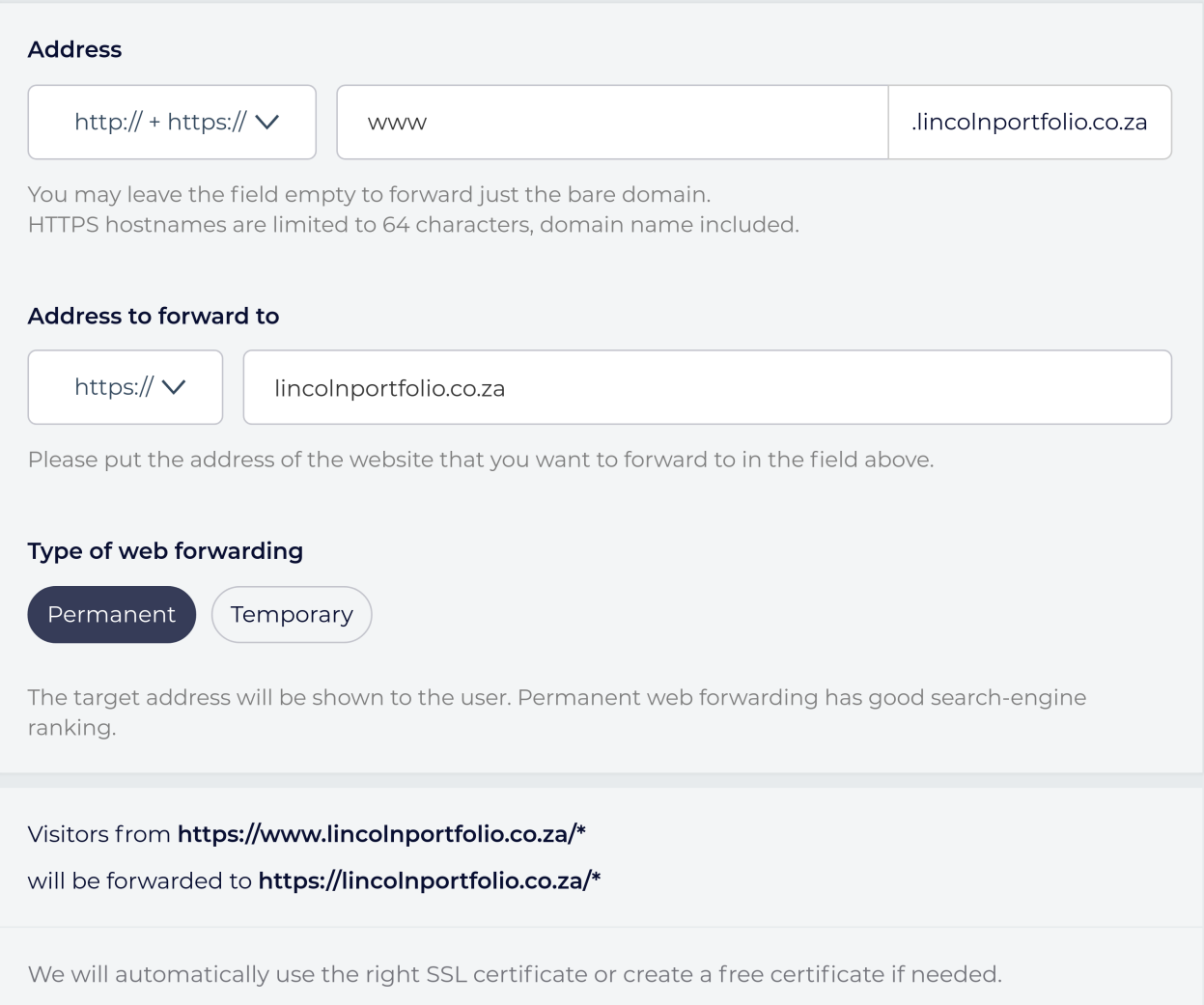



**image4**

9. The above image shows an example form – click create when done.
10. Repeat this process, but choose “http://” in the **Address drop-down** and type “*” in the textbox next to it.

This will forward any users connect to `http://www.yourdomainhere.tld` or `http://yourdomainhere.tld` to `https://yourdomainhere.tld` – it forces users to take advantage of HTTPS. After creating this forwarding address, Gandi automatically creates an SSL certificate. This can take some time to process.

You’ll need to verify your email address with Gandi before receiving the SSL certificate, so check your email for a verification link from Gandi.

Now that the domain has an SSL certificate, we'll route your Code Capsules web-application to the domain. Navigate to your domain on the Gandi dashboard.

# Routing Your Web-application to The Domain

Click the "DNS Records" tab at the top of the page. DNS records contain your domain's "information". When users enter your domain in their search bar, their computer will receive these records (or information).

Gandi supplies numerous DNS records with default values upon domain creation. We'll only concern ourselves with entries containing the "A" and "CNAME" types.

An A record stores the IP address of the server that hosts your web-application (in this case, Code Capsules). When you type in a domain name, your computer requests the A record associated with the domain from the DNS. The DNS returns the A record containing the IP address – this is what you finally connect to.

Let's modify the default A record to route to your web-application:

1. On Code Capsules[17], navigate to the Capsule you wish to route to your new domain.
2. Click **Overview** then **Add A Custom Domain**.
3. Copy the supplied IP address and type in the name of the web-address purchased.
4. Click **Create Domain**.
5. At the DNS record tab in domain view on Gandi, edit the entry with "A" as the type.
6. Enter "@" for its name and paste the Code Capsules supplied IP address in the IPv4 address text box.
7. Click create.

**It may take up to 3 hours for these changes to process**. View your web-application by typing `https://yourdomainname.tld`, replacing your domain name with "yourdomainname" and ".tld" with your extension (such as .com).

Notice that if you type `https://www.yourdomainhere.tld`, you'll receive a 404 error. To fix this, we'll add a new "CNAME" record. A CNAME is like an alias for a domain – we're going to create one that tells the DNS that it should direct users who enter the leading "www." to the same place as those who leave it out.

To allow users to enter in "www." before your domain name:

1. Return to Code Capsules and press the **Add A Custom Domain** button again.
2. Under domain name, enter `www.yourdomainname.tld`, replacing your name and TLD appropriately.
3. Return to the DNS record tab on Gandi, and press **Add** at the top right.

[17] https://codecapsules.io

4. Choose the CNAME type.
5. Enter “www” in the name text-box.
6. Type your default Code Capsules web-application URL under **Hostname** (find this in the “Overview” tab in your web-application’s Capsule), with a period at the end. It should look like the below:

