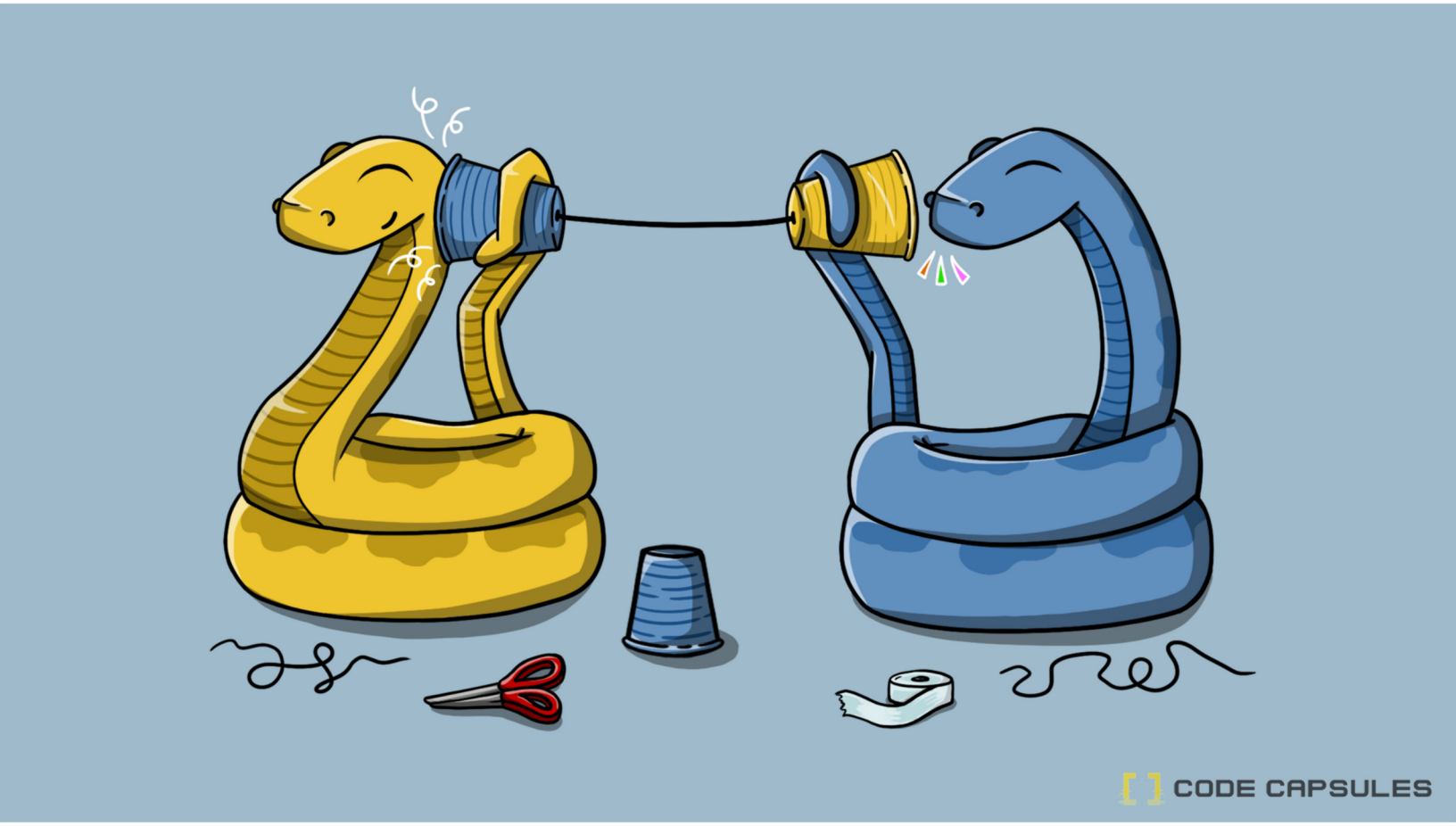
# Learn to Deploy Code Capsules




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".

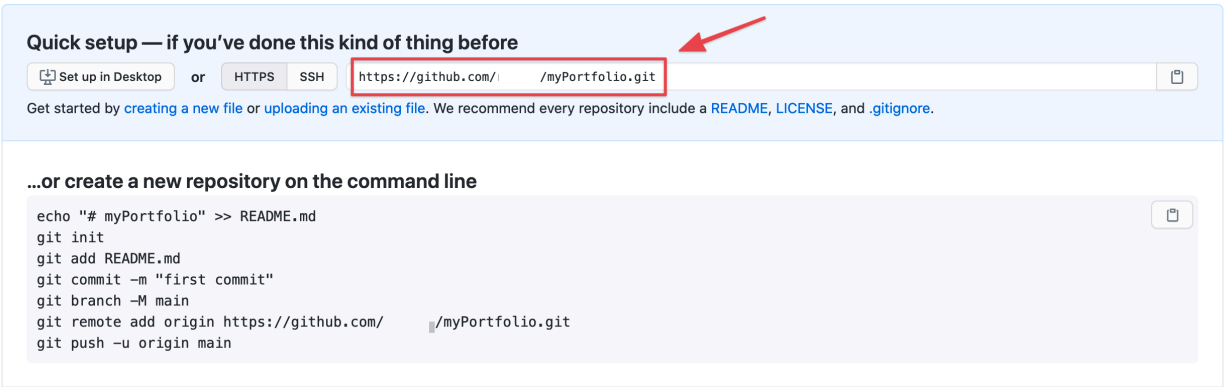



**image10**

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"):

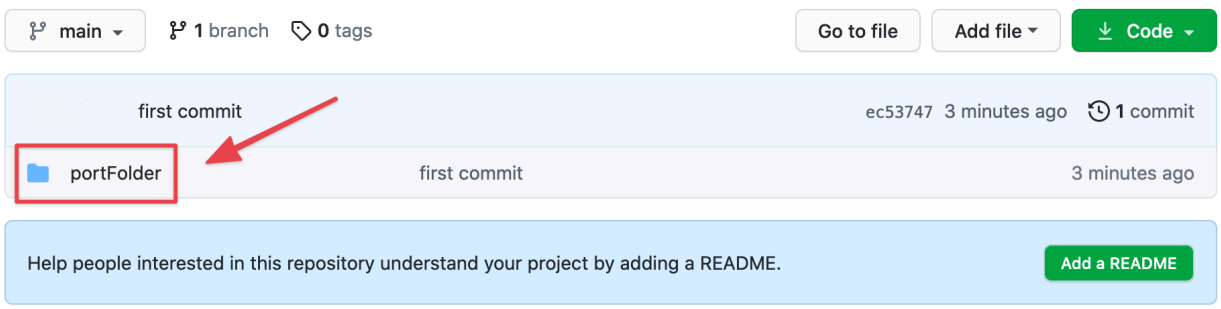


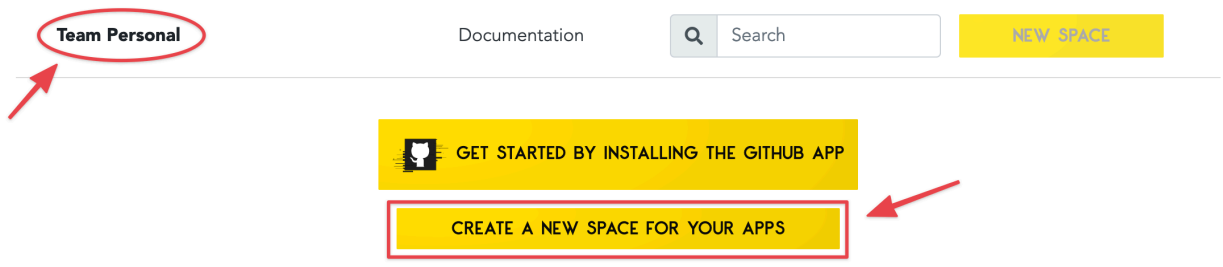

**image11**

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.



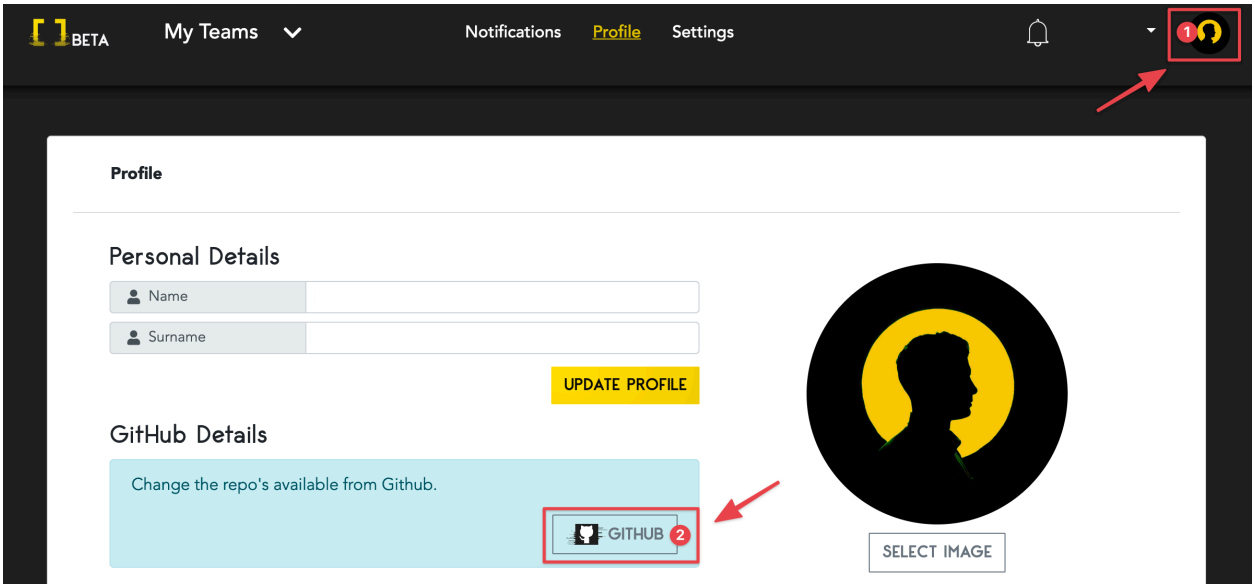

**team**

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.

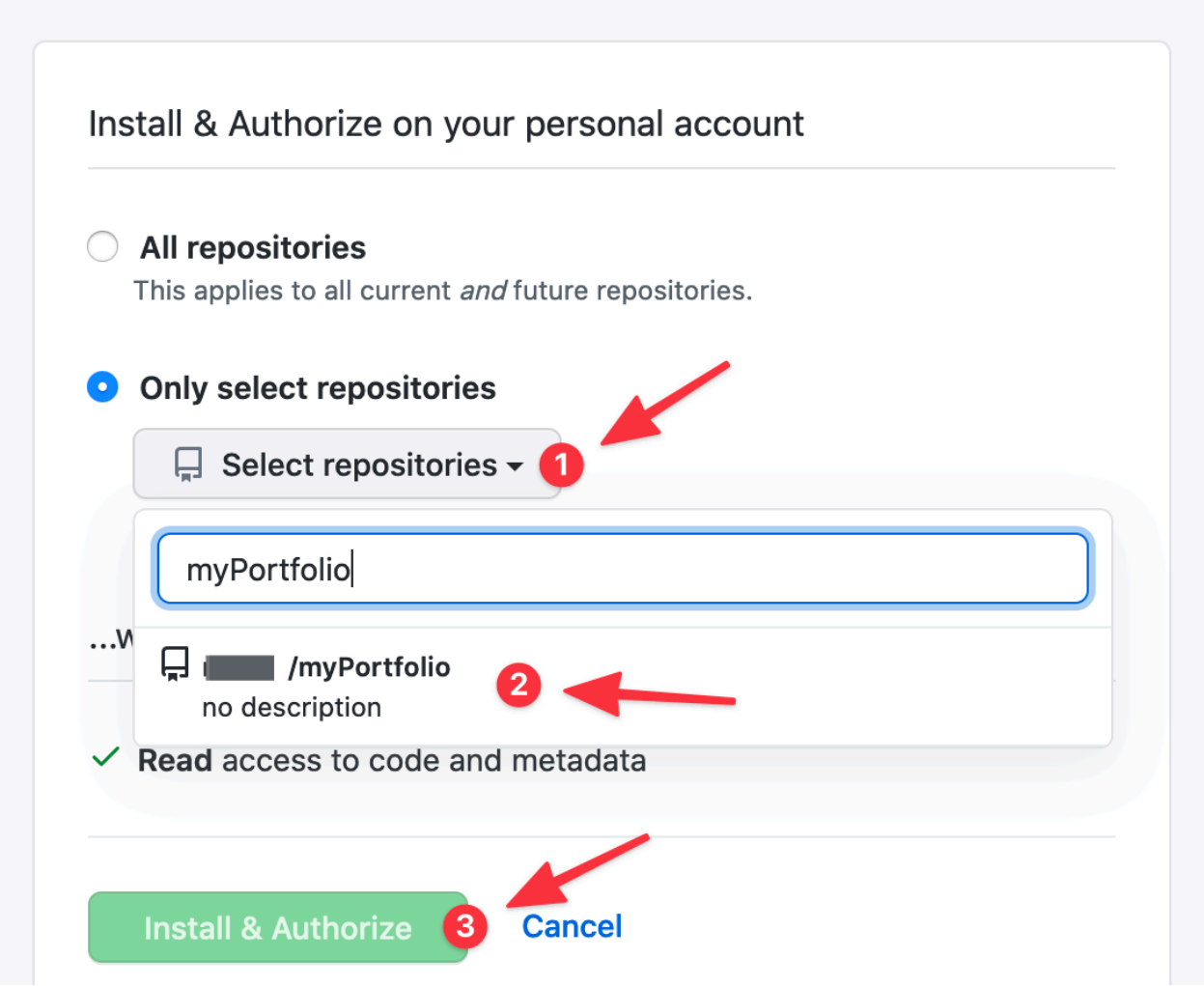



**GitHubButton**

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

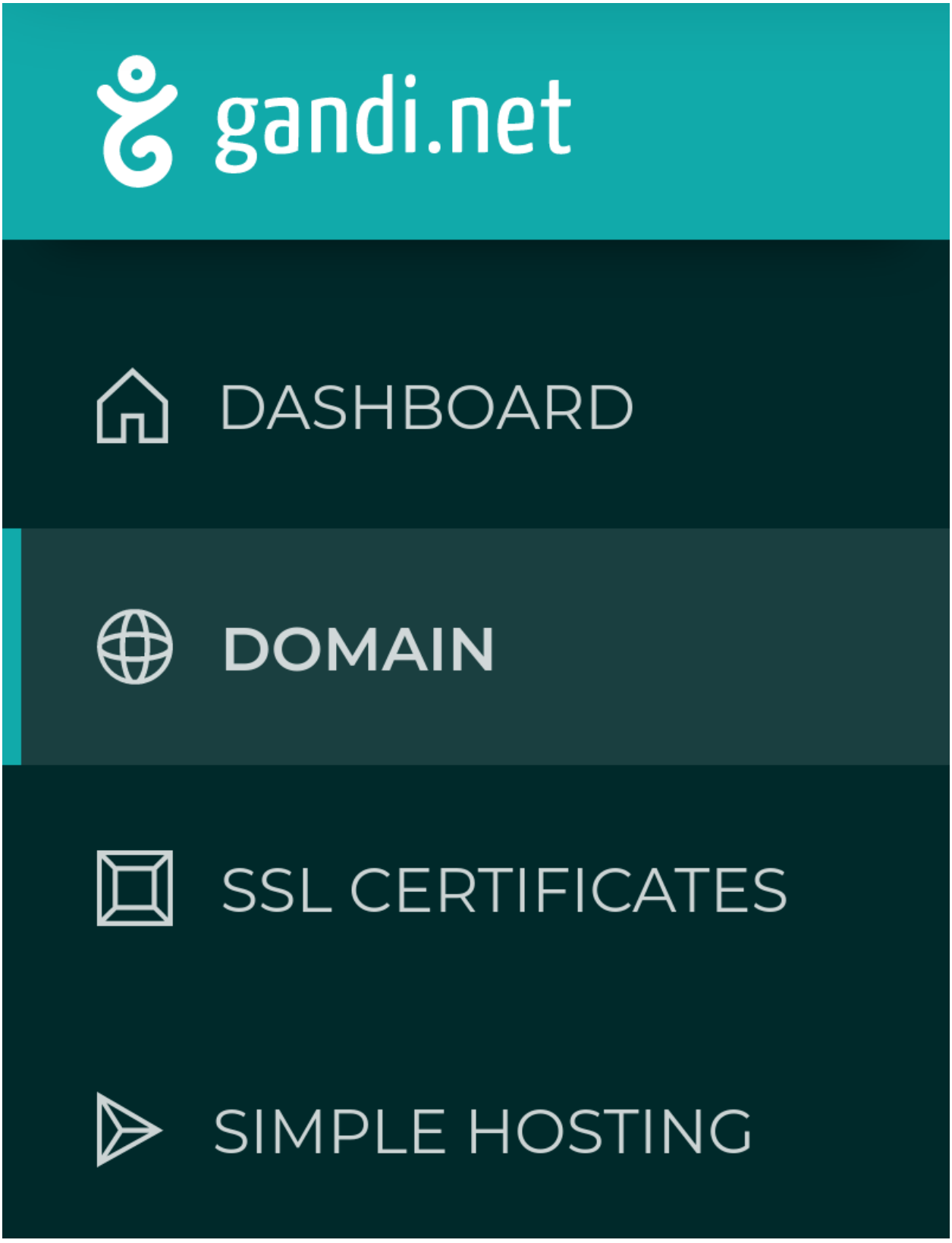

gandi.net
DASHBOARD
DOMAIN
SSL CERTIFICATES
SIMPLE HOSTING

**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**

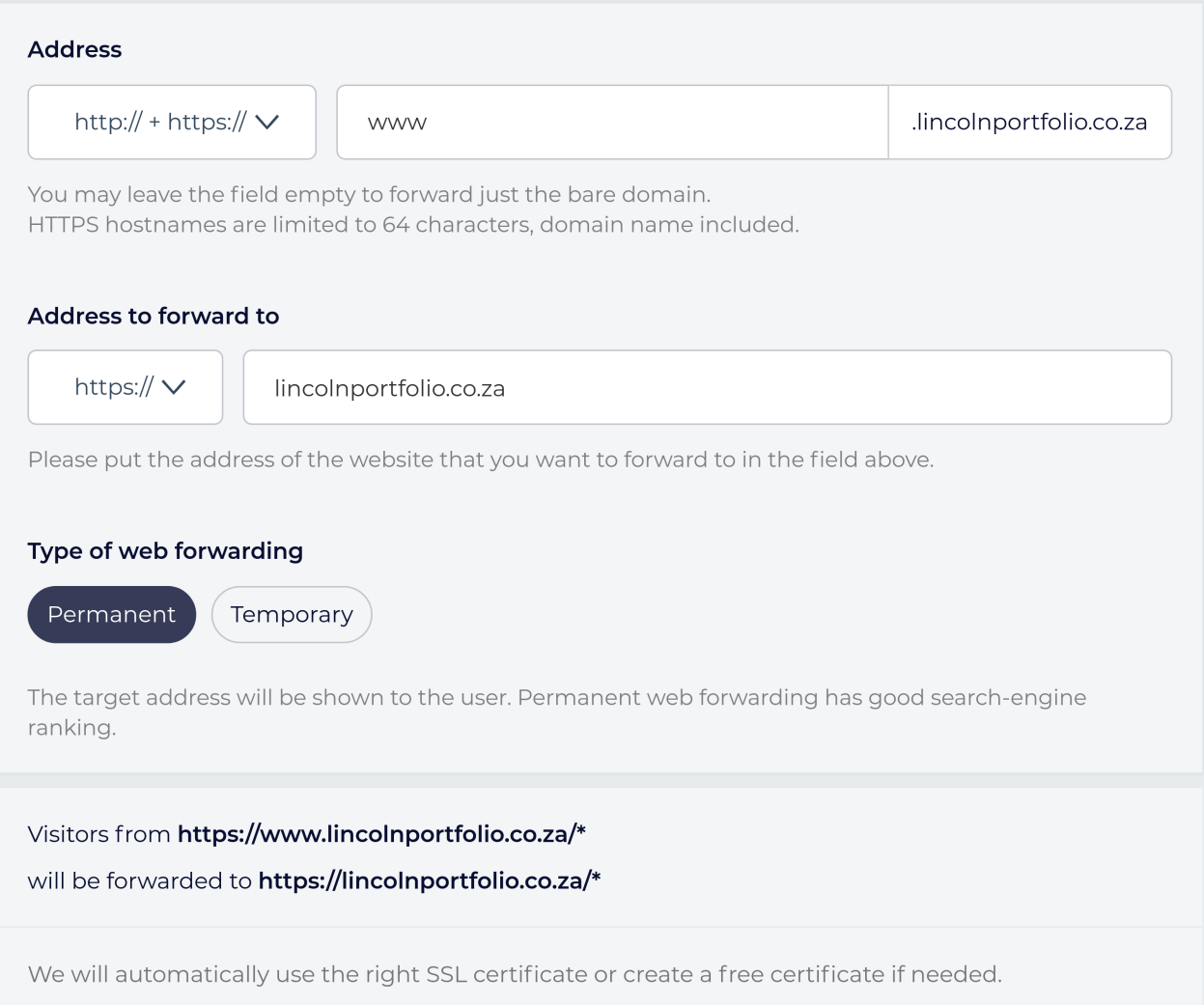



**image4**

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:

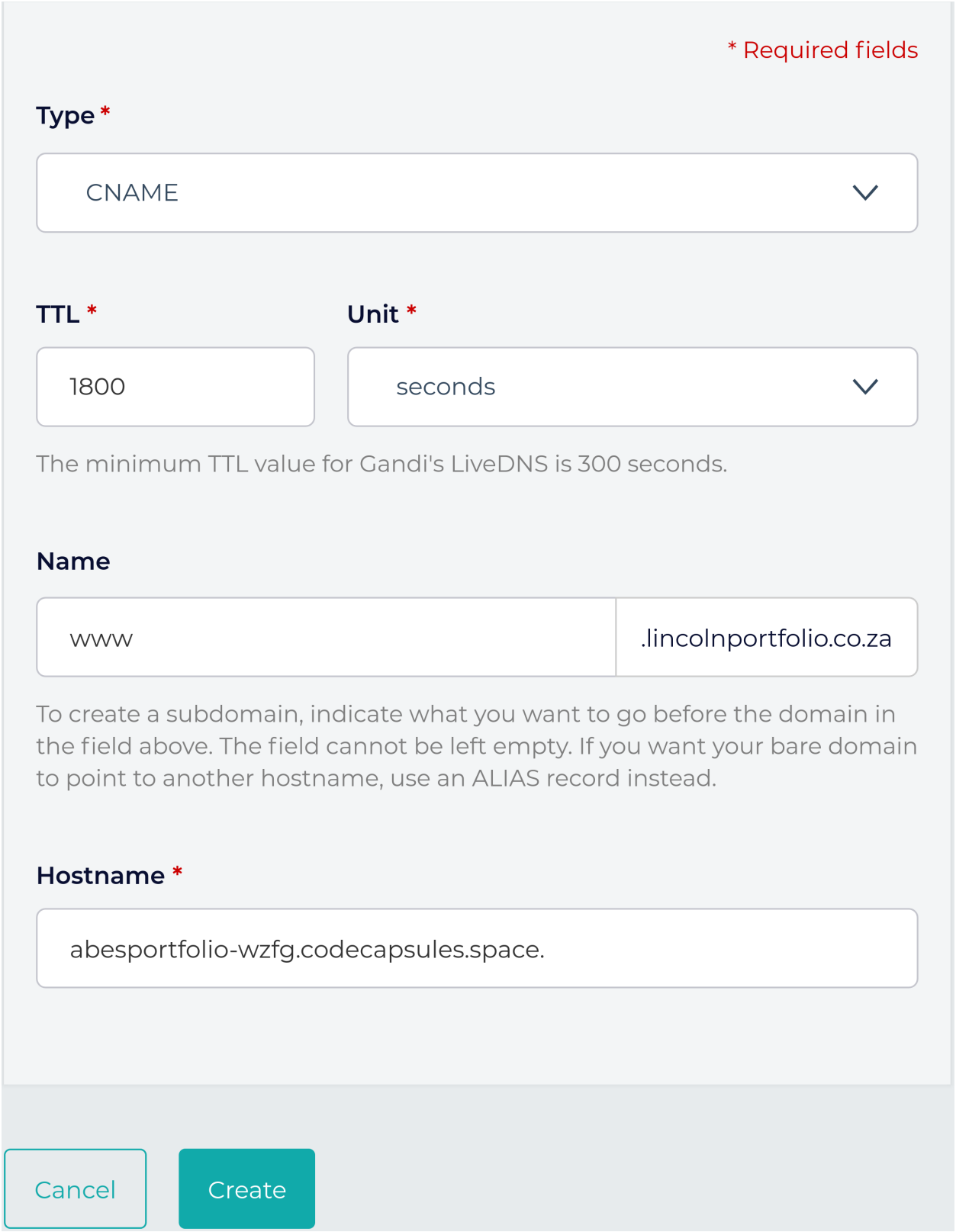

* Required fields
Type *
CNAME
TTL *
1800
Unit *
seconds
The minimum TTL value for Gandi's LiveDNS is 300 seconds.
Name
www
.lincolnportfolio.co.za
To create a subdomain, indicate what you want to go before the domain in the field above. The field cannot be left empty. If you want your bare domain to 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:

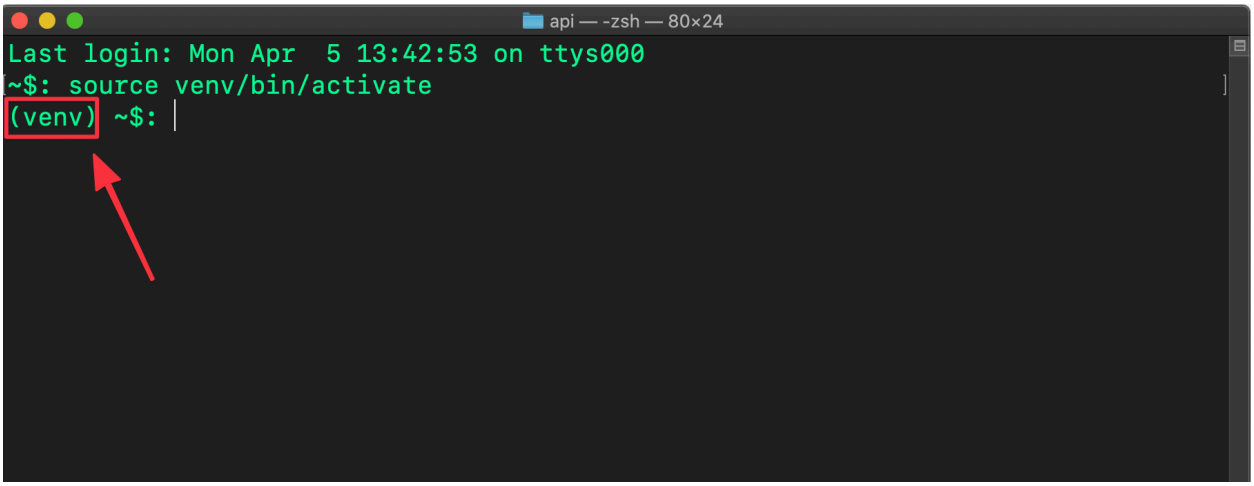
**Linux/MacOSX**

```
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.

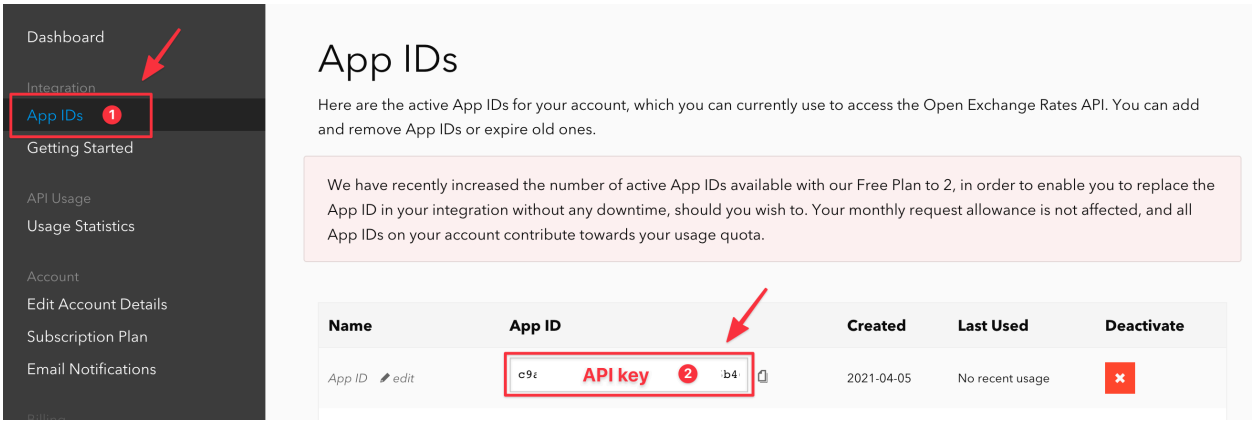
Registering an account is required so that we can receive a unique API key. An API key is a password that lets us use a particular API. In APIs with more sensitive data, these are used to prevent unauthorised access, but for open APIs like WeatherStack and OpenExchangeRates, they're used for rate limiting[37]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

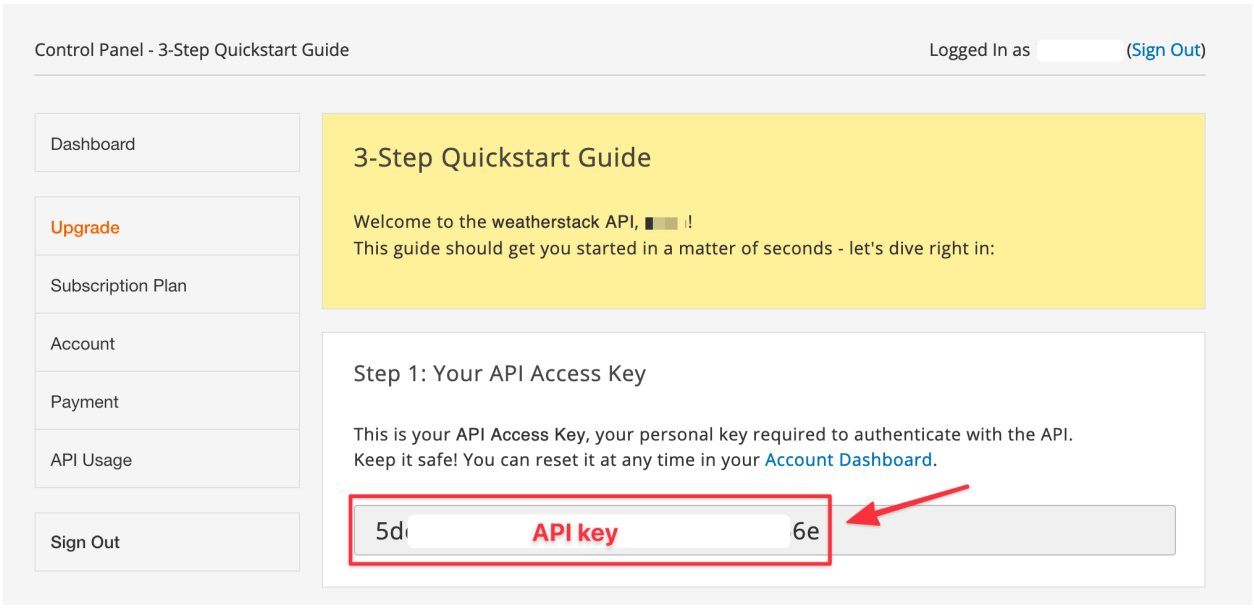
[31] https://palletsprojects.com/p/flask/
[32] https://gunicorn.org/
[33] https://medium.com/analytics-vidhya/what-is-wsgi-web-server-gateway-interface-ed2d290449e
[34] https://pypi.org/project/requests/
[35] https://weatherstack.com
[36] https://openexchangerates.org/
[37] https://en.wikipedia.org/wiki/Rate_limiting




**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)
```

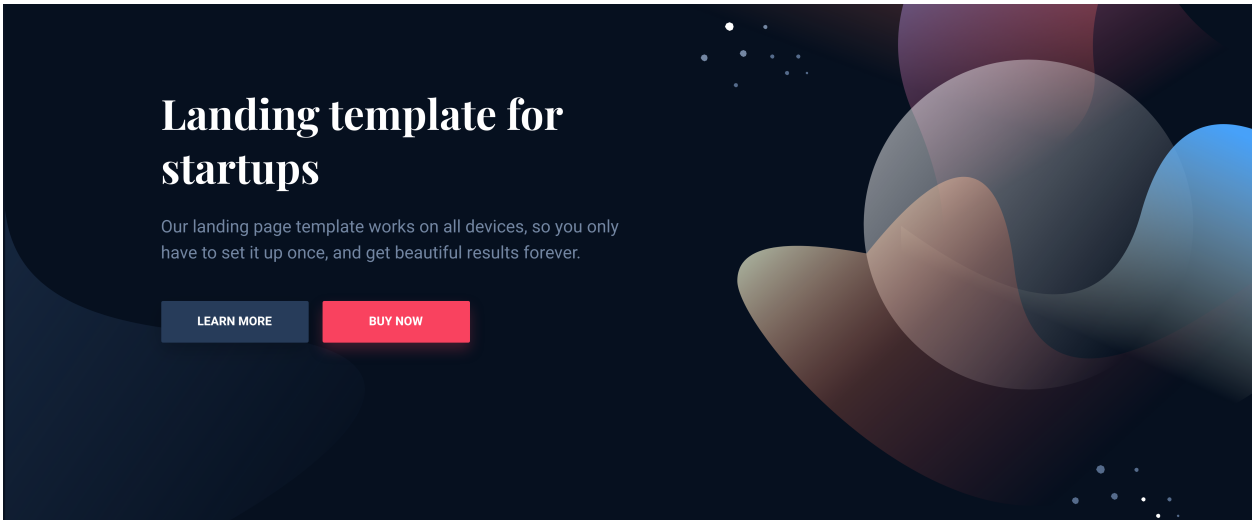
We import the following functions from `flask`:

- `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

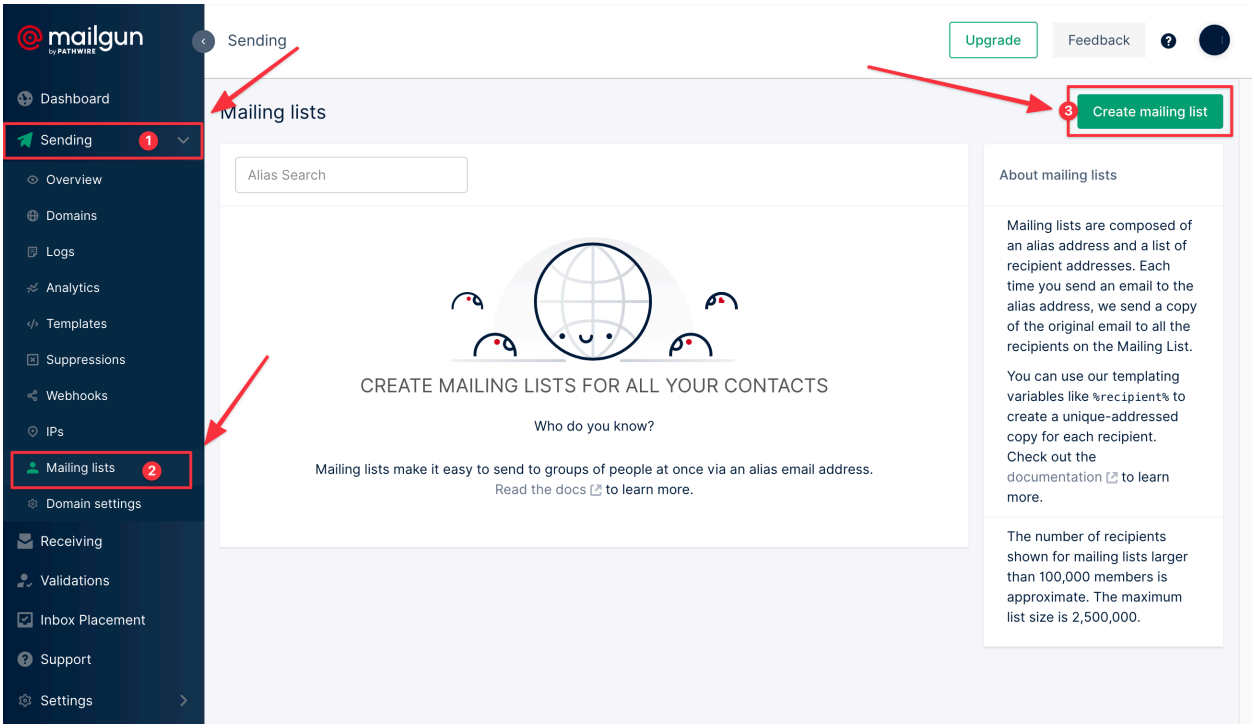
We'll use Mailgun[68] to handle our email subscriber list. Mailgun is free up to 5,000 emails per month. Register with Mailgun[69] and continue.

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.

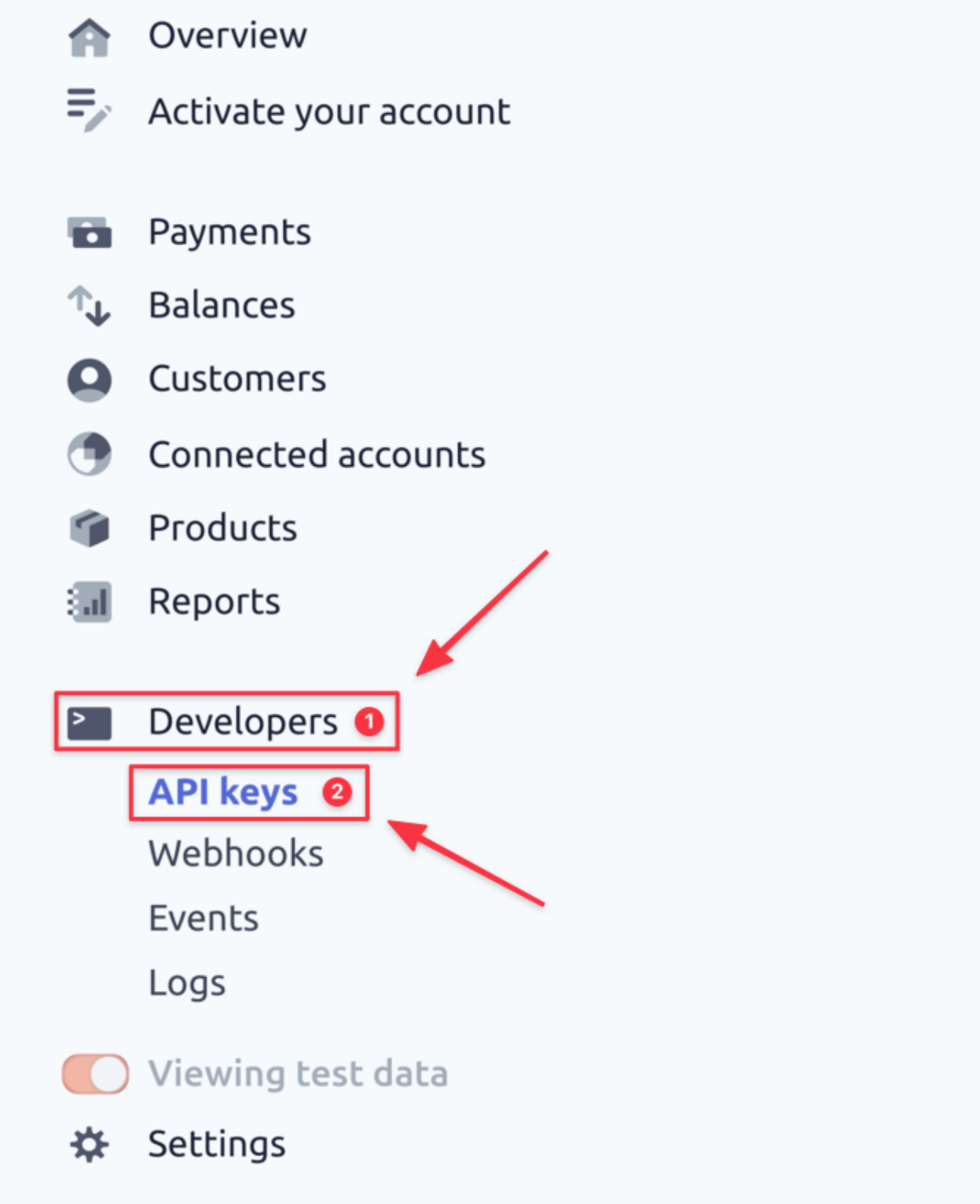
Try it out: enter an email address and hit "Subscribe". Navigate back to the "Mailing lists" tab on Mailgun and click on the list we created. You will find the email address you just submitted under "Recipients".

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:

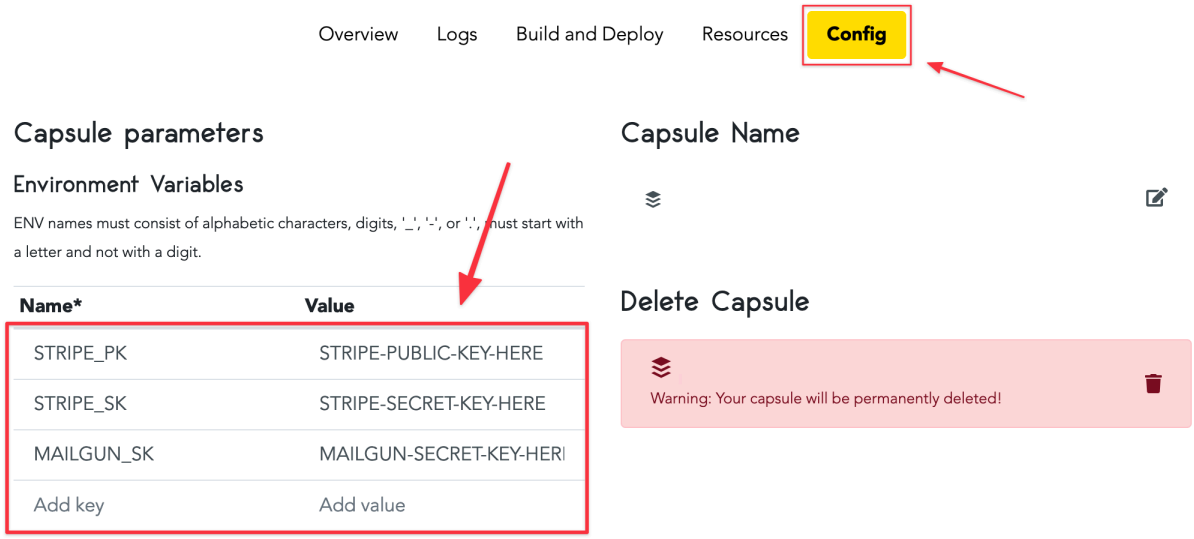
1. Create a GitHub repository for the application.
2. Send all code within the `project` directory to the repository on GitHub.
3. Log in to Code Capsules[77].
4. Grant Code Capsules access to the repository.
5. Create a Team and Space as necessary.

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

# How to Create and Host a Telegram Bot on Code Capsules

In a previous tutorial[81], we created and hosted an API on Code Capsules. In this tutorial, we'll create a client for this API in the form of a Telegram bot. This will allow us to pull temperature, weather and exchange rate data on the go by messaging our bot in the Telegram app.

We'll also learn how to host this bot on Code Capsules[82] so it can be used by others. Along the way, we'll learn some key concepts about hosting bots securely and efficiently.

Let's get started!

## Requirements

To create a Telegram bot, we'll need:

- Python[83] 3.6+ installed.
- A GitHub account[84] and Git[85] installed.
- Virtualenv[86] installed.
- A Telegram[87] account.
- A Code Capsules[88] account.
- An API on Code Capsules, created using the Personal API tutorial[89].

## About Telegram Bots

Telegram bots appear as contacts on the Telegram interface. Users interact with Telegram bots by messaging them with commands – these are words preceded by a forward slash, e.g. `/weather`, or `/currency`. Commands sent to the bot's account on Telegram will be passed to the bot's backend code (in our case, this will be the code we host on Code Capsules).

For example, when we send the command `/weather` to our bot later in this article, the bot will reply with the weather data from our personal API.

Let's create a Telegram bot.

[81] https://codecapsules.io/docs/creating-and-hosting-a-personal-api-with-flask-and-code-capsules/
[82] https://codecapsules.io/
[83] https://www.python.org/
[84] https://github.com/
[85] https://git-scm.com/
[86] https://pypi.org/project/virtualenv/
[87] https://telegram.org/
[88] https://codecapsules.io/
[89] https://codecapsules.io/docs/creating-and-hosting-a-personal-api-with-flask-and-code-capsules/

# Registering a Bot Account and Talking to the BotFather

To create a Telegram bot, we need to download Telegram[90] and create a user account. You can use Telegram from either your PC or your phone, or both.

Once you have a Telegram account, you can register a new bot by sending a message to BotFather, a bot managed by Telegram themselves. Search for "BotFather" and initiate a chat. From the chat interface, follow these steps:

1. Press "start".
2. Type `/newbot`.
3. Choose a name for your bot.
4. Choose a username for your bot (must end in "bot").

Once you've chosen a username, the BotFather will reply with an *authorisation token*. This is a string that enables your bot to send requests to the Telegram Bot API, similar to the authorisation tokens we used to retrieve weather and exchange rate data in the personal API tutorial. Make sure to save this token somewhere safe and private.

To see if your bot was successfully created, search for the bot's username. You should see the bot and be able to start a conversation with it. Right now, our bot won't reply to anything you send it, as it doesnt have any backend code yet. Let's change that.

# Planning and Setup

We're going to implement two commands for our bot.

- When we send the command `/weather`, our bot will reply with the weather data from the API we created.
- When we send the command `/currency`, our bot will reply with the exchange rates from USD to CAD, EUR, and ZAR.

## Creating a virtual environment and installing requirements

First, we need to create a local directory. Give it the same name as our bot. Then, from this directory, open a terminal and create a Python virtual environment[91] by entering the following command:

[90] https://telegram.org/
[91] https://docs.python.org/3/tutorial/venv.html

```
virtualenv env
```

Enter the virtual environment using the appropriate command for your system:

+ **Linux/MacOSX**: `source env/bin/activate`
+ **Windows**: `env\Scripts\activate.bat`

The virtual environment will help manage our dependencies for when we host the bot on Code Capsules.

To interact with the Telegram Bot API, we need to install the python-telegram-bot[92] library, a Python wrapper for the Telegram Bot API[93]. We'll also use the Python library `requests` to retrieve data from the weather and currency exchange rate API. To install these requirements, enter the following in your terminal:

```
pip install python-telegram-bot requests
```

## Retrieving Data from the API

Now we can start coding. Create a file named `bot.py` in the same directory where we activated the virtual environment. In this file, enter the following code, replacing `YOUR-URL-HERE` with the URL pointing to the weather and exchange rate API hosted on Code Capsules.

```
import requests

url = 'YOUR-URL-HERE/GET'
data = requests.get(url) # requests data from API
data = data.json() # converts return data to json

# Retrieve values from API
curr_temp = data['curr_temp']
cad_rate = data['usd_rates']['CAD']
eur_rate = data['usd_rates']['EUR']
zar_rate = data['usd_rates']['ZAR']


def return_weather():
    print('Hello. The current temperature in Cape Town is: '+str(curr_temp)+" celsiu\
s.")


```

[92] https://github.com/python-telegram-bot/python-telegram-bot
[93] https://core.telegram.org/bots/api

```
def return_rates():
    print("Hello. Today, USD conversion rates are as follows: USD->CAD = "+str(cad_r\
ate)+
    ", USD->EUR = "+str(eur_rate)+", USD->ZAR = "+str(zar_rate))


return_weather()

return_rates()
```

Here we request the currency and weather data from the API and parse the temperature and conversion rates. Then we print out the data using `return_weather()` and `return_rates()`.

Try it out! Run the program to ensure everything works, then continue.

## Creating the Bot

Now we can get to creating the actual bot. At the top of the `bot.py` file, add this line:

```
from telegram.ext import Updater, CommandHandler
```

From the `python-telegram-bot` library, we import two classes: `Updater` and `CommandHandler`. We'll talk about these classes soon.

We don't need to print our data anymore – instead, we'll return a string to our bot, so the bot can display it on Telegram. Replace `def return_weather()` and `def return_rates()` with the following:

```
def return_weather():
    return 'Hello. The current temperature in Cape Town is: '+str(curr_temp)+" celsi\
us."


def return_rates():
    return "Hello. Today, USD conversion rates are as follows: USD->CAD = "+str(cad_\
rate)+", USD->EUR = "+str(eur_rate)+", USD->ZAR = "+str(zar_rate)
```

Now, replace the `return_weather()` and `return_rates()` function calls with the code below:

```
def main():
    TOKEN = "YOUR-BOT-TOKEN-HERE"
    updater = Updater(token=TOKEN, use_context=True)
    dispatcher = updater.dispatcher

    weather_handler = CommandHandler("weather", weather)
    currency_handler = CommandHandler("currency", currency)
    start_handler = CommandHandler("start", start)

    dispatcher.add_handler(weather_handler)
    dispatcher.add_handler(currency_handler)
    dispatcher.add_handler(start_handler)

    updater.start_polling()

if __name__ == '__main__':
    main()
```

At the top of our new `main` method, which will be called when this file is run, we instantiate `updater`, an instance of the Telegram library's `Updater`[94] class. This object will retrieve commands sent to our bot and pass them to an instance of the `Dispatcher`[95] class. We've assigned this `Dispatcher` instance to the variable `dispatcher` for further use.

Next, we create three different `CommandHandler` classes, one for each command that can be sent to our bot: `/start`, `/weather` and `/currency`. We pass two arguments into each instantiation: the command text (without the preceding `/`), and a function to call. For example, when a user enters the command `/weather`, the `weather()` function will be called.

Let's define that function, and the other two. Just above `def main()`, enter the following three function definitions.

```
def weather(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id, text=return_weather())

def currency(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id, text=return_rates())

def start(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id, text="Hi! I respond t\
o /weather and /currency. Try them!")
```

Each of these functions calls the `python-telegram-bot` function `send_message()` with the ID of the

[94] https://python-telegram-bot.readthedocs.io/en/latest/telegram.ext.updater.html#telegram.ext.updater.Updater
[95] https://python-telegram-bot.readthedocs.io/en/latest/telegram.ext.dispatcher.html#telegram.ext.Dispatcher

current chat and the appropriate text, either returned from one of our other functions or specified as a string. The `update` and `context` arguments are supplied automatically by the dispatcher.

Back in our `main()` function, we use `dispatch.add_handler` to add all three handlers to our dispatcher.

Finally, `updater.start_polling()` will begin polling[96] for updates from Telegram. This means our code will regularly ask Telegram's servers if any commands have been sent to it. Upon receiving commands, the appropriate handler will be invoked. In the next section, we'll discuss the pitfalls of polling and consider an alternative.