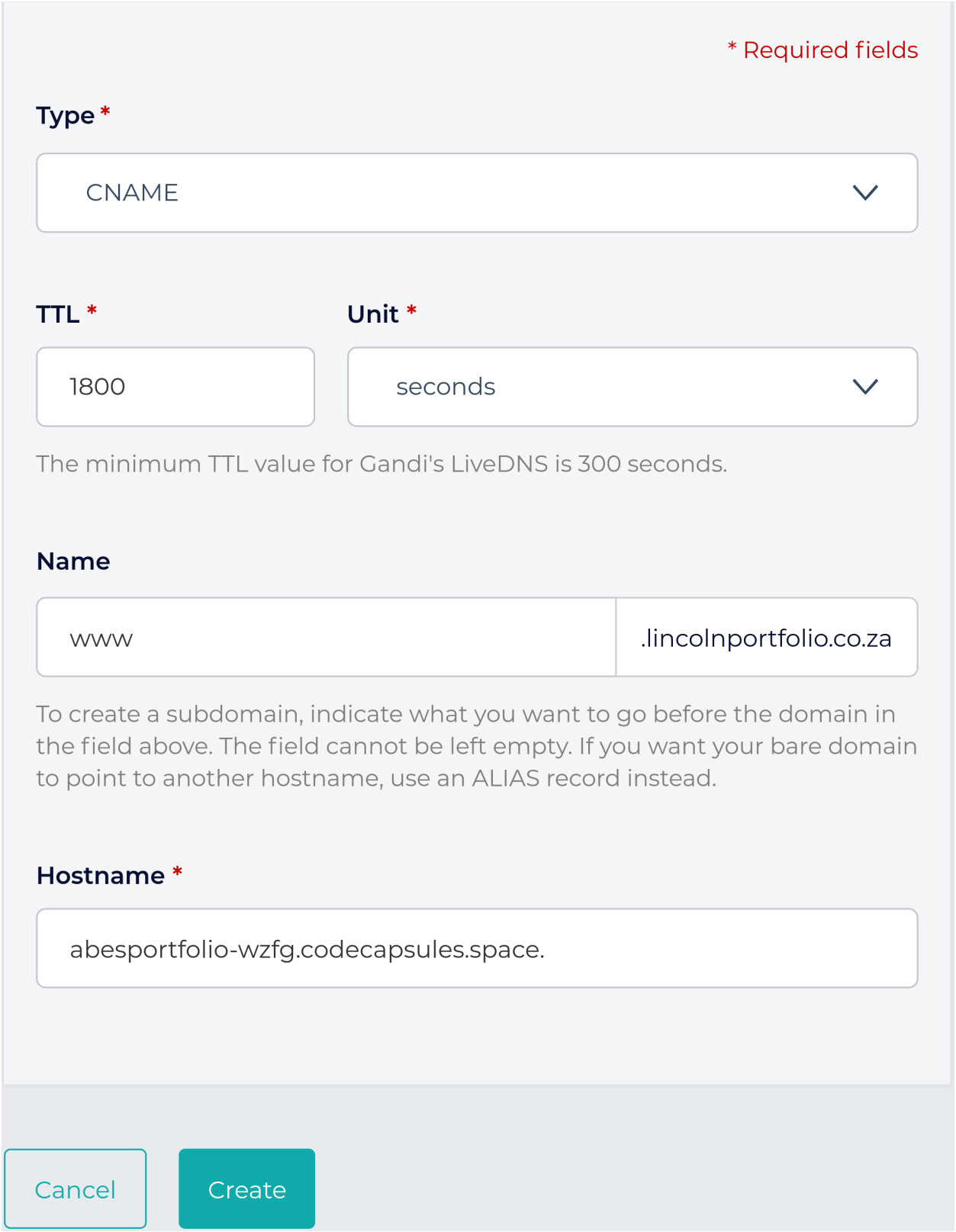

* Required fields
Type *
CNAME
TTL *
1800
Unit *
seconds
The minimum TTL value for Gandi's LiveDNS is 300 seconds.
Name
www
.lincolnportfolio.co.za
To create a subdomain, indicate what you want to go before the domain in the field above. The field cannot be left empty. If you want your bare domain to point to another hostname, use an ALIAS record instead.
Hostname *
abesportfolio-wzfg.codecapsules.space.
Cancel
Create

image2

7. Click create.

You can now view your web-application by entering either `https://yourdomainname.tld` or `https://www.yourdomainname.tld`. Once more, it **may take up to 3 hours for these changes to process**.

# What Next?

We've learned how to purchase, secure, and configure a domain, route a domain to your Code Capsules application, and even a little bit about DNS.

If you're interested, there is still a lot to learn about DNS. A fine place to start is Amazon Web Services' page on DNS[18].

If you'd like to know about the rest of the DNS records associated with your new domain, this Google help page[19] contains a good overview

Finally, if you'd like to read more about how the HTTP protocol works, this Mozilla Developers Network page[20] is a good place to start.

[18] https://aws.amazon.com/route53/what-is-dns/
[19] https://support.google.com/a/answer/48090?hl=en
[20] https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

# Creating and Hosting an API with Flask and Code Capsules

An API, or Application Programming Interface, is a tool enabling developers to interact with data online. Imagine: you navigate to some website and see your location's temperature displayed on their homepage. How did they present this information?

Without a doubt, they used an API. APIs are hosted on a server and operate as an access point between the user and some data.

Part of this guide takes a look at the WeatherStack[21] API – an API providing weather data. For the website to retrieve your location's temperature, they would've sent a request to an API like WeatherStack. In the request, they would include information about your computer's location. WeatherStack's API would then return weather data related to your locale, such as the temperature and cloud cover. The weather website will then display this data on their homepage for you to view.

In this tutorial, we'll learn how to create a personal API with Python (using Flask[22]). Our API will use data from the WeatherStack[23] and OpenExchangeRates[24] APIs to give us up-to-the-minute USD exchange rates and the temperature of a given city.

We'll host our API on Code Capsules[25] so that anyone will be able to request information from it, no matter their location.

## Prerequisites

Before starting, we'll need a GitHub[26] account and knowledge of how to push code from a local repository to a remote repository[27].

Also ensure you've installed the following:

- Git[28]
- Python[29] 3.XX+
- Virtualenv[30]

[21] https://weatherstack.com/
[22] https://palletsprojects.com/p/flask/
[23] https://weatherstack.com/
[24] https://openexchangerates.org/
[25] https://codecapsules.io/
[26] https://github.com
[27] https://docs.github.com/en/github/importing-your-projects-to-github/adding-an-existing-project-to-github-using-the-command-line
[28] https://git-scm.com/downloads
[29] https://www.python.org/downloads/
[30] https://virtualenv.pypa.io/en/latest/installation.html

# Setting Up Our Environment

First, let's set up a virtual Python environment using Virtualenv. Virtualenv provides a clean Python install with no third-party libraries or packages, allowing us to work on this project without interfering with the dependencies of our other projects.

1. Open your terminal and create an empty folder.
2. Navigate to the folder via your terminal, and enter `virtualenv env`.

To activate the virtual environment, enter one of the following:

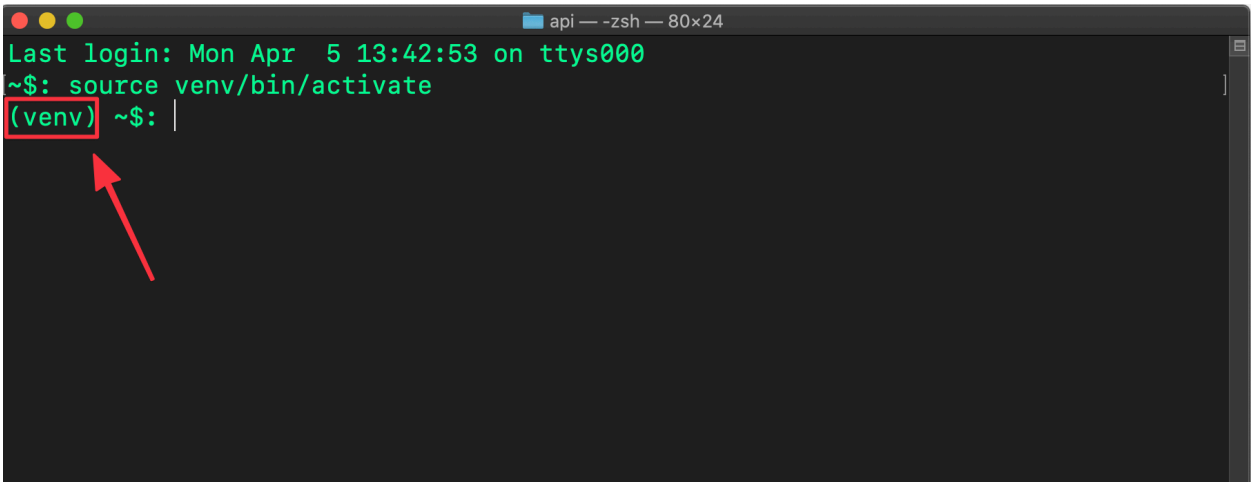
**Linux/MacOSX**

```
source env/bin/activate
```

**Windows**

```
\env\Scripts\activate.bat
```

If the virtual environment has activated correctly, you'll see `(env)` to the left of your name in the terminal.




**command prompt**

## Installing the Dependencies

Now that we've activated the virtual environment, let's take a look at the packages we'll use to create our API:

- Flask[31] is a minimal web development framework for Python. Flask provides resources and tools for building and maintaining web applications, websites, and more.
- Gunicorn[32] is a WSGI[33] server that will help serve our Python application (the API hosted on Code Capsules).
- Requests[34] is a Python library we will use to interact with APIs.

From your terminal where you activated the virtual environment, install these packages with `pip3 install flask gunicorn requests`.

# Registering Accounts on OpenExchangeRates and WeatherStack

Our API will return the current temperature of a chosen city and the USD exchange rates for three currencies. We'll create our API by combining data from two other APIs – WeatherStack[35] and OpenExchangeRates[36]. As their names suggest, WeatherStack will provide the temperature data, and OpenExchangeRates the exchange rate data.

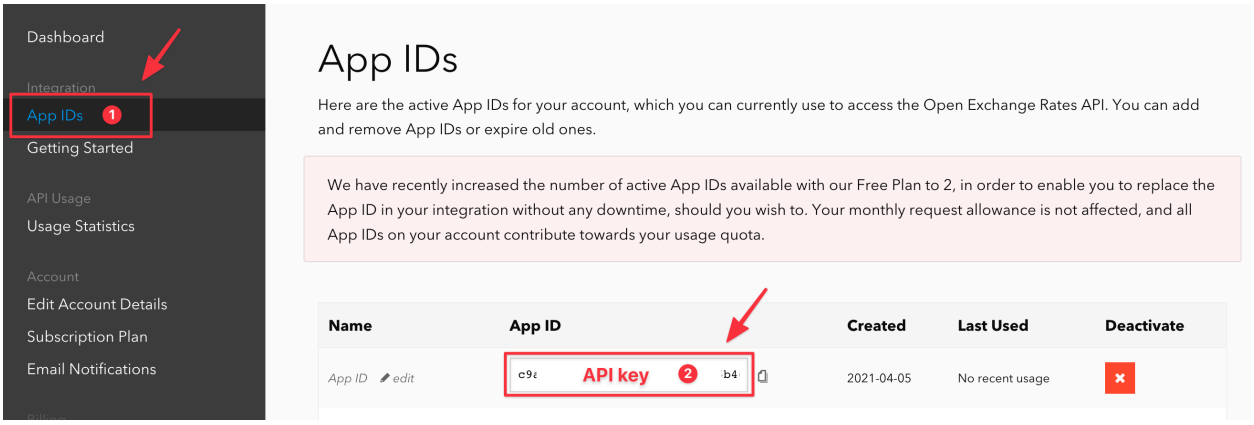
Registering an account is required so that we can receive a unique API key. An API key is a password that lets us use a particular API. In APIs with more sensitive data, these are used to prevent unauthorised access, but for open APIs like WeatherStack and OpenExchangeRates, they're used for rate limiting[37] to prevent users from sending too many requests at once and overwhelming the system.

## Creating our accounts

First, let's register an account on OpenExchangeRates. Navigate to https://openexchangerates.org/signup/free and:

1. Sign up and log in.
2. On the dashboard, click "App IDs".
3. **Save** your "App ID" (API key) on your computer.

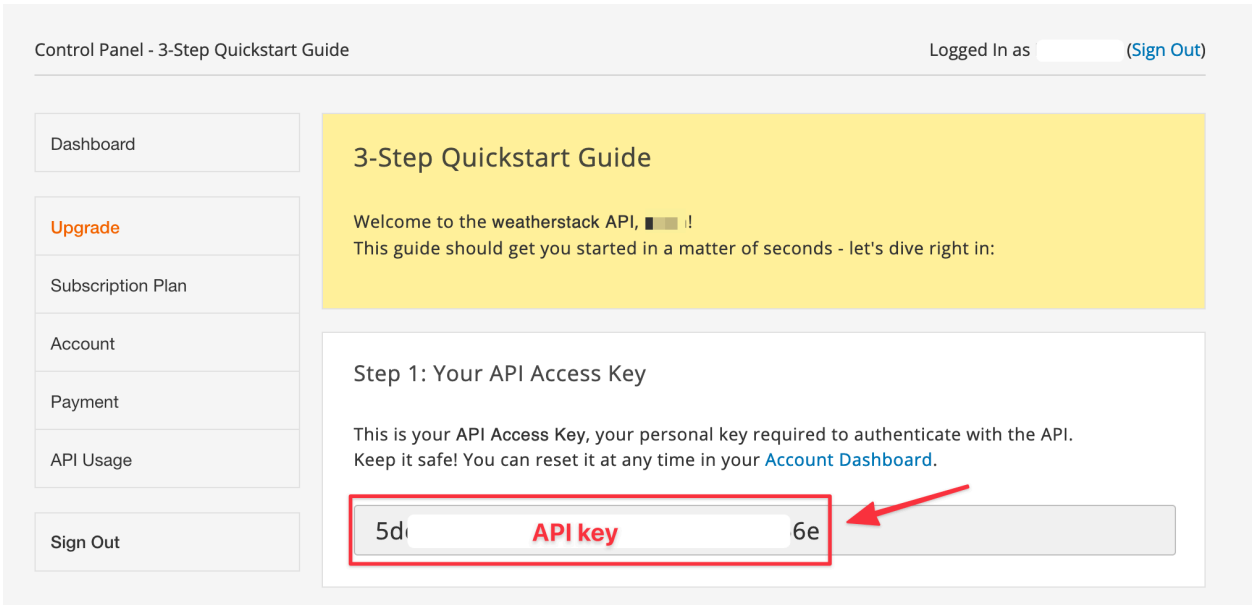
[31] https://palletsprojects.com/p/flask/
[32] https://gunicorn.org/
[33] https://medium.com/analytics-vidhya/what-is-wsgi-web-server-gateway-interface-ed2d290449e
[34] https://pypi.org/project/requests/
[35] https://weatherstack.com
[36] https://openexchangerates.org/
[37] https://en.wikipedia.org/wiki/Rate_limiting




**OpenExchangeRates api key**

Obtaining the WeatherStack API key is similar:

com/product)
Now we can retrieve data from the OpenExchangeRates and WeatherStack APIs using our API keys. Let's try that out now.
2. Log in and save the API key presented to you.




**WeatherStack api key**

Now we can retrieve data from the OpenExchangeRates and WeatherS

## Getting exchange rates

First, let's see how requesting data from OpenExchangeRates works. Create a file named `app.py` and open it.

To request data from an API, we need an *endpoint* for the type of data we want. APIs often provide multiple endpoints for different information – for example, a weather API may have one endpoint for temperature and another for humidity.

In the code below, the `EXCHANGE_URL` variable contains the OpenExchangeRates endpoint for retrieving the latest exchange rates. Enter it in your `app.py` file now, replacing `YOUR-API-KEY-HERE` with the **OpenExchangeRates** API key you saved earlier.

```
import requests

EXCHANGE_URL = 'https://openexchangerates.org/api/latest.json?app_id=YOUR-API-KEY-HE\
RE'
exchange_data = requests.get(EXCHANGE_URL)
```

Note that we are including a secret API key in our codebase, which is bad practice. In later tutorials, you'll see how to use environment variables with Code Capsules for better security.

In this code, we're using the `requests` module to fetch data from the API. It does this over HTTPS, the same way your browser would. In fact, if you copy the value of `EXCHANGE_URL` to your browser now, you'll see exactly what data your code is fetching.

Note the format of the URL:

- `https://openexchangerates.org` is the website.
- `/api/` is the path containing the API portion of the website.
- `latest.json` is the API endpoint which returns the latest exchange rates.
- `?app_id=YOUR-API-KEY-HERE` specifies our password for accessing the API.

OpenExchangeRates has many other endpoints, each of which provides a different set of data. For example, you could request data from the `historical` endpoint (`https://openexchangerates.org/api/historical/`) to access past exchange rates.

Now let's print the data using the `.json()` method. This method converts the data from raw text into in JSON[38] (Javascript Object Notation), which we can work with like a Python dictionary.

```
print(exchange_data.json())
```

When running the program, you will see a lot of output. This is because we are currently retrieving every exchange rate OpenExchangeRates provides. Let's modify the code to only receive exchange rates from USD to EUR, CAD, and ZAR.