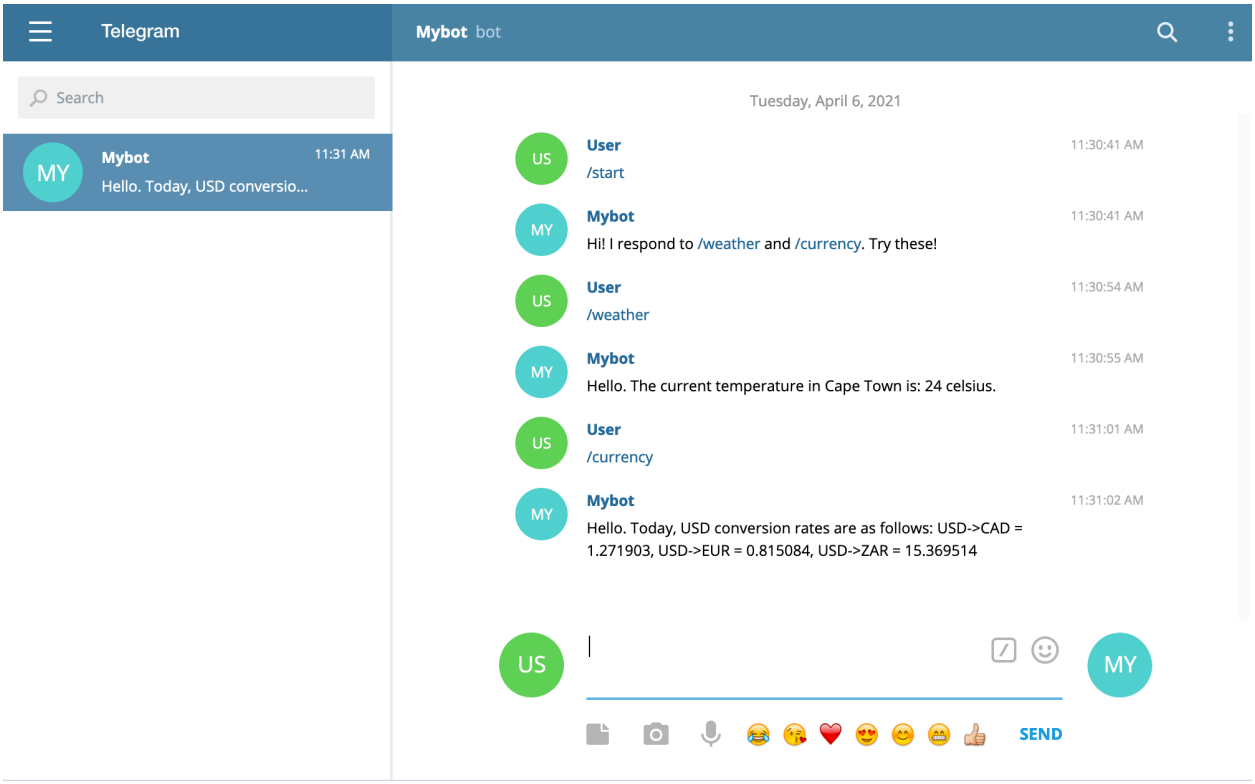
The code `bot.py` file should now look like the code below. Once again, make sure to replace `YOUR-URL-HERE` with the URL of the API you created in the API tutorial.

```
from telegram.ext import Updater, CommandHandler
import requests


url = 'YOUR-URL-HERE/GET'
data = requests.get(url) # requests data from API
data = data.json() # converts return data to json

# Retrieve values from API
curr_temp = data['curr_temp']
cad_rate = data['usd_rates']['CAD']
eur_rate = data['usd_rates']['EUR']
zar_rate = data['usd_rates']['ZAR']


def return_weather():
    return'Hello. The current temperature in Cape Town is: '+str(curr_temp)+" celsiu\
s."

def return_rates():
    return "Hello. Today, USD conversion rates are as follows: USD->CAD = "+str(cad_\
rate)+ ", USD->EUR = "+str(eur_rate)+", USD->ZAR = "+str(zar_rate)

def weather(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id, text=return_weather())

def currency(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id, text=return_rates())

```

[96] https://en.wikipedia.org/wiki/Polling_(computer_science)

```
def start(update, context):
    context.bot.send_message(chat_id=update.effective_chat.id, text='Hi! I respond t\
o /weather and /currency. Try these!')

def main():
    TOKEN = "YOUR-BOT-TOKEN-HERE"
    updater = Updater(token=TOKEN, use_context=True)
    dispatcher = updater.dispatcher

    weather_handler = CommandHandler('weather', weather)
    currency_handler = CommandHandler('currency',currency)
    start_handler = CommandHandler('start',start)

    dispatcher.add_handler(weather_handler)
    dispatcher.add_handler(currency_handler)
    dispatcher.add_handler(start_handler)

    updater.start_polling()

if __name__ == '__main__':
    main()
```

Below is a conversation with a bot created using this program. Run `bot.py` and try it out yourself.




**conversation**

We won't be able to send messages to our bot if this program isn't running, so hosting it on Code Capsules will allow us to interact with the bot without having to keep this code permanently running on our development PC.

While we could deploy our bot to Code Capsules in its current state, there is a downside to our current implementation that we should remedy first.

# Polling versus Webhooks

There are two ways for our `bot.py` file to receive commands sent to it on Telegram. Currently, the code polls Telegram constantly, regardless of whether the bot is in use. If we hosted this current version on Code Capsules, we would be wasting bandwidth, as the vast majority of polls would return nothing.

Instead of polling Telegram for changes, we can create a *webhook*[97]. This will allow us to receive commands as they are sent by Telegram users, without having to continuously ask Telegram servers for them.

We'll set up a webhook by telling Telegram to send commands sent to our bot account to our bot's Code Capsules URL. Our dispatcher will then process the command using the appropriate handler and send back the requested information.

[97] https://en.wikipedia.org/wiki/Webhook

## Creating a webhook

To set up the webhook, replace the line `updater.start_polling()` in the `main` function with the code below:

```
    PORT = int(os.environ.get('PORT','8443'))
    updater.start_webhook(listen='0.0.0.0', port=PORT, url_path=TOKEN)
    updater.bot.setWebhook('YOUR-CODECAPSULES-URL-HERE' + "/" + TOKEN)
    updater.idle()
```

Here we start a webhook that will listen on our Code Capsules URL at TCP port 8443 and with the path of our token. Thus, Telegram will relay commands sent to our bot to the following URL:

```
https://YOUR-CODECAPSULES-SUBDOMAIN.codecapsules.io:8443/TOKEN
```

If you've completed some of our other backend tutorials, you will be familiar with setting up web servers that receive `GET` and `POST` requests to different routes. You can think of a webhook as a very simple HTTP server that is intended to be used by bots and automated services rather than humans.

# Preparing For Deployment

Before we push our code to GitHub and deploy it on Code Capsules, we need to make one small code change and create some files.

## Creating an API key environment variable

Because we'll push our code to GitHub, we need to hide our bot's authentication key. If we don't, anyone could use our authentication key and take control of our bot.

Replace this line

```
TOKEN = "YOUR-BOT-TOKEN-HERE"
```

with the below

```
TOKEN = os.getenv('BOTAPIKEY')
```

`os.getenv('BOTAPIKEY')` will look for an environment variable[98] with the name "BOTAPIKEY". When we host our bot on Code Capsules, we'll set this environment variable to the key we received from the BotFather.

With that done, we must now create some files before we can push our code to GitHub and deploy it on Code Capsules.

[98] https://medium.com/chingu/an-introduction-to-environment-variables-and-how-to-use-them-f602f66d15fa

## Creating a Procfile and requirements.txt

Code Capsules requires a couple of files to deploy our application: `Procfile` and `requirements.txt`. The first one tells Code Capsules how to run our application, and the second one tells it which libraries it needs to install.

To create the `Procfile`:

1. Navigate to the directory containing the `bot.py` file and enter the virtual environment.
2. Create a file named `Procfile` (with no file extension).
3. Open `Procfile`, enter `web: python3 app.py`, and save the file.

In the same directory, open a terminal and activate the virtual environment. Then enter `pip3 freeze > requirements.txt` to generate a list of requirements for our Code Capsules server.

Now we can push our code to GitHub. Create a GitHub repository and send the `requirements.txt`, `Procfile`, and `bot.py` files to the repository.

# Deploying the Bot to Code Capsules

With all of the files sent to GitHub, let's deploy the bot to Code Capsules:

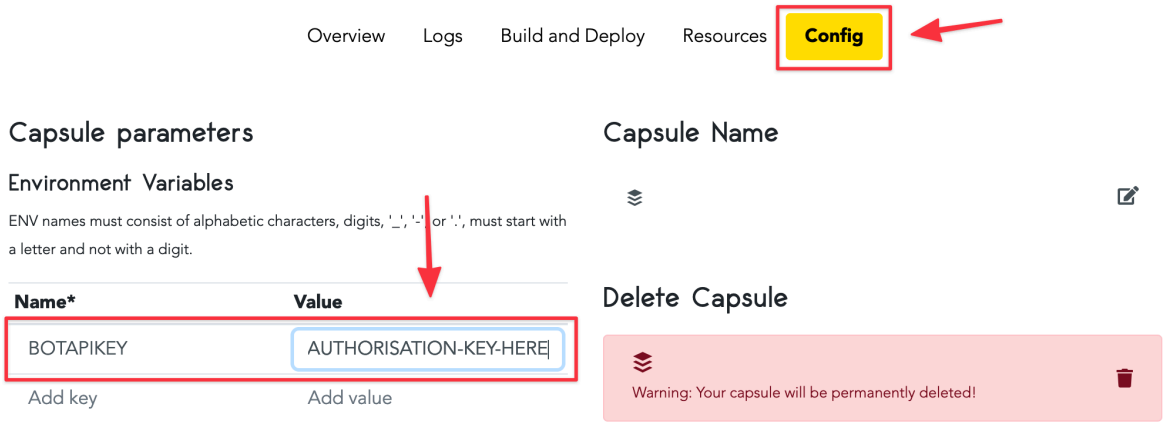
1. Log in to Code Capsules and create a Team and Space as necessary.
2. Link Code Capsules to the GitHub repository created [#creating-the-procfile-and-detailing-requirements

).

3. Enter your Code Capsules Space.
4. Create a new Capsule, selecting the "Backend" capsule type.
5. Select the GitHub repository containing the bot – leave "Repo subpath" empty and click "Next".
6. Leave the "Run Command" blank and click "Create Capsule".

We haven't supplied our webhook a URL yet, and we still need to create an environment variable for our bot's authorisation token. To create an environment variable:

1. Navigate to your Capsule.
2. Click the "Config" tab.
3. Add an environment variable with the name "BOTAPIKEY" and give it your bot's API key as a value. Make sure to hit the update button after adding the variable.




**env-var**

Next, let's supply our webhook with the correct domain.

1. Navigate to the "Overview" tab.
2. Copy the domain found under "Domains".
3. Open the `bot.py` file and find the line `updater.bot.setWebhook('YOUR-CODECAPSULES-URL HERE'+TOKEN)`.
4. Replace "YOUR-CODECAPSULES_URL" with the domain just copied.
5. Commit and push these changes to GitHub.

After pushing these changes, the Capsule will rebuild. Once this is done, the bot is ready. Give it a try!

# Further Reading

We've covered a lot above, from creating a Telegram bot to the differences between webhooks and polling.

If you're interested in learning more about what you can do with Telegram bots, check out Telegram's bot developer introduction[99]. If you have some ideas but require a deeper understanding of the `python-telegram-bot` library, browse their GitHub repository[100].

You can find a thorough explanation of webhooks in this blog post[101].

[99] https://core.telegram.org/bots
[100] https://github.com/python-telegram-bot/python-telegram-bot
[101] https://www.chargebee.com/blog/what-are-webhooks-explained/

# Developing a Persistent Sleep Tracker Part 1: Handling Users with Flask-Login

## Introduction

In this two-part tutorial series, we'll learn how to create a sleep tracker web application hosted on Code Capsules. Users will register an account with the sleep tracker and log in. To track their sleep data, users will enter a date and number of hours slept. We'll present users with a graph showing the sleep data they've logged, so users can get a visual representation of their sleep habits over time.

Throughout this tutorial series, we'll use many tools to create an interactive experience. We'll learn how to:

- Create a user login and register system with Python's Flask[102].
- Use a MongoDB[103] NoSQL database to store data.
- Create interactive Plotly[104] graphs.

This tutorial series is best suited for those with some Python, HTML, and Flask experience. But even if you feel you don't have much experience with these, don't worry. We'll walk through this application step-by-step. Let's get started!

## MongoDB Atlas

One of the most important aspects of this tutorial is using a Mongo Database (MongoDB). With this MongoDB, we can track users' login information and sleep data. MongoDB is a NoSQL databases, which means we can store data easily, in variable formats, without having to first create tables, as we would have to with a traditional SQL database. If you're unfamiliar with NoSQL databases or MongoDB in general, take a look at this explainer[105] by the MongoDB organisation.

MongoDB Atlas clusters are free to use. Follow this short tutorial[106] to create the MongoDB Atlas cluster that we'll use to store the user data for this sleep tracker application. This step is *extremely* important – without a database, our application will not function.

Once you've set up a MongoDB Atlas cluster, continue with this tutorial.

[102] https://flask.palletsprojects.com/en/1.1.x/
[103] https://www.mongodb.com/what-is-mongodb
[104] https://plotly.com/python/
[105] https://www.mongodb.com/nosql-explained
[106] https://codecapsules.io/docs/how-to-connect-a-mongodb-using-mongodb-atlas-with-your-code-capsules-application

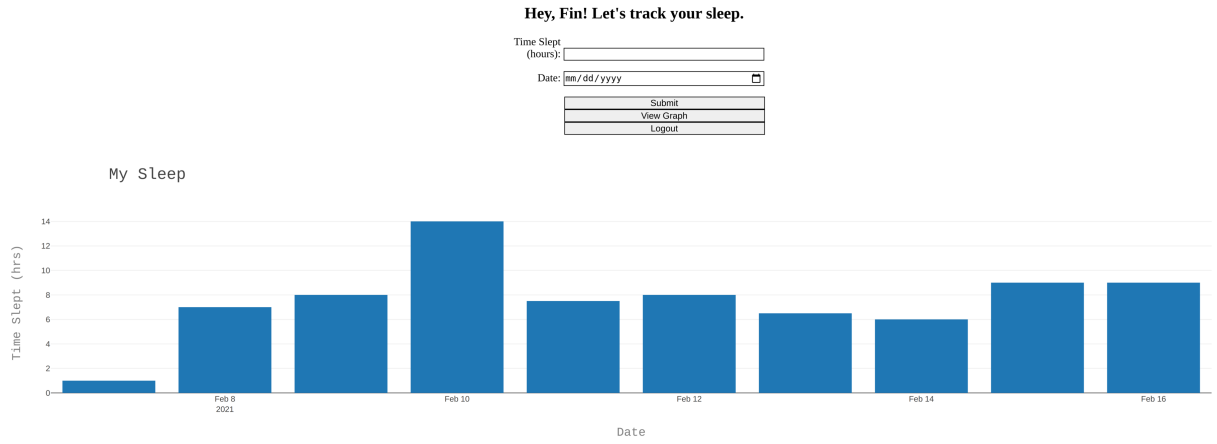
# Requirements

In addition to creating a MongoDB Atlas Cluster, make sure you have the following:

- Git[107] installed and a registered GitHub[108] account.
- Virtualenv[109] installed.
- A registered Code Capsules[110] account.

# Project Setup and Introduction

Creating this sleep tracker will be a two-part process. First, we will create a login and registration page and a user management backend. Second, we will create a page where users enter their sleep data and view a graph.




**sleep**

This tutorial will focus on the first part: dealing with user management. To start, create a `sleep-tracker` directory somewhere on your computer. All of our project's files will be in this directory.

## Setting up Virtual Env

With our `sleep-tracker` directory created, we need to set up a virtual environment[111]. Setting up a virtual environment will be useful when we host our web application on Code Capsules. Virtual

[107] https://git-scm.com/
[108] https://github.com
[109] https://pypi.org/project/virtualenv/
[110] https://codecapsules.io
[111] https://realpython.com/python-virtual-environments-a-primer/

environments ensure that only the libraries used in the development of our sleep tracker application will be installed by Code Capsule's servers.

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

Then, activate the virtual environment with:

+ **Linux/MacOSX**: `source env/bin/activate`
+ **Windows**: `\env\Scripts\activate.bat`

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

## Installing requirements

Our sleep tracker will use the following Python libraries:

- Flask[112] is a lightweight Python web development framework.
- Flask-Login[113] provides user session management for Flask. This will help us implement a user login and registration system without having to create one from scratch.
- Flask Bcrypt[114] is a hashing[115] extension for Flask. This allows us to store users' passwords securely and without knowing what they are.
- Gunicorn[116] is the WSGI server[117] we'll use to host our application on Code Capsules.
- Pymongo[118] is a Python library that has tools for interacting with MongoDBs. We'll use Pymongo to connect and send data to our MongoDB hosted on MongoDB Atlas.

To install these libraries, activate the virtual environment in your terminal and type the following:

```
1 pip3 install flask flask-login flask-bcrypt gunicorn pymongo
```

Next, we'll create all the files and directories that we'll use in both parts of this series.

## Creating the file structure

Because we'll use Flask to render our HTML files and serve static content, we need to have a specific project structure. Flask expects to find HTML files in a directory named `templates` and static content such as CSS stylesheets and images in a directory named `static`. In the `sleep-tracker` directory, create both of these directories.

Inside `templates`, create three files: `base.html`, `login.html`, and `main.html`.