Add the following lines below `EXCHANGE_URL`:

[38] https://www.json.org/json-en.html

```
EXCHANGE_PARAMS =  {'symbols':'ZAR,EUR,CAD'}

exchange_data = requests.get(EXCHANGE_URL, EXCHANGE_PARAMS)
```

Then change your `print` statement as follows:

```
print(exchange_data.json()['rates']) # Print only exchange rates
```

Now we've included an `EXCHANGE_PARAMS` variable. Providing parameters to an API endpoint will alter which data is retrieved. The parameters available will depend on the API endpoint. You can find a list of parameters for the `latest` endpoint here[39].

In our case, we supplied the parameter `symbols` with the three currencies we want data for. When you run the program again, you should only see three exchange rates.

## Getting the temperature

Now that we've obtained the exchange rates, we can retrieve the temperature for a city. Let's modify the program by adding the following below the `print` statement. Make sure to replace `YOUR-API-KEY-HERE` with the **WeatherStack** API key.

```
WEATHER_URL = 'http://api.weatherstack.com/current?access_key=YOUR-API-KEY-HERE'
WEATHER_PARAMS = {'query':'Cape Town'}

weather = requests.get(WEATHER_URL, params=WEATHER_PARAMS)

print(weather.json()['current']['temperature']) # will print only the temperature; p\
rint without indexing to see all the values returned!
```

Here we retrieve the temperature for Cape Town, South Africa. You can replace "Cape Town" with another city of your choice to see its temperature.

# Creating our API

Now we'll get to creating the API with Flask. Our API will package the WeatherStack and OpenExchangeRates data together in a single endpoint.

This means we can build other applications later which will be able to retrieve all of the data above by calling `requests.get(MY_CODE_CAPSULES_URL)`.

## Beginning steps with Flask

First, we can remove all the print statements in our `app.py` file. Afterwards, edit the file accordingly:

[39] https://docs.openexchangerates.org/docs/latest-json

```
import requests
from flask import Flask, jsonify

EXCHANGE_URL = 'https://openexchangerates.org/api/latest.json?app_id=YOUR-API-KEY-HE\
RE'
EXCHANGE_PARAMS =  {'symbols':'ZAR,EUR,CAD'}

WEATHER_URL = 'http://api.weatherstack.com/current?access_key=YOUR-API-KEY-HERE'
WEATHER_PARAMS = {'query':'Cape Town'}

app = Flask(__name__)

@app.route('/') # Create main page of web-application
def index():
    return "Welcome to my API!" # Display text on main page

if __name__ == '__main__':
        app.run() # Run the application
```

After instantiating a Flask object, we add `@app.route('/')`. The `@` symbol is known as a Python decorator[40] – their use isn't very important for our application. Just understand that the below creates the homepage for your API:

```
@app.route(`/`)
def index():
    return "Welcome to my API!"
```

Once it's hosting it on Code Capsules, you'll see "Welcome to my API!" when you visit its URL.

Next, we'll implement the ability to "get" (using `requests.get()`) our data from the API when it's hosted.

## Combining the APIs

We've already written code to retrieve our data – now we just need to combine it and create an endpoint to fetch it. We'll do this by creating a new endpoint called `/get` that returns our selected data.

[40] https://realpython.com/primer-on-python-decorators/

```
@app.route('/get',methods=['GET']) # Add an endpoint to access our API
def get():
    exchange_data = requests.get(EXCHANGE_URL, EXCHANGE_PARAMS)
    weather = requests.get(WEATHER_URL,params=WEATHER_PARAMS)

    return jsonify({
        'usd_rates': exchange_data.json()['rates'],
        'curr_temp': weather.json()['current']['temperature']
    })
```

`@app.route('/get', methods=['GET'])` adds an endpoint, `/get`, allowing us to retrieve data from the API. When Code Capsules gives us a URL for our API, we'll be able to use this URL plus the endpoint `/get` to retrieve data from our API, combining the inputs from the two APIs we are calling out to in turn.

Next, the statement below returns our data in JSON:

```
return jsonify({
    'usd_rates' : exchange_data.json()['rates'],
    'curr_temp' : weather.json()['current']['temperature']
    })
```

Here, the exchange rate data is stored under `'usd_rates'` and the temperature data under `curr_temp`. This means that if we request our data and store it in a variable like `my_data`, we'll be able to print out the exchange rates by executing `print(my_data['usd_rates'])`, and print the temperature by executing `print(my_data['curr_temp'])`.

The API is complete – only a few steps left before hosting it on Code Capsules.

## Freezing Requirements and Creating the Procfile

Before sending our API to GitHub (so Code Capsules can host it), we need the `requirements.txt` file, and a Procfile.

The `requirements.txt` file contains information about the libraries we've used to make our API, which will allow Code Capsules to install those same libraries when we deploy it. To create this file, first ensure your terminal is still in the virtual environment. Then, in the same directory as the `app.py` file, enter `pip3 freeze > requirements.txt` in your terminal.

Next, create a new file named `Procfile` in the same directory. Open the `Procfile` and enter:

```
web: gunicorn app:app
```

This tells Code Capsules to use the Gunicorn WSGI server to serve the HTTP data sent and recieved by our Flask API.

# Hosting the API on Code Capsules

The API is now ready to host on Code Capsules. Follow these steps to get it online:

1. Create a remote repository on Github.
2. Push the `Procfile`, `requirements.txt`, and `app.py` files to the repository.
3. Link the repository to your Code Capsules account.
4. Create a new Team and Space (as necessary).

With the repository linked to Code Capsules, we just need to store the API on a Capsule:

1. Create a new Capsule.
2. Choose Backend Capsule and continue.
3. Select your product type and GitHub repository, click next.
4. Leave the "Run Command" field blank (our `Procfile` handles this step).
5. Create the Capsule.

Once the Capsule has built, the API is hosted! Let's take a quick look at how to interact with it.

## Viewing and interacting with our API

Once the Capsule has been built, Code Capsules will provide you with a URL (found in the "Overview" tab). Enter the URL in your browser, and you'll be greeted with "Welcome to my API!". To view the API data, add `/get` to the end of the URL.

Depending on your browser (Google Chrome was used below), you'll see something like this:


```
{"curr_temp":24,"usd_rates":{"CAD":1.271903,"EUR":0.815084,"ZAR":15.369514}}
```


**image4**

Now try interacting with the API through code. In a new file, enter the following, replacing the URL with your Code Capsules URL (ensure `/get` is at the end of the URL):

```
import requests

MY_URL = 'https://my-code-capsules-url.codecapsules.space/get'

api_data = requests.get(MY_URL)

print(api_data.json())
```

All done!

# Further Reading

We've learned a lot about APIs; how to interact with them, how to use API endpoints, and how to create and host an API with Flask and Code Capsules. If you'd like a more in-depth look at APIs, check out this article[41].

If you're interested in learning more about Flask or want to know what else you can do with it, start with their tutorial[42] or their documentation[43].

[41] https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/
[42] https://flask.palletsprojects.com/en/1.1.x/tutorial/
[43] https://flask.palletsprojects.com/en/1.1.x/

# Adding Functionality to Your Web Application: Setting up Stripe Checkout and Email Subscription with Flask and Code Capsules

## What We'll Cover

Constructing a frontend for your web application is the first step towards providing an interactive user experience. The next step is building a working backend, or making your web application functional. Buttons are nice to have, but it's more interesting to have those buttons do something. That's what we'll focus on today.

Through a step-by-step process, we'll develop this functionality. We'll use Flask[44] and a frontend template to create a web application that allows users to buy products through Stripe Checkout[45] (a tool for creating a "checkout" process for products) and subscribe to an email list with help from the Mailgun[46] email API service.

Then, we'll host the web application on Code Capsules so people around the world can buy your product and subscribe to your mailing list.

## Requirements

To successfully complete this project, we'll need:

- A text editor (like Sublime[47] or VSCode[48]) installed.
- Python 3.XX+[49] installed.
- Virtualenv[50] installed.
- Git[51] installed and a GitHub[52] account.
- A Code Capsules[53] account.