[112] https://flask.palletsprojects.com/en/1.1.x/
[113] https://flask-login.readthedocs.io/en/latest/
[114] https://flask-bcrypt.readthedocs.io/en/latest/
[115] https://en.wikipedia.org/wiki/Cryptographic_hash_function
[116] https://gunicorn.org/
[117] https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
[118] https://pymongo.readthedocs.io/en/stable/

- `base.html` will contain the skeleton for our website's other HTML pages.
- `login.html` will contain the HTML for the main page of our web application. Here, users will log in or register an account with our sleep tracker.
- `main.html` will contain the HTML for the page where users will enter their sleep data and view a graph of this data. We'll deal with this file in the second part of this tutorial.

Next, open the `static` directory and create a file named `style.css`. This will be the only file in this directory, and will contain the CSS style for our website.

Finally, in the main `sleep-tracker` directory, create a file named `app.py`. In this file, we'll write the Python code that serves our HTML content to user and manages their activities on the web application.

# Creating the HTML Templates

Flask uses the Jinja[119] templating library to allow us to embed Python-like code in HTML. This will allow us to create web pages which change dynamically in response to user actions such as registration, login and entering sleep tracking data. In this section, we will be populating the files we created in `templates` above, starting with `base.html`.

## Base.html

The `base.html` file will contain all of the HTML code common throughout our application. This allows us to use it as a skeleton for every other page and avoid repeating standard markup such as stylesheet links. Open the `base.html` file and enter the following markup:

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <title> Sleep Tracker </title>
    <meta name="author" content="your-name-here">
    <meta name="description" content="This web-application
    helps you track your sleep!">
    <link rel="stylesheet" href="{{url_for('static',filename='style.css')}}">
  </head>

<body>
{% block content %}{% endblock %}
```

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

```
15
16 </body>
17 </html>
```

This is our skeleton. When Flask serves our `login.html` or `main.html` pages, it will replace the `{% block content %}{% endblock %}` with that page's unique content.

Any code between `{%` and `%}` or `{{` and `}}` is Jinja syntax, which is largely similar to Python code. Flask will evaluate this code before rendering[120] HTML files and serving them to users. In this example, we use Jinja syntax to call Flask's `url_for()` function. To link our stylesheet, Flask uses the `url_for()` function to find the `style.css` file in the `static` directory.

## Creating the Login Page

Now that we've created the `base.html` file, we can implement our `login.html` file. Open `login.html`and add the following markup:

```
1  {% extends "base.html" %}
2
3  {% block content %}
4  <h2>Login or register here to track your sleep!</h2>
5  <form action="" method="POST">
6    <ul>
7      <li>
8        <label for="username">Username:</label>
9      <input type="text" id="name" name="user_name">
10     </li>
11     <li>
12       <label for="password">Password:</label>
13     <input type="password" id="password" name="user_pw">
14     </li>
15     <li class = "button">
16       <input type="submit" name='login' value='Login'>
17     <input type="submit" name='register' value='Register'>
18     </li>
19   </ul>
20 </form>
21 {% endblock %}
```

The line `{% extends "base.html" %}` tells Jinja to render this page by populating each of `base.html`'s named `block` directives[121] with the corresponding `block` content defined in `login.html`. In this

[120] https://flask.palletsprojects.com/en/1.1.x/tutorial/templates/
[121] https://jinja.palletsprojects.com/en/2.11.x/templates/#template-inheritance

instance, we've only defined a single block named `content`, but we could define multiple blocks. For example, we might want to have a `head` block that defines some page-specific content that needs to be in the HTML `<head>` tag.

On this page, we've created a form containing input fields for users to enter their username and password, as well as login and register buttons. Note the `POST`[122] HTTP method – this will ensure that the username and password are sent as `POST` parameters in the HTTP request body, rather than as `GET` parameters in the URL. This allows us to differentiate between when a user visits the login page (a `GET` request) versus when they click the register or login button.

Before we take a look at our work so far, let's populate our `style.css` file to make our HTML look a bit better.

## Adding styles

Open the `style.css` file in the `static` directory and add the following:

```
form {
  margin: auto;
  width: 500px;
}

form li + li {
  margin-top: 1em;
}

ul {
  /* Remove unordered list dots */
  list-style: none;
  padding: .1;
  margin: .1;
}

label {
  display: inline-block;
  width: 100px;
  text-align: right;
}

input,
textarea {
  font: "Times New Roman", serif;
```

[122] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

```
  /* Change border & width of textarea */
  border: 1px solid #000000;
  width: 300px;
  box-sizing: border-box;
}


.button {
  /* Align w/ text box */
  padding-left: 105px;
}

h2 {
  text-align: center;
  font: "Times New Roman", serif;
}
```

Feel free to add your own personal touches, such as a favourite colour or font.

## Testing what we have

Now that we've created our templates and stylesheet, let's add some initial functionality to `app.py`. Open the file and enter the following code:

```
from flask import Flask, render_template, url_for, request, redirect

app = Flask(__name__)

## Login/Register page
@app.route('/')
def login():
    return render_template('login.html')

@app.route('/main')
def main():
    return render_template('main.html')
```

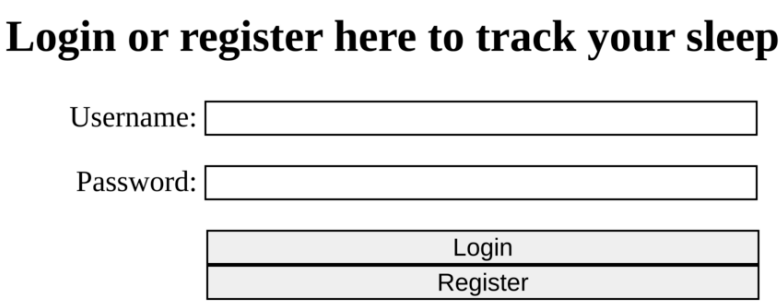
Here we've created two routes[123]:
+ `/`, which will be our web application's homepage and serve the content of `login.html`.
+ `/main`, which will be our web application's sleep tracker page and serve the content of `main.html`.

[123] https://flask.palletsprojects.com/en/1.1.x/quickstart/#routing

Flask uses the `@app.route` function decorator[124] to determine which functions will be executed when the user visits different routes.

View the application by opening a terminal in the `sleep-tracker` directory and and entering `flask run`.

After running the application, Flask will provide you with URL. Enter this URL in your web browser, and you'll see the login screen. It should look something like this:




**login**

# Handling User Registration and Login

Rather than implementing a user registration and login system ourselves, we'll use the Flask Login library that we installed earlier to provide this functionality. This will help us to save time and avoid common usability and security pitfalls. We will store user information in the database we created at the beginning of this tutorial.

First, let's import the functionality we'll need from:
+ `flask-login`, `flask-bcrypt`, to handle users.
+ `pymongo`, to interface with MongoDB.
+ `re`, to validate user input.

Enter the following code below the line that starts with `from flask import`:

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

```
from flask_login import LoginManager, UserMixin, login_required, login_user, logout_\
user, current_user
from flask_bcrypt import Bcrypt
import pymongo, re
```

Now we can create an instance of the `LoginManager` class and set a secret key[125] for our application. Just below the line that defines `app`, add:

```
app.config['SECRET_KEY'] = 'your-secret-key-here'
login_manager = LoginManager(app)
```

This class does exactly what it says – manages logged in users and communicates any necessary information about a user to Flask. Replace `your-secret-key-here` with a long randomly generated string. UUIDs[126] are good for this purpose. You can generate one with the following terminal command:

```
python3 -c 'import uuid; print(uuid.uuid4().hex.upper())'
```

Next, we need to initialise a bcrypt object for our application. Add the following code below the line that defines `login_manager`:

```
bcrypt = Bcrypt(app)
```

When a user registers with our sleep tracker, we'll create a new entry in our MongoDB with the user's username and password. That way, when a user logs in to our sleep tracker, we can see if the information they entered matches the information in our MongoDB. Beneath the last line we added, add the following:

```
client = pymongo.MongoClient('mongodb+srv://YOURUSERNAME:YOURPASSWORD@cluster0.e2fw3\
.mongodb.net/<dbname>?retryWrites=true&w=majorhostity')
db = client.user_login
```

Here we import the `pymongo` library and use it to connect to our MongoDB instance on MongoDB Atlas. Replace `YOURUSERNAME` and `YOURPASSWORD` with the MongoDB Atlas user account information you created in this tutorial[127].

[125] https://flask.palletsprojects.com/en/1.1.x/quickstart/#sessions
[126] https://en.wikipedia.org/wiki/Universally_unique_identifier
[127] https://codecapsules.io/docs/how-to-connect-a-mongodb-using-mongodb-atlas-with-your-code-capsules-application/

## Create the user class

For Flask Login to work, we need to create a `User` class. This `User` class will contain information pertaining to the user that is currently logged in to our sleep tracker.

Flask Login expects us to implement four methods in our `User` class: `is_authenticated`, `is_active`, `is_anonymous` and `get_id`. Rather than implementing all of these ourselves, we will have our `User` class inherit from Flask's `UserMixin`, which provides generic implementations for the first three, leaving us with only `get_id`. In addition, we'll need to implement our own `load_user` and `check_password` methods.

Below the line `db = client.user_login`, enter the following code:

```
class User(UserMixin):
    def __init__(self, username):
        self.username = username

    def get_id(self):
        return self.username
```

Here we've created the class, inherited from `UserMixin` and implemented the `get_id` method, which simply returns the user's username. To facilitate user login, we need to implement two methods, `load_user` and `check_password`. Let's implement `load_user` first. This method will be used to fetch a user from the MongoDB database corresponding to the username entered in the login form. Add the following code to the `User` class:

```
    @login_manager.user_loader
    def load_user(username):
        user = db.users.find_one({ "username": username })
        if user is None:
            return None
        return User(username=user['username'])
```

The decorate `@login_manager.user_loader` tells the `login_manager` to use this method to load users.

Now let's add a `check_password` method. This method will be called to determine whether a user has entered the correct password for the username they specify. When we register users, we will hash their passwords with the bcrypt[128] one-way encryption function before storing them in MongoDB. This will keep our users' passwords secret from us and, should our application be compromised in future, ensure that hackers cannot easily recover our users' passwords and use them on other websites those users may have account on. Therefore, when checking the password a user has entered on login, we need to hash this input with bcrypt before comparing it to the password entry in our MongoDB database. To do this, add the following code to the `User` class:

[128] https://en.wikipedia.org/wiki/Bcrypt

```
    @staticmethod
    def check_password(password_entered, password):
        if bcrypt.check_password_hash(password, password_entered):
            return True
        return False
```

By making `check_password` a static method[129], we enabled it to be called without instantiating an instance of our `User` class.

That's all we need for user login. Now we need to link our login and registration form to this functionality.

## Add functionality to the login and register buttons

When a user clicks the register button, we will create a new user document in our MongoDB users collection, containing their username and hashed password. When a user clicks "Login", we will log them in if they've entered a valid username and password combination and redirect them to the `main.html` file.

To do this, we'll create a new function that handles login and registration. Add the following code below the `def login()` function:

```
@app.route("/", methods = ["POST"])
def login_or_register():
    if request.method == 'POST':
        name_entered = str(request.form.get('user_name')) # Get username and passwor\
d from form
        pw_entered = str(request.form.get('user_pw'))

        if request.form.get('login'): # Log in logic
            user = db.users.find_one({ 'username': name_entered })
            if user and User.check_password(pw_entered, user['password']):
                usr_obj = User(username=user['username'])
                login_user(usr_obj)
                return redirect(url_for('main'))
            else:
                return "Incorrect username or password."

        elif request.form.get('register'): # Register logic
            # Validate username and password
            if not re.match("[a-zA-Z0-9_]{1,20}", name_entered):
                return "Username must be between 1 and 20 characters. Letters, numbe\
```

[129] https://realpython.com/instance-class-and-static-methods-demystified/

```
rs and underscores allowed."
            if len(pw_entered) < 8:
                return "Password must be at least 8 characters."

            if db.users.find_one({ 'username': name_entered }):
                return "User already exists."

            new_user = { 'username': name_entered,
                         'password': bcrypt.generate_password_hash(pw_entered) }
            db.users.insert_one(new_user) # insert new user to db
            return redirect(url_for('login')) # redirect after register
```

First, note the `POST` method. As mentioned when we created our `login.html` file, using a `POST` method for user login and registration allows our application to transmit username and information more securely and allows us to differentiate between a user login or registration (`POST`) and a user merely visiting the page (`GET`).

If the login button is pressed (`if request.form.get('login')`), we check our MongoDB for a username that matches the one entered. Then we check if the password entered matches that user's password in the MongoDB. If `check_password` evaluates to true, we log the user in and redirect to the `main` route (which we'll create in the next part of this series[130]. Otherwise, we provide the user with an error message. To redirect users, we use Flask's `redirect` function and `url_for` functions. The `url_for` function finds the `main` route, and the `redirect` function sends users to that route.

If a user clicks "Register", we first validate the username and password they've provided. We're restricting usernames to a length of 20 characters, containing only alphanumeric characters and underscores. We're also ensuring that the chosen password is eight or more characters long. We then check whether the username they're trying to use is already taken. If their username and password are acceptable, we create a `new_user` dictionary with the specified name and a bcrypt hash of the specified password, which we then insert it into our MongoDB. Then we send the user back to the login page.

## Trying out the login system

We've implemented the login and register buttons. Try running the program by opening a terminal in the `sleep-tracker` directory and entering `flask run`. Test out registering a few new accounts and logging into them. Remember, we haven't put any HTML in our `main.html` file, so when you log in, you'll see a blank page. Don't worry, everything is working!

In the next part of this series[131], we'll implement the rest of the sleep tracker application. This means populating the `main.html` file and learning how to store user sleep data in MongoDB.

[130] https://codecapsules.io/docs/developing-a-persistent-sleep-tracker-part-2-tracking-and-graphing-sleep-data
[131] https://codecapsules.io/docs/developing-a-persistent-sleep-tracker-part-2-tracking-and-graphing-sleep-data

# Further Reading

To learn more about Flask-Login, take a look at their documentation[132]. The Explore Flask documentation site also has a guide[133] for handling users that goes into further depth than we have, including features such as a way for users to reset forgotten passwords.

For more information on the Jinja templating language, their documentation can be found here[134].

Finally, when you're ready, finish the sleep tracker application by following the second tutorial in this series[135].

[132] https://flask-login.readthedocs.io/en/latest/
[133] https://explore-flask.readthedocs.io/en/latest/users.html
[134] https://jinja.palletsprojects.com/en/2.11.x/templates/
[135] https://codecapsules.io/docs/developing-a-persistent-sleep-tracker-part-2-tracking-and-graphing-sleep-data

# Developing a Persistent Sleep Tracker Part 2: Tracking and Graphing Sleep Data

This is the second part of our sleep tracker web application tutorial, covering the creation of the sleep tracker interface.

## Recap

In the first part of this series[136], we built a web application with user registration and login using Python's Flask web framework and a hosted NoSQL database on MongoDB Atlas for data persistence. We will now build on the code we wrote in that tutorial, so you must have completed it.

In this second part of the series, we'll implement the logic that allows users to enter their sleep data and see that data on an interactive graph, generated using Plotly[137]. Registered users will be able to log the number of hours slept on different days and visualise this data as a graph.

## Creating the Sleep Tracker Front-end

At the end of our last tutorial, we saw a blank page when we logged in and were redirected to the application's `/main` page. This happened because our `main.html` file was empty, so let's add some content to it. We'll need the following:

1. A form with fields for the date and hours slept, so users can provide sleep data for different days.
2. A way to view a graph of this data.
3. A logout button, that logs the user out.

To do all of this, we will first need to build a `main.html` containing both static HTML and dynamic Jinja template components that will change depending on which user is logged in and what sleep data they've provided. We will also implement some front-end JavaScript code to make our sleep data graph interactive.

Let's add the sleep data logging form first. Open the `main.html` file in the `templates` directory. Add the following:

[136] https://codecapsules.io/docs/developing-a-persistent-sleep-tracker-part-1-handling-users-with-flask-login
[137] https://plotly.com/

```
{% extends "base.html" %}

{% block content %}
<h2>Hey, {{ user['username'] }}! Let's track your sleep.</h2>
<form action="" method="POST">
  <ul>
    <li>
      <label for="time">Time Slept (hours):</label>
    <input type="text" id="time" name="time">
    </li>
    <li>
      <label for="date">Date:</label>
    <input type="date" id="date" name="date"
        value="">
    </li>
    <li class = "button">
      <input type="submit" name='submit' value='Submit'>
    <input type="submit" name='graph' value='View Graph'>
    <input type="submit" name='logout' value='Logout'>
    </li>
  </ul>
</form>
{% endblock %}
```

This works similarly to the `login.html` file we created in the previous tutorial, with `base.html` acting as the page skeleton and our unique content being entered between the `{% block content %}` and `{% endblock %}` lines. We also ensure that our form uses the `POST` method so we can differentiate between a user visiting the page and clicking one of the three form buttons. To determine which button a user has clicked in a given `POST` request, we'll use the button's HTML `name` attribute (`submit`, `graph` or `logout`) in Flask.

Notice the Jinja snippet `{{ user['username'] }}`. This will display data sent from our Flask back-end code in the page – in this case, the user's name.

The `<input type="date" id="date" name="date" value="">` line creates an interactive calendar so users can click on dates rather than typing them out.

## Adding Sleep Data Submission and Logout

In the `app.py` file, add the following line below the `app = Flask(__name__)` line.