[44] https://flask.palletsprojects.com/en/1.1.x/
[45] https://stripe.com/payments/checkout
[46] https://www.mailgun.com/
[47] https://www.sublimetext.com
[48] https://code.visualstudio.com/
[49] https://www.python.org/downloads/
[50] https://pypi.org/project/virtualenv/
[51] https://git-scm.com/
[52] https://github.com/
[53] https://codecapsules.io

# Setting Up the Frontend

We'll use the Laurel[54] frontend template from https://cruip.com to add our functionality. This template is perfect for our project – there is already an email subscription box that just needs to be implemented and, with a few modifications, we'll implement a "Buy Now" button.

After downloading the Laurel template:

1. Create a directory named `project`.
2. Within the `project` directory, create a sub-directory named `templates`.
3. Open the downloaded template and extract the files within the `laurel` directory into the `templates` subdirectory.

You can view the template by opening the `index.html` file in the `templates` subdirectory. We'll change the first "Early access" button to "Buy Now" and implement Stripe Checkout functionality for it.

Then we'll change the second "Early access" button at the bottom of the template to "Subscribe", and implement the email subscription functionality for it.

First, let's change the "Early access" button texts.

## Modifying the "Early access" text

We'll start with changing the first "Early access" button to "Buy Now". Open the `index.html` file in a text editor.

Find this line:

```
<div class="hero-cta"><a class="button button-shadow" href="#">Learn more</a><a clas\
s="button button-primary button-shadow" href="#">Early access</a></div>
```

Replace it with:

```
<div class="hero-cta">
    <a class="button button-shadow" href="#">Learn more</a>
    <a id="checkout-button" class="button button-primary button-shadow" href="#">Buy\
 Now</a>
</div>
```

As well as changing "Early access" to "Buy Now", we've added [SOMETHING?], and given it an ID with `id="checkout-button"`. This will be useful when implementing Stripe Checkout.

Next, find this line:

[54] https://cruip.com/laurel/

```
<a class="button button-primary button-block button-shadow" href="#">Early access</a>
```

Replace “Early access” with “Subscribe”.

View the changes by saving the `index.html` file and re-opening it in a web browser. We have one more task before building our Flask backend.

## Project directory restructuring

To make a functional web application out of our template, Flask requires a specific directory structure, so we will need to reorganise the `project` directory. To do this:

1. Create a new directory named `static` in the `project` directory.
2. Navigate to the `templates` directory and then the `dist` directory.
3. Copy all of the directories located in `dist` into the `static` directory that we created above.

Your `project` file structure should look like this:

```
project
    static
      css
        + style.css
      images
        + iphone-mockup.png
      js
        + main.min.js
    templates
```

This was necessary because Flask **strictly** serves CSS, JavaScript, and images from the `static` directory, and renders HTML files in the `templates` directory. Because we’ve moved our template’s files around, we now need to edit our `index.html` file to point to their new locations.

Flask uses the Jinja[55] templating library to allow us to embed backend code in HTML. This code will be executed on the webserver before a given page is served to the user, allowing us to give that page dynamic functionality and make it responsive to user input.

The first thing we will use Jinja templating for is to dynamically locate and load our `index.html` file’s stylesheet. Open the `index.html` file in the `templates` folder and find this line:

```
<link rel="stylesheet" href="dist/css/style.css">
```

Replace the value of `href` with the string below.

[55] https://jinja.palletsprojects.com/en/2.11.x/templates/

```
<link rel="stylesheet" href="{{url_for('static',filename='css/style.css')}}">
```

In Jinja, anything between `{{` and `}}` is server-side code that will be evaluated before the page is served to users, i.e. when it is *rendered*[56]. In this way, we can include the output of Python functions and the values of Python variables in our HTML. Jinja syntax is similar to Python, but not identical.

In the Jinja code above, we're calling the function `url_for()`, which asks Flask to find the location of our `style.css` file in our `static` directory.

Speaking of Flask, we're almost ready to implement our functionality.

# Setting Up the Virtual Environment

We'll create a virtual environment[57] for our project. The virtual environment will be useful later on when we host our web application on Code Capsules, as it will ensure that the Python libraries we use for development are installed in the Capsule.

To create a virtual environment, navigate to the `project` directory in a terminal and enter `virtualenv env`.

Activate the virtual environment with:

**Linux/MacOSX**: `source env/bin/activate`

**Windows**: `\env\Scripts\activate.bat`

If the virtual environment has activated correctly, you'll notice `(env)` to the left of your name in the terminal. Keep this terminal open – we'll install the project requirements next.

## Installing the requirements

For our project, we'll use the following libraries:

- Flask[58] is a lightweight Python web development framework.
- Gunicorn[59] is the WSGI server[60] we'll use to host our application on Code Capsules.
- Requests[61] is a Python library that allows us to send HTTP requests[62].
- Stripe[63] is another Python library that will help us interact with the Stripe API[64].

Install these by entering the command below in the virtual environment.

[56] https://flask.palletsprojects.com/en/1.1.x/tutorial/templates/
[57] https://docs.python.org/3/library/venv.html
[58] https://flask.palletsprojects.com/en/1.1.x/
[59] https://gunicorn.org/
[60] https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
[61] https://pypi.org/project/requests/
[62] https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
[63] https://pypi.org/project/stripe/
[64] https://stripe.com/docs/api

```
pip3 install flask gunicorn requests stripe
```

Now we can build the backend for our web application.

# Creating the Flask Application

In the `projects` folder, create a new file named `app.py`. This file will contain all of our Flask code.

Open the `app.py` file in a text editor and enter the following:

```
from flask import Flask, render_template, request
import requests, stripe

app = Flask(__name__)

@app.route("/",methods=["GET","POST"])
def index():
  return render_template("index.html")

if __name__== '__main__':
  app.run(debug=True)
```

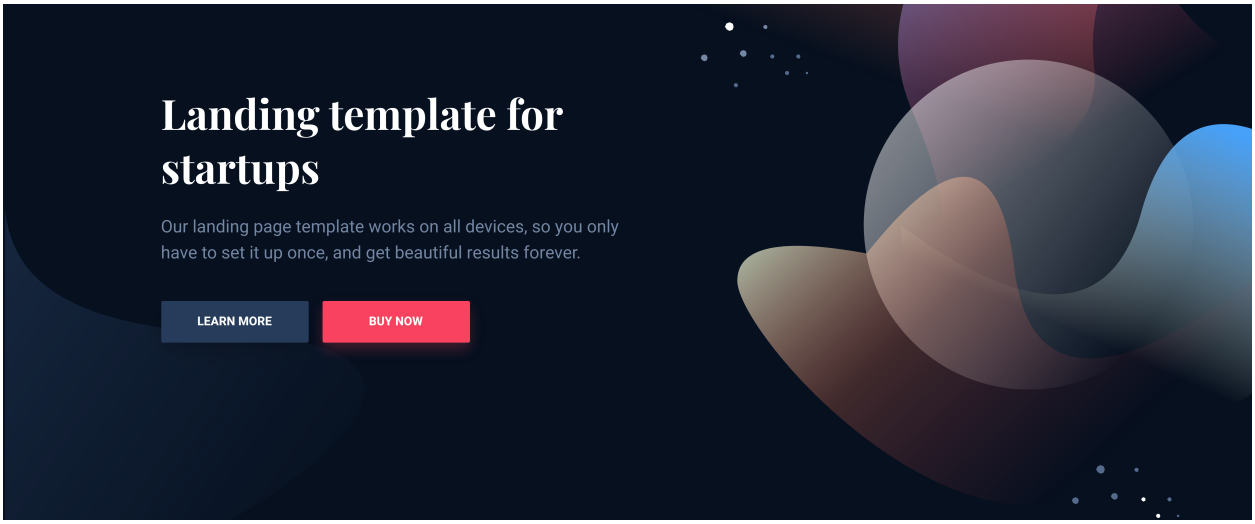
We import the following functions from `flask`:

- `Flask`, which provides the Flask application object.
- `render_template()`, which will render our `index.html` file.
- `request`, an object which contains any information sent to our web application – later this will be used to retrieve the email address entered in our subscription box. Be careful not to confuse this with the `requests` library.

The `index` function has a route decorator[65] which causes it to execute when Flask receives an HTTP `GET`[66] or `POST`[67] request for the "/" URL, i.e. when someone navigates to the website's domain or IP address in a browser.

When `render_template("index.html")` runs, Flask will look in the `templates` directory for a file named `index.html` and render it by executing its Jinja template code and serving the resulting HTML. To see this, run `app.py` with `flask run` in your terminal. Open the provided IP address in your browser – the web application should look like this:

[65] https://flask.palletsprojects.com/en/1.1.x/api/#flask.Flask.route
[66] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET
[67] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST




**website initialise**

Let's make this web application useful and implement the first bit of functionality – the email list feature.

## Signing up for Mailgun

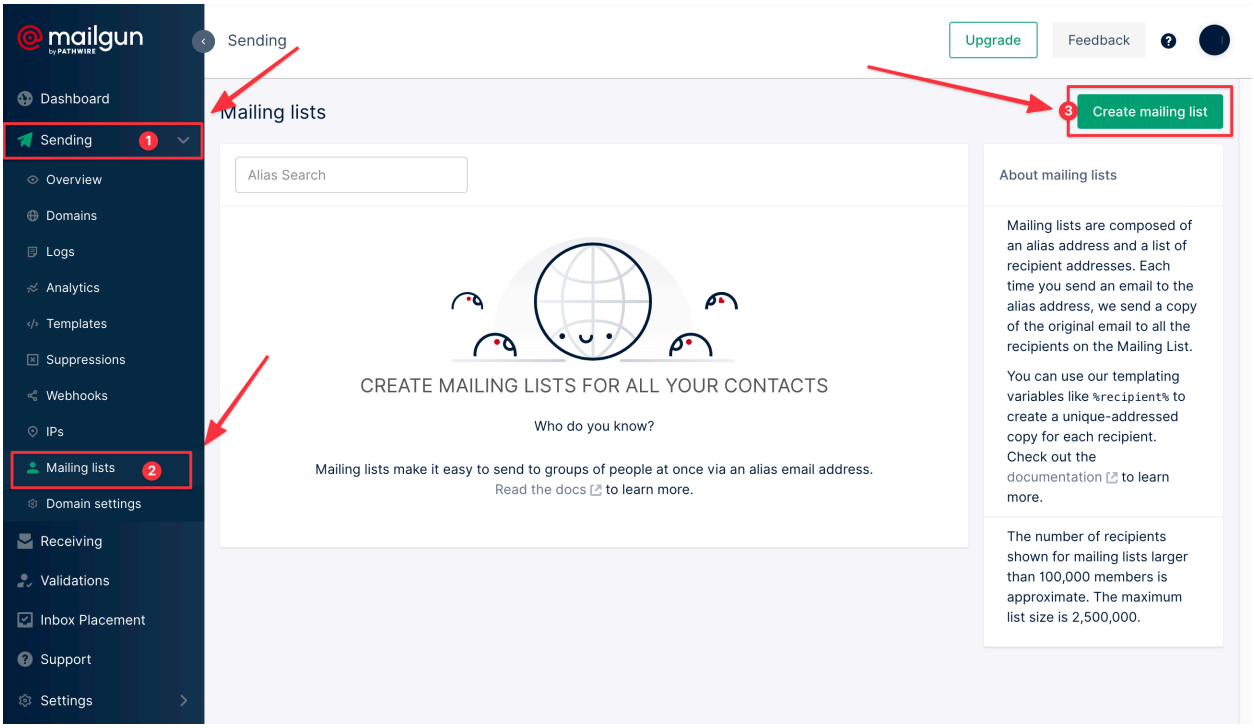
We'll use Mailgun[68] to handle our email subscriber list. Mailgun is free up to 5,000 emails per month. Register with Mailgun[69] and continue.

With an account registered, create a mailing list by doing the following:

1. Log in to Mailgun.
2. Click "Sending" then "Mailing lists" on the dashboard.

   Mailing list

[68] https://www.mailgun.com/
[69] https://signup.mailgun.com/new/signup




**Mailing list**

3. At the top right, click "Create mailing list".
4. Enter whatever you'd like for the address prefix, name, and description – leave everything else default.
5. Click "Add mailing list".

Navigate to the mailing list we just created. You'll see something called an alias address – Mailgun provides every new mailing list with one. When you send an email to your alias address, Mailgun sends a copy of the email to everyone who is subscribed to your mailing list. Jot down your alias address, we'll use it soon.

Next, you'll need to retrieve the API key[70] for your Mailgun account. We'll use this API key in our web application to validate your Mailgun account when people subscribe to your mailing list.

Find the API key by clicking on your account at the top right of the screen. Click "API keys", and make a note of your **private** API key.

# Implementing the Subscribe Button

With the mailing list created, we can implement the subscribe button.

Re-open the `index.html` file. At the bottom of the file, find the line:

[70] https://cloud.google.com/endpoints/docs/openapi/when-why-api-key

```
 <section class="newsletter section">
```

From the above line down to its corresponding `</section>` tag, replace all of the markup with the following:

```
<section class="newsletter section">
  <div class="container-sm">
    <div class="newsletter-inner section-inner">
      <div class="newsletter-header text-center">
        <h2 class="section-title mt-0">Stay in the know</h2>
        <p class="section-paragraph">Lorem ipsum is common placeholder text used to \
demonstrate the graphic elements of a document or visual presentation.</p>
      </div>
      <form method="POST">
        <div class="footer-form newsletter-form field field-grouped">
          <div class="control control-expanded">
            <input class="input" type="email" name="email" placeholder="Your best em\
ail&hellip;">
          </div>
          <div class="control">
            <button class="button button-primary button-block button-shadow" type="s\
ubmit">Subscribe</a>
            </div>
          </div>
        </form>
      </div>
    </div>
</section>
```

The important part of this HTML for our functionality is the `form` tag. Let's take a closer look at it.

```
      <form method="POST">
        <div class="footer-form newsletter-form field field-grouped">
          <div class="control control-expanded">
            <input class="input" type="email" name="email" placeholder="Your best em\
ail&hellip;">
          </div>
          <div class="control">
            <button class="button button-primary button-block button-shadow" type="s\
ubmit">Subscribe</a>
            </div>
          </div>
        </form>
```

This contains one `input`, for the user's email address, and a `button` for submitting that email address. When the user clicks on the button, an HTTP request will be sent from their browser to our Flask backend, containing the email address. As per the `method` attribute of the `form` tag, this will be a [POST request.

Recall that the Python code we entered in the last section provided for both `GET` and `POST` HTTP requests. As submitting this form will also send a request to "/", we can differentiate between a user browsing to our website (`GET`) and subscribing to our mailing list (`POST`) by looking at the HTTP method. We'll do that in the next section.

## Subscribe functionality in Flask

Return to the `app.py` file. Find this line:

```
@app.route("/",methods["GET","POST"])
```

Just above it, enter this code:

```
def subscribe(user_email, email_list, api_key):
  return requests.post(
        "https://api.mailgun.net/v3/lists/"+email_list+"/members",
        auth=('api', api_key),
        data={'subscribed': True,
              'address': user_email,})
```

This function is called when a user clicks the "Subscribe" button. It takes three arguments:

- `user_email`: The email the user has entered.
- `email_list`: Your Mailgun alias address.
- `api_key`: Your Mailgun secret API key.

The real logic is contained in the `return` line. Here, we use `requests.post()` to add the `user_email` to our `email_list`, by sending (or "posting") all of the values in `data` to Mailgun's email list API[71].

Next, modify the current `index()` function like below, replacing `MAILGUN_ALIAS` and `YOUR-MAILGUN-PRIVATE-KEY` with the email alias and private API key we previously retrieved:

[71] https://documentation.mailgun.com/en/latest/api_reference.html

```
@app.route("/",methods["GET","POST"])
def index():
  if request.method == "POST":
    user_email = request.form.get('email')
    response = subscribe(user_email,
      'MAILGUN_ALIAS',
      'YOUR-MAILGUN-PRIVATE-KEY')

  return render_template("index.html")
```

In the previous section, we added the `POST` method to the subscribe button. We will therefore know when someone has clicked the subscribe button with the line:

```
if request.method == "POST":
```

If they've clicked the subscribe button, we obtain the email they entered by referencing the relevant `input` field's `name` attribute in `request.form.get`, and adding the email to the mailing list our `subscribe` method.