```
app.config['plotting'] = False
```

Here we add a new entry in our Flask application's configuration settings. This will come in handy soon – we'll use this line to tell whether or not a user has clicked the "View Graph" button in our `main.html` file. If a user has clicked the button, we'll set this line to "True" and a graph with the user's sleep data will display.

Now, we can implement the functionality for our "Submit", "View Graph" and "Logout" buttons. Add the following code below the `main()` function.

```
@app.route('/main', methods = ['POST'])
def submit_sleep():
    if request.form.get('submit'): # if submitting new sleep data
        time_entered = float(request.form.get('time'))
        date_entered = request.form.get('date')
        message = add_sleep(time_entered,date_entered,db.users.find_one({'username':\
current_user.get_id()}))
        if message:
          return message

    if request.form.get('logout'):
        logout_user()
        app.config['plotting'] = False
        return 'You logged out!'

    elif request.form.get('graph'):
        app.config['plotting'] = True

    return redirect(url_for('main'))
```

This function operates similarly as our `login_or_register` function:

+ If a user clicks "Submit", the data they entered will be stored in their MongoDB entry via the `add_sleep` function (that we'll create next). This function will return a string with an error message if it encounters an error, and `None` if it succeeds.
+ If a user clicks "Logout", the user will be logged out.
+ If a user clicks "View Graph", the `app.config['plotting']` entry is set to `True`. Later, we'll expand `main.html` to display the graph.

Let's wrap up our button functionality by creating the `add_sleep` function that is called when a user clicks "Submit". Add the following code **above** the `submit_logout_plot` function to create the `add_sleep` function:

```
def add_sleep(time, date, user):
    if not re.match("[0-9]{4}-[0-1][0-9]-[0-3][0-9]", date):
        return "Invalid date supplied."

    if time < 0.0 or time > 24.0:
        return "Sleep time must be between 0 and 24 hours."

    if 'date' in user:
        user['date'].append(date)
        user['time'].append(time)
    else: # adding sleep data for the first time
        user['date'] = [date]
        user['time'] = [time]

    # Update MongoDB Atlas
    db.users.update_one({ 'username': user['username'] },
        { '$set': { 'date': user['date'],
          'time': user['time'] }})
```

Our `add_sleep` function takes three variables:

- `time`: The number of hours slept that the user entered.
- `date`: The calendar date the user selected.
- `user`: The user's MongoDB entry.

First, we validate the user's input to ensure that a correctly formatted date has been provided and that the time given is not a negative number or larger than 24. If either value does not pass validation, we return a relevant error message from the function without writing to the database. Otherwise we continue.

If a user has never entered any sleep data, we add a new `'date'` and `'time'` entry to the user's MongoDB entry with the date and time entered. Otherwise, we take the data they entered and add it to their existing sleep data. Finally, we update the user's MongoDB entry with `db.users.update_one`.

We've implemented functionality for all three buttons. Now we need to modify the `main()` function to pass the current user's data to `main.html`. This will allow us to display their username on the page, and to graph their sleep data. Find the `main()` function and modify it like so:

```
@app.route('/main')
def main():
    if current_user.get_id() is None:
        return redirect(url_for('login')) # redirect to login page if not logged in

    user_data = db.users.find_one({ 'username': current_user.get_id() })
    return render_template("main.html", user=user_data, plot=app.config['plotting'])
```

First, we leverage Flask-Login's anonymous users[138] functionality to check if the current user is not logged in and, if so, we redirect them to the login page. If the current user is logged it, we retrieve their MongoDB entry and assign it to `user_data`. Then we pass this variable to the `render_template` function as `user`. This is how the line below will access and display the user's name.

```
<h2>Hey, {{ user['username'] }}! Let's track your sleep.</h2>
```

As `user_data` contains the entire MongoDB user entry, our template will be able to access the current user's sleep data from the `user` variable as well.

We've also passed the template the value of `app.config['plotting']` in `plot`. This is how our application will know whether or not to display a graph on the `/main` page.

All that's left now is to add the sleep data graph in our `main.html` file. After that, we can deploy our application to Code Capsules.

## Adding the Plotly Graph

As mentioned at the beginning of the article, we'll add the ability to graph sleep data with the help of Plotly[139]. Plotly provides an external JavaScript library that we can use to create interactive graphs for the web.

In the `main.html` file, find the `</form>` line. Right below this line, add the following:

```
{% if plot %}
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<div id="graph">
    <script>
      var x = {{user['date'] | safe }};
      var y = {{user['time'] | safe }};

    var trace1 = {
    x: x,
```

[138] https://flask-login.readthedocs.io/en/latest/#anonymous-users
[139] https://plotly.com/

```
    y: y,
    type: 'bar'
  };

  var data = [trace1];

  var layout = {

    title: {
      text:'My Sleep',
      font: {
        family: 'Courier New, monospace',
        size: 24
      },
      xref: 'paper',
      x: 0.05,
    },
    xaxis: {
      title: {
        text: 'Date',
        font: {
          family: 'Courier New, monospace',
          size: 18,
          color: '#7f7f7f'
        }
      },
    },
    yaxis: {
      title: {
        text: 'Time Slept (hrs)',
        font: {
          family: 'Courier New, monospace',
          size: 18,
          color: '#7f7f7f'
        }
      }
    }
  };
  Plotly.newPlot('graph', data, layout);
    </script>
</div>
{% endif %}
```

Let's break this down, starting with the line `{% if plot %}`. This line references the `plot` variable we created in our `app.py` file in the `main` function. If someone has clicked "View Graph", we set `plot` to true. If `plot` is true, the HTML between `{% if plot %}` and `{% endif %}` will be included in the page served to the user, otherwise it will be left out.

The line `<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>` imports the Plotly graphing library. This is similar to an `import` statement in Python.

Under this line, we see a lot of code enclosed in `<script>...</script>` tags. This is all JavaScript code. In this code, we create two variables, `x` and `y`, which contain arrays of the dates and number of hours slept that a user has logged.

Note that we have set these variables as `safe`, which means that Jinja will not attempt to escape or encode any of the characters within them when it renders the HTML. This is dangerous to do with user input, which is why we validated both the date and time values in our Python code. If we had not validated them, a malicious user might be able to supply JavaScript code in the date or time input fields and alter the behaviour of this page.

The code inside `var trace1 = {...}` tells Plotly which data to use for the x and y axes, and the type of graph we'll make – a bar graph.

All the code in `var layout = {...}` effects things like x and y axis labelling, font type, and size of font. Customise this to your liking!

Finally, The line `Plotly.newPlot('graph', data, layout)` creates the actual graph and displays it to a user.

Try running the application by opening a terminal in the `sleep-tracker` directory, activating the virtual environment and entering `flask run`. You should be able to register a new user account, log in, enter sleep data, and view your graph.

# Preparing for Deployment

With our sleep tracker functionally complete, we need to make one last modification to our `app.py` file and add some files in the `sleep-tracker` directory before we can push our code to GitHub and deploy it on Code Capsules.

## Creating environment variables

Before we push our code to GitHub, we need to remove our `app.config['SECRET_KEY']` and the MongoDB user credentials. If we were to push our code now, anyone could use our secret key to forge user sessions on our application or our MongoDB credentials to alter our database. Luckily, there is an easy fix.

First, save your secret key and MongoDB credentials somewhere safe, outside of this project's directory so that you don't lose them. Then, at the top of `app.py`, add the line:

```
import os
```

Then, replace the line:

```
app.config['SECRET_KEY'] = 'your-secret-key-here'
```

with this:

```
app.config['SECRET_KEY'] = os.getenv('SECRET_KEY')
```

And replace this line:

```
client = pymongo.MongoClient('mongodb+srv://YOURUSERNAME:<password>@cluster0.e2fw3.m\
ongodb.net/<dbname>?retryWrites=true&w=majorhostity')
```

with this:

```
client = pymongo.MongoClient('MONGO_CONNECTION_STRING')
```

`os.getenv('SECRET_KEY')` and `os.getenv('MONGO_CONNECTION_STRING')` will look for environment variables[140] with the names "SECRET_KEY" and "MONGO_CONNECTION_STRING". When we host the sleep tracker application on Code Capsules, we'll set these environment variables to the values we removed from the code.

## Creating a Procfile and requirements.txt

Code Capsules requires a couple of files to deploy our application: `Procfile` and `requirements.txt`. The first one tells Code Capsules how to run our application, and the second one tells it which libraries it needs to install.

To create the `Procfile`:

1. Create a file named `Procfile` in your project directory (do **not** add a file extension).
2. Open the `Procfile`, enter `web: gunicorn app:app`, and save the file. This tells Code Capsules to use the Gunicorn WSGI server to run `app.py`.

In the same terminal, activate the virtual environment and enter `pip3 freeze > requirements.txt` to create `requirements.txt` and populate it with all the libraries we've used to create this application.

Now we can push our code to GitHub. Create a GitHub repository and send every file and directory to GitHub, **except for virtual env's `env` directory**.

[140] https://medium.com/chingu/an-introduction-to-environment-variables-and-how-to-use-them-f602f66d15fa

# Deploying the Sleep Tracker to Code Capsules

With all of the files on GitHub, we can deploy the sleep tracer to Code Capsules:

1. Log in to Code Capsules, and create a Team and Space as necessary.
2. Link Code Capsules to the GitHub repository created previously.
3. Enter your Code Capsules Space.
4. Create a new Capsule, selecting the "Backend" capsule type.
5. Select the GitHub repository containing the sleep tracker – leave "Repo subpath" empty and click "Next".
6. Leave the "Run Command" blank and click "Create Capsule".

Now we just need to set those environment variables we mentioned previously.

## Creating environment variables in Code Capsules

Let's create set the environment variables so our sleep tracker will work properly:

1. Navigate to your Capsule.
2. Click the "Config" tab.
3. Add two environment variables, one named "SECRET_KEY" and another "MONGO_CONNECTION_STRING". Enter the secret key and connection string values you saved earlier.

When done, **make sure** to click "Update".

Now the sleep tracker is ready to try out! The application is complete.

# What Next?

There are many ways to expand or improve this application. Some ideas include:

- Improve the application's styling – it's fairly simple right now. If you want to learn more about CSS styling, this tutorial written by Mozilla[141] is a great place to start.
- Add a way for users to keep track of other data (calories, daily notes, exercise).
- Display better looking error messages, preferably somewhere in the current page.

[141] https://developer.mozilla.org/en-US/docs/Web/CSS

# Build a Slackbot with Node.js to Monitor your Applications

Slack is a really useful communication tool when working in teams. Many developers find themselves using it almost constantly when working on projects.

One of the stand out features of Slack is the rich API it exposes, to allow developers to integrate with it.

In this tutorial, we'll use the Slack API to give our apps a voice. We'll be able to talk to our apps running on Code Capsules, to ask their status and see if they are up and running. They will also be able to alert us when they boot up, so we know if they have been successfully deployed or restarted.

## Overview and Requirements

As we're building a Slackbot, you'll need to sign up for an account on Slack[142], if you haven't already got one. Ideally, for this tutorial you should use a Slack workspace that you can safely send many test messages to while we are creating this bot, without disturbing people.

We'll also need the following:

- Git[143] set up and installed, and a registered GitHub[144] account.
- Node.js[145] installed.
- A registered Code Capsules[146] account.
- An IDE or text editor to create the project in. This tutorial was made using Visual Studio Code[147], but feel free to use any tool you like.

## Setting Up the Project

With our requirements in place, we can get started on setting them up to work as needed for our Slackbot project.

[142] https://slack.com
[143] https://git-scm.com/downloads
[144] https://github.com/join
[145] https://nodejs.org/en/download/
[146] https://codecapsules.io/
[147] https://code.visualstudio.com/

## Create a new repo on GitHub

We need a place to store our code from which Code Capsules can deploy to a capsule. A repository on GitHub is just what we need.

Head over to GitHub, and create a new repo. We're calling it slackbot here, but you can call it whatever you like.

Note: You can also add this code to an existing backend project if you would like to monitor it; perhaps something you built in an earlier tutorial.

## Initialise the base project

Now we can get some base code set up. Let's start by cloning the new GitHub repo onto our local computer.

Now, go into the directory of the repo you've just cloned.

We can initialise a new Node.js project by typing the following at the terminal (or command prompt, if you're on Windows):

```
1 npm init
```

We can just press enter for each of the questions it asks; the defaults are good to start with.

## Install our packages

Now that we have our project initialised, we can add the packages we need to create our bot. These are:

- Express[148]: This acts as our web server and HTTP request router. We'll use this to route requests from Slack to the correct logic.
- body-parser[149]: This interprets and parses payload data from HTTP requests. We'll need this to parse the URL-encoded data Slack sends to us with a request.
- superagent[150]: This package allows us to make outgoing HTTP requests. We'll need this to send a message to Slack.

Let's type in the following at the terminal to install the packages:

[148] https://expressjs.com
[149] https://www.npmjs.com/package/body-parser
[150] https://www.npmjs.com/package/superagent

```
npm install express body-parser superagent
```

Now let's create an `index.js` file, which will be the main file for our app. A simple way to do this is to open up your project folder in an editor, like Visual Studio Code[151]. Now create a blank file called `index.js`.

Great, it's time to push this boilerplate project up to Git. We can do it from the terminal with the following:

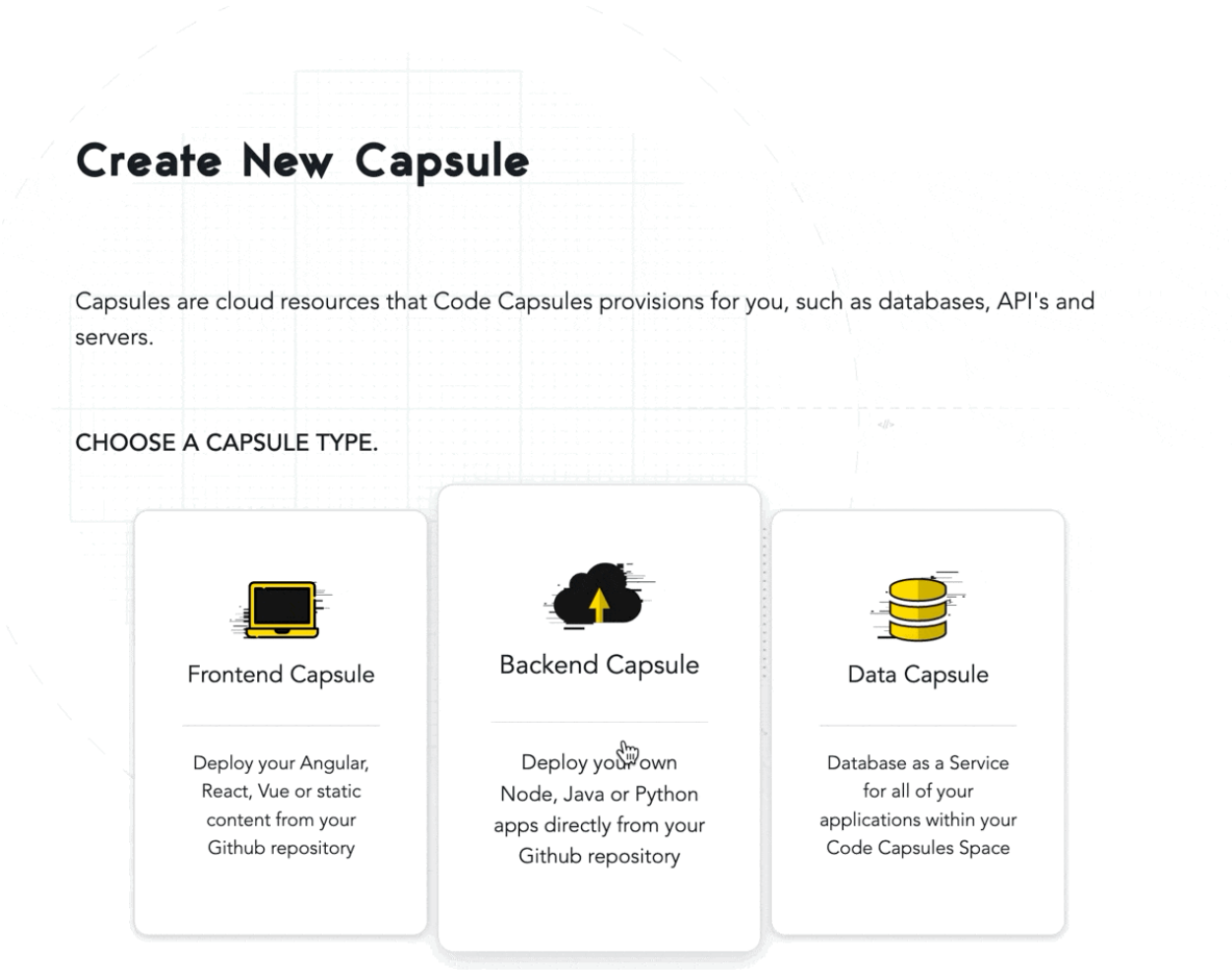
```
git add .
git commit -am 'added base files for project'
git push origin
```

## Create a new Code Capsule

We'll need a place to host our app.

1. Log in to Code Capsules[152], and create a Team and Space as necessary.
2. Link Code Capsules to the GitHub repository you created earlier. You can do this by clicking your username at the top right, and choosing "Edit Profile". Now click the "GitHub" button to link to a repo.
3. Create a new Capsule, selecting the "Backend" capsule type.
4. Select the GitHub repository you created above. If you are only using the repo for this project, you can leave the "Repo Subpath" field empty. You may need to add your repo to the team repo if you haven't already. Click the "Modify Team Repos" to do so.
5. Click "Next", then on the following page, click "Create Capsule".

[151] https://code.visualstudio.com
[152] https://codecapsules.io




**create capsule**

## Register an app on Slack

After you've created a workspace on Slack, or logged into an existing one, head over to https://api.slack.com[153] and click on "Create a custom app".

On the dialog that comes up, we can give our app a name, and choose which workspace we want to add it to. You can choose any name you wish – we've used Serverbot here. Now we can click "Create App".

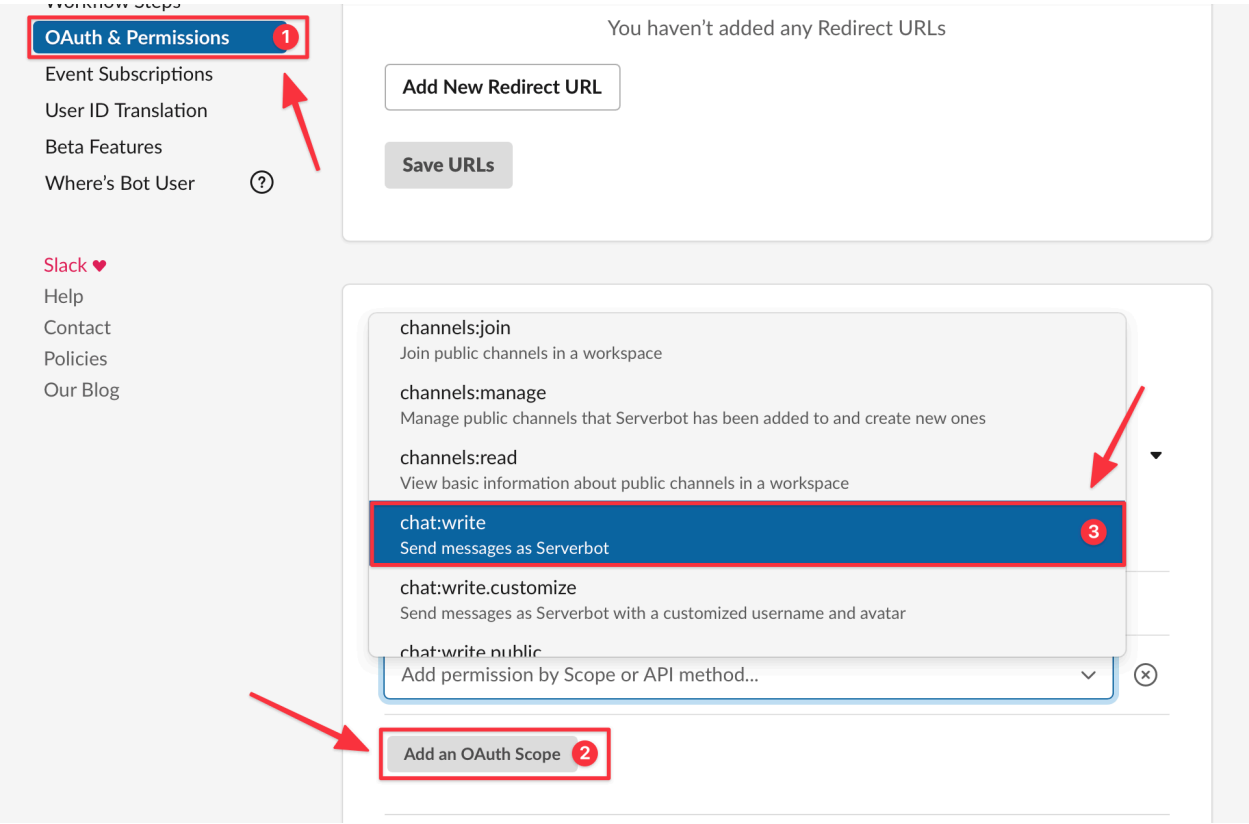
[153] https://api.slack.com

Great! We've created our app. Now we can configure it.

For this tutorial, we would like the following two functions:

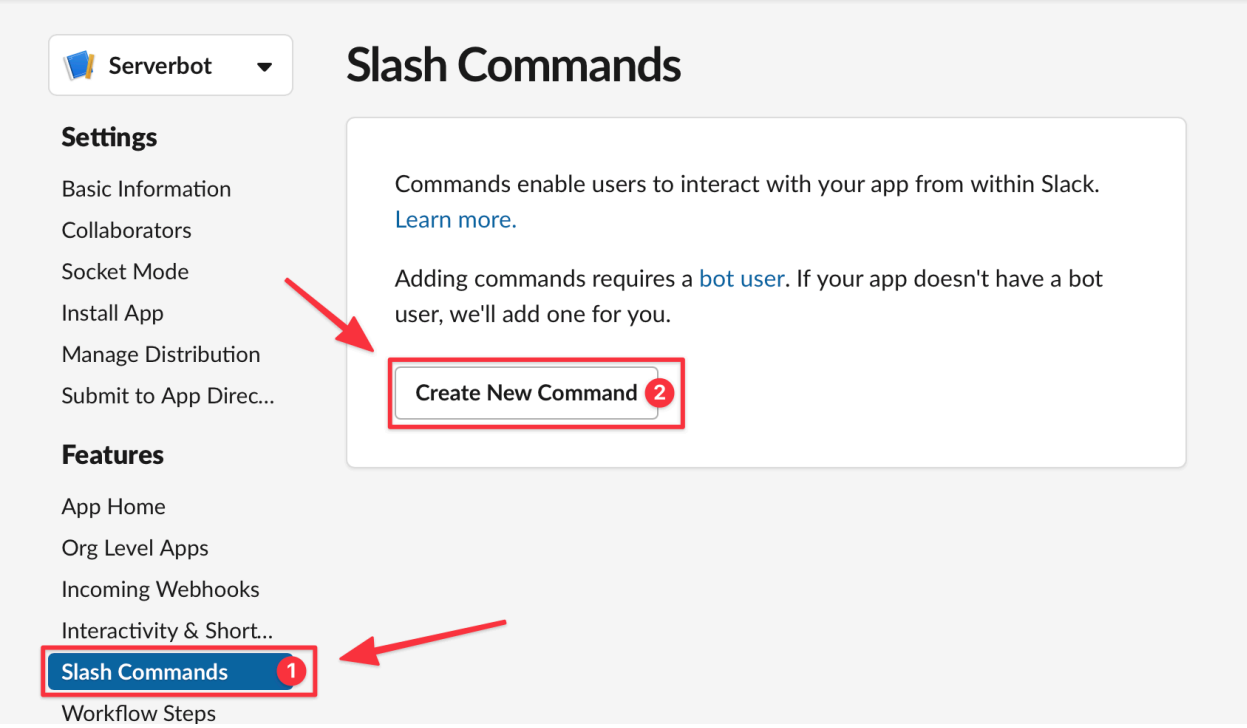
1. Our Code Capsules app should automatically send us a notification whenever it starts up. This allows us to easily know when a new deployment is successful. It can also alert us to any potential crashes and restarts.
2. We want to query our Code Capsules app from Slack at any time to see how it's doing.

Our first requirement can be configured on the Slack side by clicking "OAuth & Permissions" on the left panel. Scroll down to the Scopes section, and click "Add an OAuth Scope" under the Bot Token Scopes section, and choose "Chat:Write" from the options list. This now allows our bot to initiate and post messages to us when it starts up.



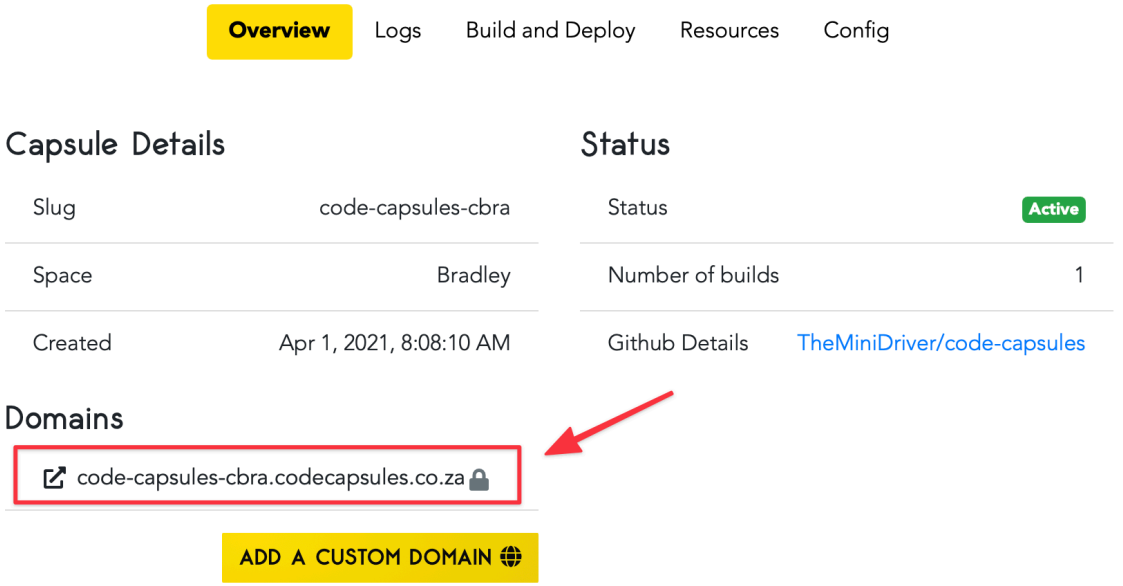

**select scopes slack**

Our second requirement can be configured by setting up a slash command. Click on the "Slash Commands" menu item on the left, under Features.



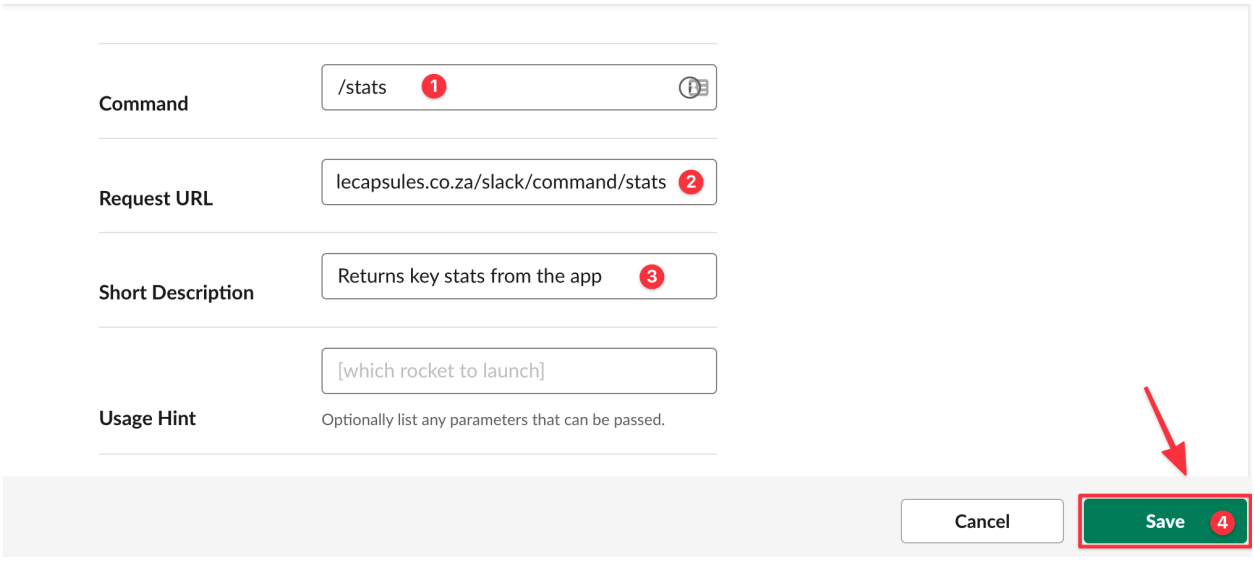

**slash command menu**

Then click “Create a new Command”. We’ll give the command the name /stats. For the Request URL, copy the Domain name from your Code Capsules Overview page.




**code capsules domain**

Paste your domain into the Request URL box on Slack, and add `/slack/command/stats` to the end of it. We can fill in a description as well, something like 'Returns key stats from the app'.




**create new command settings**

Great, now we can click "Save" at the bottom of the page to finish setting up our slash command.

# Writing the Slackbot Code

Now that we have all our systems set up, we can get onto the coding part.

## Adding the base code

Let's add the boilerplate code to startup a new Express server. Open up the `index.js` file and add the following:

```
const express = require('express');

const app = express();