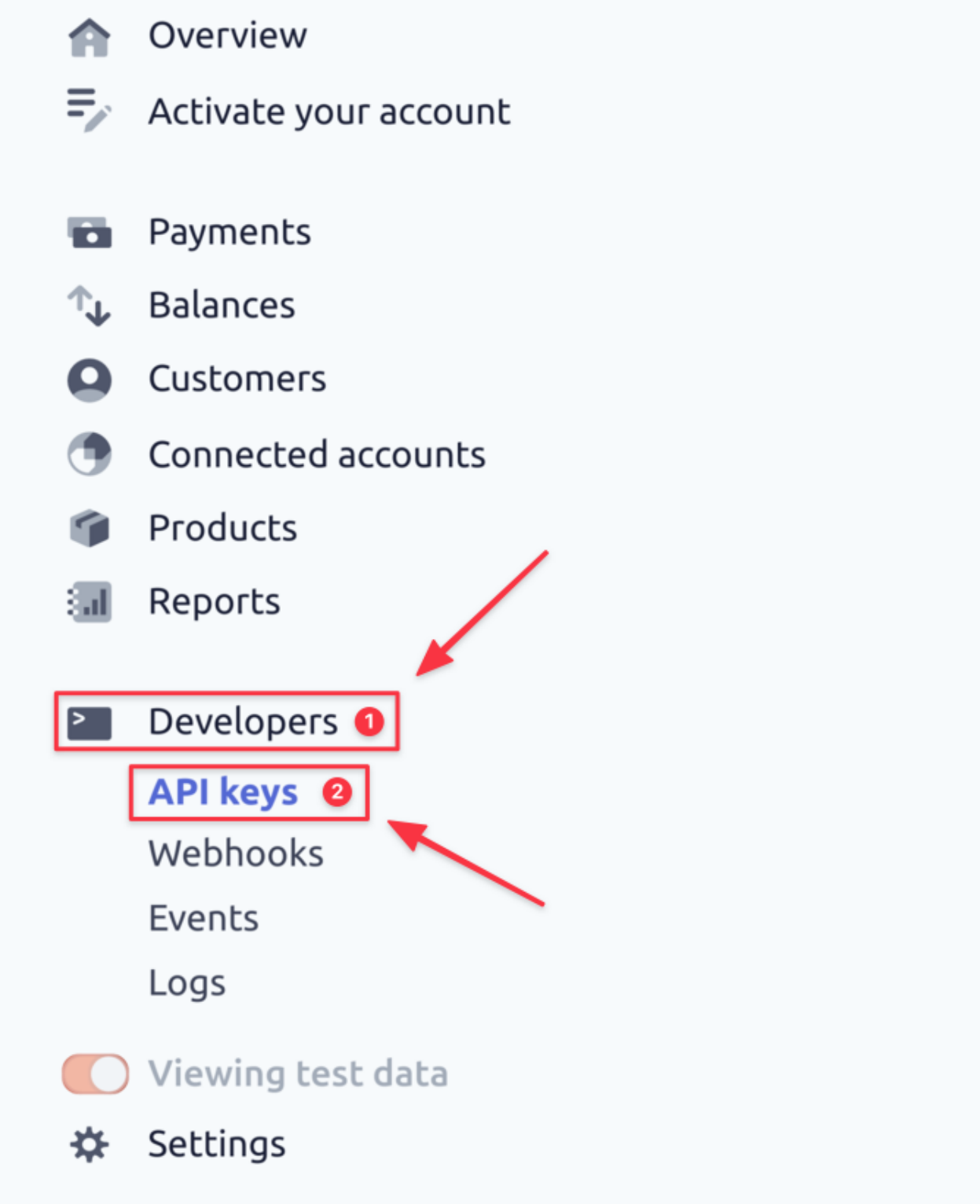
Try it out: enter an email address and hit "Subscribe". Navigate back to the "Mailing lists" tab on Mailgun and click on the list we created. You will find the email address you just submitted under "Recipients".

All that's left is to add functionality to our "Buy Now" button.

## Implementing "Buy Now" with Stripe Checkout

Stripe Checkout allows business owners to accept payments on their web applications. Let's create an account[72]. After creating an account, log in and find your API keys by clicking "Developers" then "API keys" on the dashboard.

[72] https://dashboard.stripe.com/register




**stripe-dashboard**

Here we'll see two API keys – a *publishable* API key, and a *secret* API key. You can think of these

as a username and password. Stripe uses the publishable API key to identify your account, and the secret API key to ensure it's really you using it.

Open the `app.py` file. Above the subscribe function, add the following lines, replacing `YOUR PUBLISHABLE KEY HERE` and `YOUR SECRET KEY HERE` appropriately:

```
app.config['STRIPE_PUBLISH_KEY'] = 'YOUR PUBLISHABLE KEY HERE'
app.config['STRIPE_SECRET_KEY'] = 'YOUR SECRET KEY HERE'
stripe.api_key = app.config['STRIPE_SECRET_KEY']
```

In third code, we place the two Stripe keys in our Flask app's configuration settings for ease of access, then set our Stripe secret API key.

These are test API keys that we'll use to check out our product. With these keys, no charges will be incurred when making payments. However, before we can make a payment, we need a product, so let's create one. Return to Stripe, log in, and navigate to the "Products" tab on the dashboard.

## Creating a product

Create a product by doing the following:

1. Click "Add product" on the top right.
2. Name the product.
3. Add a description and price.
4. Choose "One time" payment.
5. Click "Save product"

After the last step, save the API key found in the "Pricing" section. We'll use this API key to tell Stripe which product we want our customers to pay for.

## Adding functionality in Flask

Time to create the "Buy Now" button logic. Open `app.py` again and modify the index function accordingly. Replace "YOUR-PRICE-API-KEY" with the API key for your product, that we saved in the previous section.

```
@app.route("/",methods=["GET","POST"])
def index():
  session = stripe.checkout.Session.create(
    payment_method_types=['card'],
    mode = 'payment',
    success_url = 'https://example.com/success',
    cancel_url = 'https://example.com/cancel',
    line_items=[{'price':'YOUR-PRICE-API-KEY',
          'quantity':1,
    }]
    )

  if request.method == "POST":
    user_email = request.form.get('email')

    response = subscribe(user_email,
      'MAILGUN_ALIAS',
      'YOUR-MAILGUN-PRIVATE-KEY')

  return render_template("index.html",
    checkout_id=session['id'],
    checkout_pk=app.config['STRIPE_PK'],
    )
```

We use the `stripe` library to create a new "Session" object. This object contains multiple variables affecting how our customers interact with the "Buy Now" button. For more information on these variables, see Stripe's documentation[73].

We also return two new variables – `checkout_id` and `checkout_pk`. `checkout_id`, which we get from Stripe, stores information about the potential purchase (price, payment type, etc). `checkout_pk` stores our private API key, which we added to the Flask app's configuration settings above. When a customer buys our product, their money is sent to the account associated with this private API key.

Let's see how our HTML will use these new variables and redirect us to the Stripe Checkout page.

## "Buy Now" button functionality in the HTML file

With our Flask logic finished, we can implement the "Buy Now" button functionality in our `index.html` file. Open the `index.html` and find this section:

[73] https://stripe.com/docs/api/checkout/sessions/object

```
<div class="hero-copy">
  <h1 class="hero-title mt-0">Landing template for startups</h1>
  <p class="hero-paragraph">Our landing page template works on all devices, so you o\
nly have to set it up once, and get beautiful results forever.</p>
  <div class="hero-cta"><a class="button button-shadow" href="#">Learn more</a><a id\
='checkout-button' class="button button-primary button-shadow" href="#">Buy Now</a><\
/div>
</div>
```

Directly below the `</div>` line, add:

```
<script src="https://js.stripe.com/v3/"></script>
<script>
  const checkout_pk= '{{checkout_pk}}';
  const checkout_id = '{{checkout_id}}';
  var stripe = Stripe(checkout_pk)
  const button = document.querySelector('#checkout-button')

  button.addEventListener('click', event =>{
    stripe.redirectToCheckout({
        sessionId: checkout_id
      }).then(function(result){

        });
    })
</script>
```

This code adds a JavaScript event which will trigger when the customer clicks the "Buy Now" button, and will redirect them to the Stripe Checkout page. The Stripe Checkout page changes according to the information stored in `checkout_id`.

Also, take a look at the Jinja code in these lines:

```
const checkout_pk= '{{checkout_pk}}';
const checkout_id = '{{checkout_id}}';
```

When the `index.html` template is rendered before being served to the user, Flask will substitute in the current values of the Python variables that we passed to `render_template()`. While `checkout_pk` will remain the same throughout, `checkout_id` will be unique for each purchase.

The "Buy Now" button is good to go – run `app.py` again. Try making a payment using the test credit card number[74] `4242 4242 4242 4242` with any name, expiration date, and security code. No charges will be incurred.

[74] https://stripe.com/docs/testing

# Hosting the Application on Code Capsules

Now that we've added all the functionality, we need to create some files that Code Capsules will use when hosting our application. We'll also take a look at a security problem in our current application, and how to fix it before we go live.

## Creating the "requirements.txt" file and procfile

To host this application on Code Capsules, we need to create a `requirements.txt` file and a `Procfile`.

1. In the `project` directory, ensure the virtual environment is activated and then enter `pip3 freeze > requirements.txt`.
2. Create another file named `Procfile`, containing the line `web: gunicorn app:app`.

With the `requirements.txt` file, Code Capsules will now know what libraries to install to run the application. The `Procfile` tells Code Capsules to use the `gunicorn` WSGI server to serve HTML rendered by Flask to end-users.

Now we can fix that security problem mentioned before, and host the application.

## Removing secret keys

We need to send our application to GitHub so Code Capsules can host it. But currently this project's code contains all of our API keys. It is considered **very** poor practice to send personal API keys to GitHub, especially public repositories. Anyone could find them and incur charges on your debit card. Luckily, there is a workaround.

By working with environment variables[75], we can use our API keys on Code Capsules without exposing them on GitHub. We will alter our application to retrieve our API keys from environment variables, which we will set on Code Capsules.

To do this, change the code at the top of `app.py` as follows:

[75] https://opensource.com/article/19/8/what-are-environment-variables

```
from flask import Flask, render_template, request
import requests, stripe, os


app = Flask(__name__)
app.config['STRIPE_PK'] = os.getenv("STRIPE_PK")
app.config['STRIPE_SK'] = os.getenv("STRIPE_SK")
stripe.api_key = app.config['STRIPE_SK']
```

Notice that we've imported a new Python module, os[76], which allows us to retrieve with environment variables using the `os.getenv()` method.

We've now done this with our Stripe publishable and secret API keys and our Mailgun secret key. On Code Capsules, we'll set environment variables named `'STRIPE_PK'`, `'STRIPE_SK'`, and `'MAILGUN_SK'`. This way, our API keys do not have to be stored on GitHub and will remain secret. Notice, we aren't adding environment variables for the Stripe product API key. This doesn't contain any sensitive information. It just displays a product's price.

Your final `app.py` code should look like this:

```
from flask import Flask, render_template, request
import requests, stripe, os

app = Flask(__name__)
app.config['STRIPE_PK'] = os.getenv("STRIPE_PK")
app.config['STRIPE_SK'] = os.getenv("STRIPE_SK")
stripe.api_key = app.config['STRIPE_SK']


def subscribe(user_email, email_list, api_key):
  return requests.post(
        "https://api.mailgun.net/v3/lists/"+email_list+"/members",
        auth=('api', api_key),
        data={'subscribed': True,
              'address': user_email,})

@app.route("/",methods=["GET","POST"])
def index():

  session = stripe.checkout.Session.create(
    payment_method_types=['card'],
    mode = 'payment',
    success_url = 'https://example.com/success',
```

[76] https://docs.python.org/3/library/os.html

```
    cancel_url = 'https://example.com/cancel',
    line_items=[{'price':'YOUR-PRICE-API-KEY',
          'quantity':1,
    }]
    )

  if request.method == "POST":
    user_email = request.form.get('email')

    response = subscribe(user_email,
      'MAILGUN_ALIAS',
      os.getenv("MAILGUN_SK"))

  return render_template("index.html",
    checkout_id=session['id'],
    checkout_pk=app.config['STRIPE_PK'],
    )

if __name__== '__main__':
  app.run(debug=True)
```

We can now safely host our application.

## Pushing to GitHub and hosting the application on Code Capsules

Before we create the Capsule that will host our code, take the following steps:

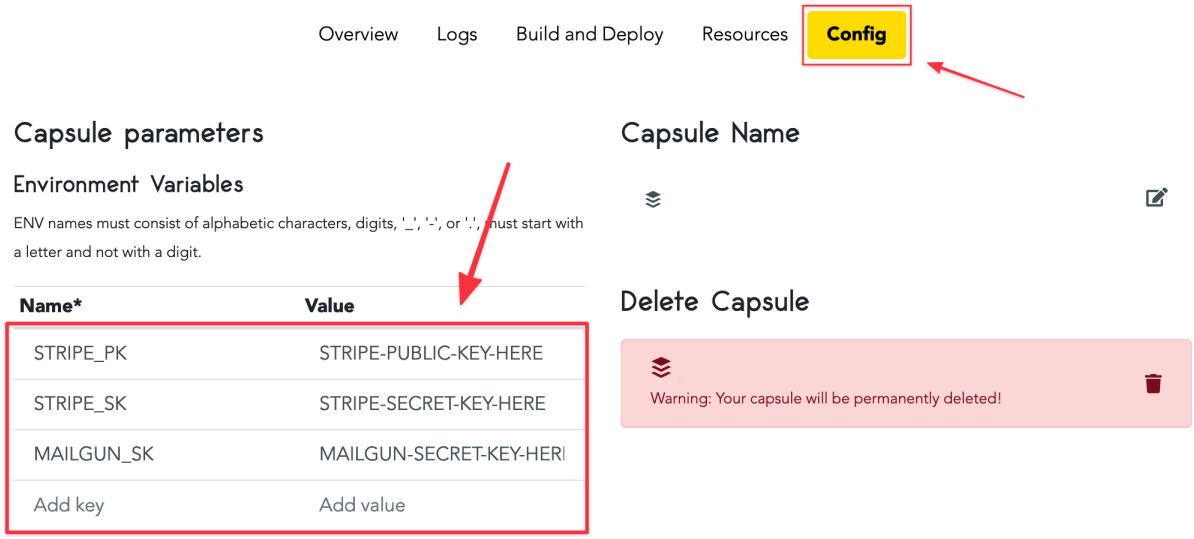
1. Create a GitHub repository for the application.
2. Send all code within the `project` directory to the repository on GitHub.
3. Log in to Code Capsules[77].
4. Grant Code Capsules access to the repository.
5. Create a Team and Space as necessary.

Now let's create the Capsule:

1. Click "Create A New Capsule".
2. Select your repository.
3. Choose the Backend Capsule type.
4. Create the Capsule.

[77] https://codecapsules.io

All that's left is to set the environment variables. Navigate to the Capsule and click on the "Config" tab. Use the image below as a guide to properly add your environment variables. Replace each value with the appropriate API key.




**environment-variables**

After entering the API keys, **make sure to click "Update"**.

All done – now anyone can view the web application and interact with the "Buy Now" and "Subscribe" buttons.

# Further Reading

We covered a lot in this tutorial: how to use Flask to implement functionality for frontend code, how to set up an email subscriber list, and how to work with Stripe.

Earlier I mentioned more information on the `url_for()` function. Check out Flask's documentation[78] for more information.

For further information on how Jinja templates work and what can be done with them, check out this link to learn more about Flask templating and Jinja2[79].

Now that you have a functional email subscriber list, you may be interested in sending emails to your list[80].

[78] https://flask.palletsprojects.com/en/1.1.x/api/#flask.url_for
[79] https://realpython.com/primer-on-jinja-templating/
[80] https://documentation.mailgun.com/en/latest/user_manual.html?highlight=template%20variables#sending-messages