let port = process.env.PORT || 3000;
app.listen(port, ()=>{
  console.log(`App listening on port ${port}`);
});
```

## Sending a startup message to Slack

Ok, cool, we've got the base code to create an Express app, and start it up to begin listening for requests. Now we can add some code to send a message to Slack when it boots up, not just locally to the console. If we look at the docs on Slack[154], we see that we can POST to the endpoint `https://slack.com/api/chat.postMessage` to send a message. In their example, they specify that we need:

1. An access token.
2. The channel ID of the channel to post the message to.
3. The message we want to post as the requirements.

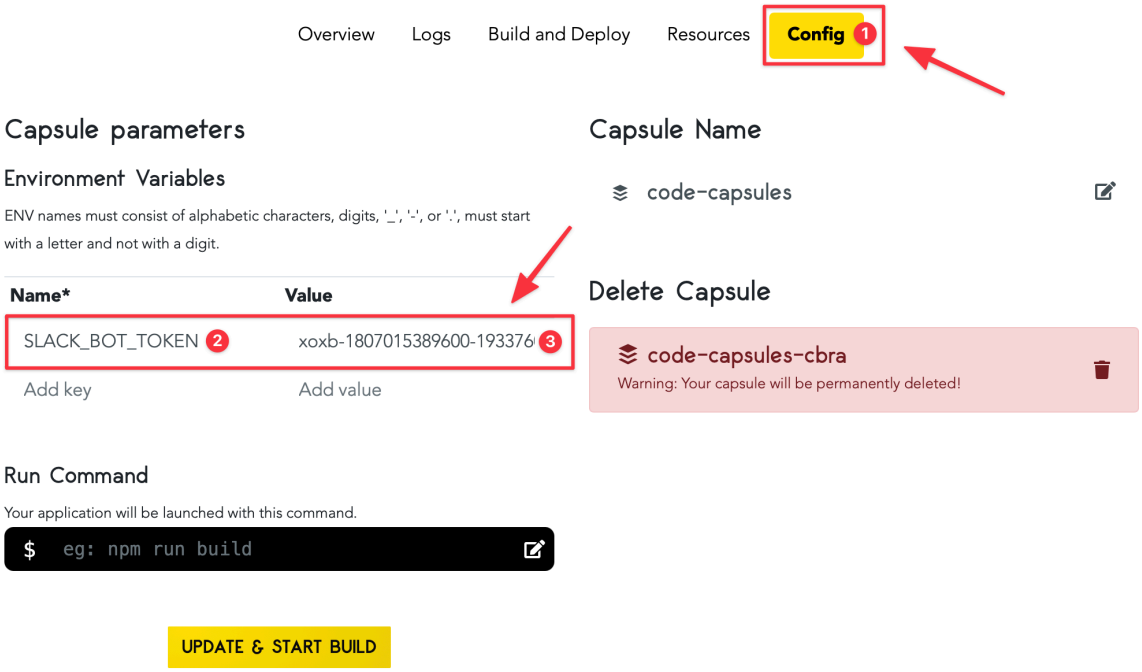
To get the access token, head over to your app dashboard on Slack, and click on the "OAuth & Permissions" menu item on the left-hand side. Then click the "Install to Workspace" button, and then the "Allow" button. After this, you should see a newly generated "Bot User OAuth Token". Copy this token – this is our access token.

We could just put this token in our code. However, this is not really considered best practice for sensitive secrets and credentials. Rather, let's add this secret as an **Environment Variable**, and access it from the Node.js process object, on the `.env` property[155].

To add the access token to the environment in Code Capsules, head over to the capsule we created earlier, and click on the "Config" tab. Now we can fill in our environment variable for the access

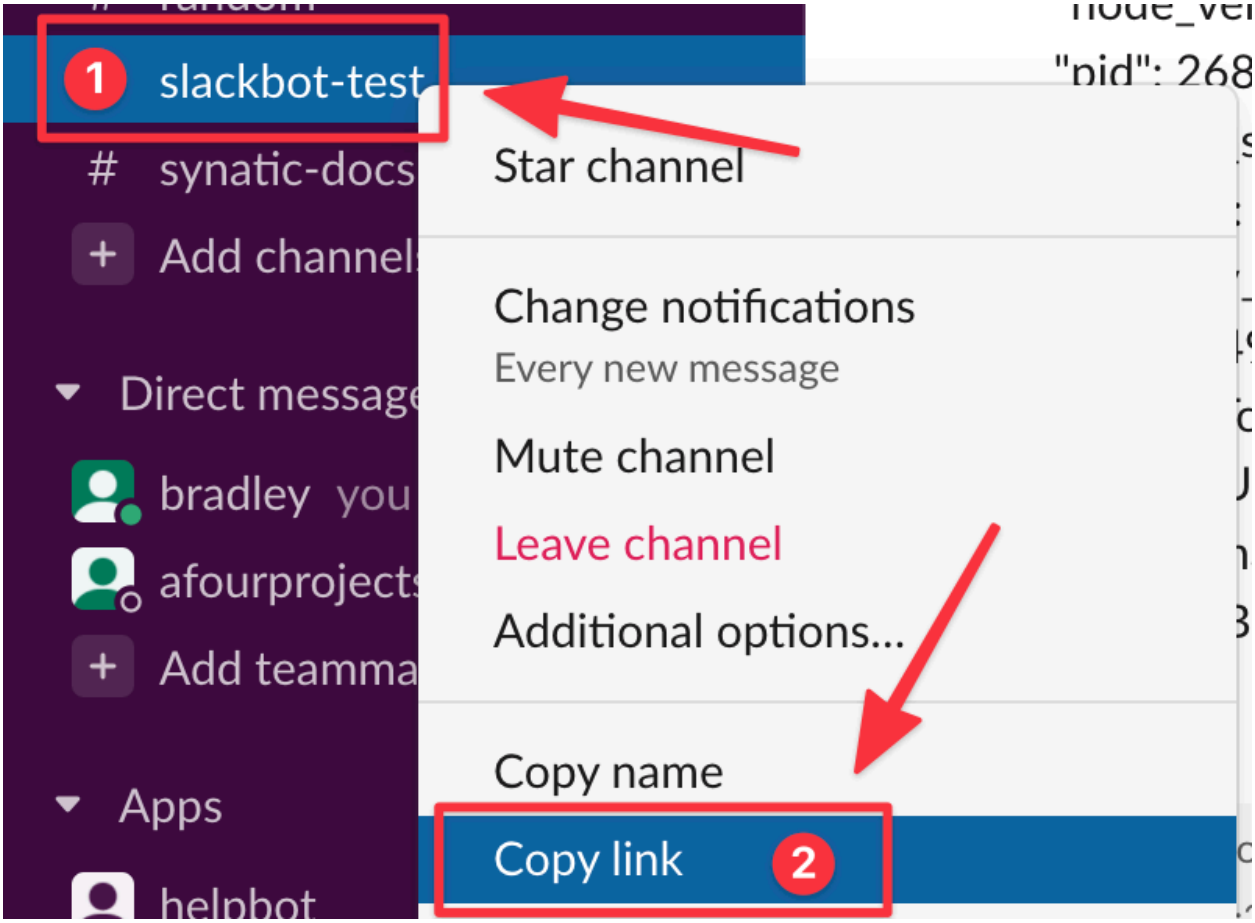
[154] https://api.slack.com/messaging/sending
[155] https://nodejs.org/api/process.html#process_process_env

token. Add a new environment variable with name `SLACK_BOT_TOKEN` and set the value to the token copied from Slack.




**add token environment variable**

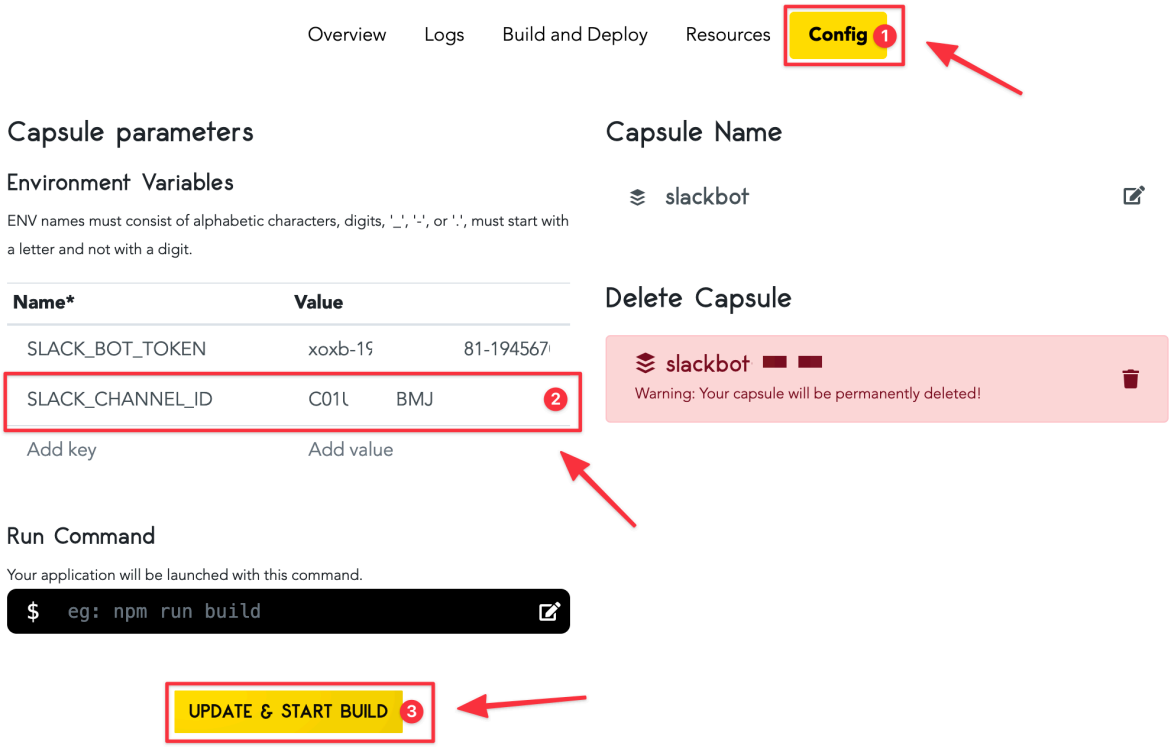
Now that we've added our access token, we need to find the ID of the channel we want to post to. Find a channel on your Slack workspace that you want to send to, or create a new channel. Now we can get the channel ID by right-clicking on the channel name to bring up a context menu. Now, we can choose "Copy Link" from that menu:




**copy channel link from Slack**

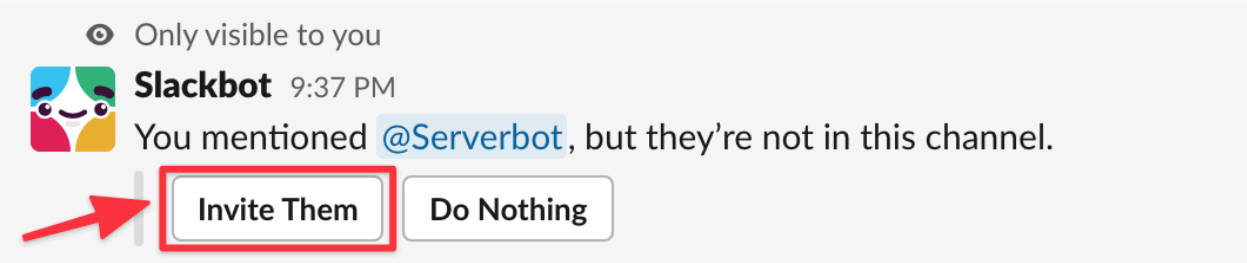
If we paste that link, we get something like `https://<workspace-name>.slack.com/archives/C01SZ6Z3TCY`. The last part of that URL is the channel ID; in this example case, `C01SZ6Z3TCY`.

Let's add this to our environment variables as well, as it keeps all the configurations in one place. Head back over to your Capsule, and add in an environment variable with the name `SLACK_CHANNEL_ID` and set the value to the channel ID we extracted above. Click the "Update & Start Build" button to save the changes to the environment variables.




**add slack channel ID environment variable**

We also need to invite our bot to the chosen channel, so that it will be able to post there. Go to the channel, and @ mention the name you gave the bot to add it. Click "Invite Them" when Slack prompts you.




**invite bot to channel**

Now let's add the code to call Slack on startup, and write a message to our channel. We can modify our boilerplate code above to make the HTTP POST to the endpoint https://slack.com/api/chat.postMessage. We'll use Superagent[156] to make the call.

[156] https://www.npmjs.com/package/superagent

```
const express = require('express');
const superagent = require('superagent');

const app = express();

let port = process.env.PORT || 3000;
app.listen(port, ()=>{
  console.log(`App listening on port ${port}`);
  sendStartupMessageToSlack();
});

function sendStartupMessageToSlack(){
    superagent
      .post('https://slack.com/api/chat.postMessage')
      .send({
        channel:process.env.SLACK_CHANNEL_ID,
        text:"I'm alive and running"
      })
      .set('accept', 'json')
      .set('Authorization', 'Bearer '+ process.env.SLACK_BOT_TOKEN)
      .end((err, result) => {
      });
}
```

We've added in a function `sendStartupMessageToSlack` which makes the call out to Slack. Notice that we send the auth token in a header, using `.set('Authorization', 'Bearer '+ process.env.SLACK_BOT_TOKEN)`. The `Authorization` header is a standard HTTP header.

The channel and the message are sent in the body. Feel free to modify the startup message from I'm alive and running to whatever you'd like.

## Deploying to Code Capsules

This seems like a great time to test out our app on Code Capsules. But before we do that, there is one thing we have to do to make it work. We need to tell Code Capsules how to run our app. By default, Code Capsules will call `npm start` after deploying the code. Therefore, we just need to add a `start` script to our `package.json` file in order for our code to be run on Code Capsules.

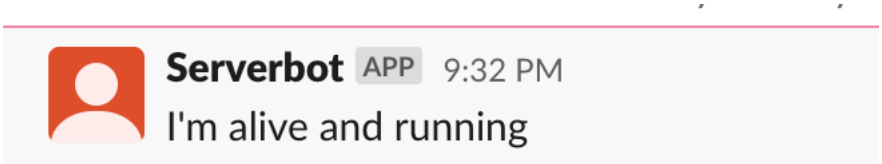
Open the `package.json` file. Under the `scripts` section, add the line `"start": "node index.js"`. The `package.json` file should look like this now:

```
{
  "name": "slackbot",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "superagent": "^6.1.0"
  }
}
```

Ok, let's save all the files we've created, add and commit, and then push to our repo. When Code Capsules sees that there is a new commit, it will automatically deploy our code.

```
git add .
git commit -am 'added code to call Slack on startup'
git push origin
```

If all goes well, in a few minutes you should get a message on your Slack channel from your code!




**startup message**

## Adding a slash command

Now that our app can send us messages, can we send messages back to it? Let's implement the slash command, which will allow us to ask our app for some of its important stats and info. This time, Slack will send an HTTP POST to our app. If we take a look at the Slack docs again[157], we notice that Slack will send the slash command instruction to the URL we specified in the command set up earlier.

[157] https://api.slack.com/interactivity/slash-commands#app_command_handling

We can also see that the POST payload is in the format `application/x-www-form-urlencoded`[158]. We can set up a `body-parser`[159] to interpret this data.

Let's extend our code with the snippet below to implement the slash command receiver as specified in the Slack docs. First add a require statement for `body-parser` at the top.

```
const bodyParser = require("body-parser");
```

Then add the code below:

```
app.use(bodyParser.urlencoded());

app.post('/slack/command/stats', [function(req,res){
  const slackReqObj = req.body;
  const packageJson = require('./package.json');

  const current_time = new Date();
  const stats = {
    name: packageJson.name,
    version: packageJson.version,
    environment: process.env.NODE_ENV,
    platform: process.platform,
    architecture: process.arch,
    node_version: process.version,
    pid: process.pid,
    current_server_time: current_time.toString(),
    uptime: process.uptime(),
    memory_usage: process.memoryUsage()
  };

  const response = {
    response_type: 'in_channel',
    channel: slackReqObj.channel_id,
    text: JSON.stringify(stats, null, '\t')
  };

  return res.json(response);
}]);
```

This code listens for incoming POST calls on the line `app.post('/slack/command/stats', [function(req,res){`. If we receive one, we build up a return object, consisting of various

[158] https://www.w3schools.com/html/html_urlencode.asp
[159] https://github.com/expressjs/body-parser/tree/1.19.0#bodyparserurlencodedoptions

interesting stats and info. This includes the current time on the server (in case it is in a different time zone to us), the name and version of our app as set in the `package.json` file, and various environment and process info.

Then it replies to the request in the format specified by Slack in their docs. We use the line `text: JSON.stringify(stats, null, '\t')` to turn our info and stats object into a nicely formatted text string, in the style of a JSON object.

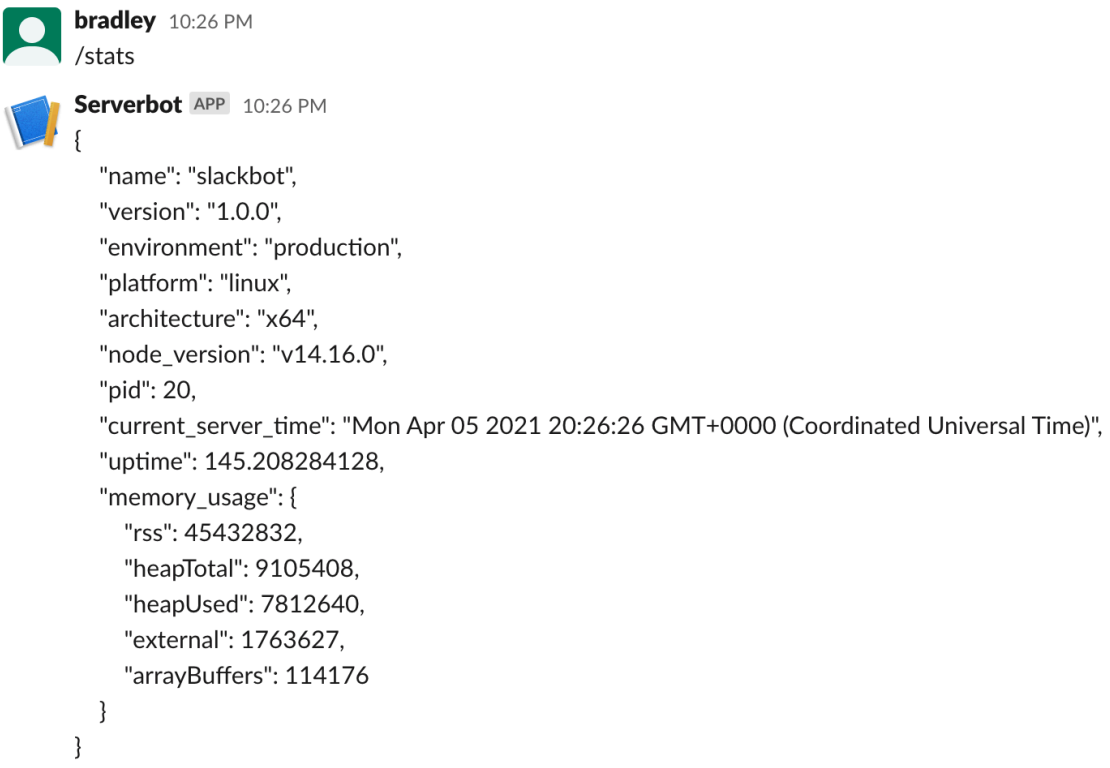
Then, in the line `return res.json(response);`, we return all the info back to Slack to display as the response to a matching slash command.

Great, now we can commit and push this code.

```
1 git commit -am 'added handler for slash command'
2 git push origin
```

After the code has finished deploying on Code Capsules (it should send a startup message again when it's ready), we can test the slash command.

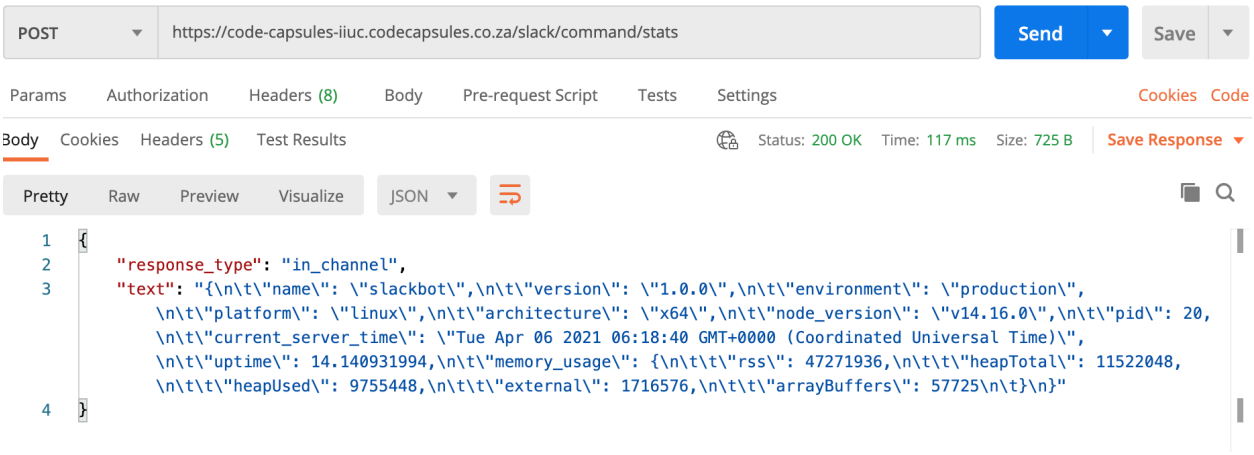
Type `/stats` in the channel we chose earlier. After a second or two, the app should respond with its current vital stats and information.




**testing the slash command**

## Adding verification

We can ask our app via Slack (which we use constantly!) how it's doing; pretty cool, huh? There is a problem though. If we call our slash command endpoint from anywhere else, for instance if we just call it using Postman[160], it also returns all the information and stats! This would not be good for a production system, as sensitive information will be easily found by attackers.




**insecure reply from server to any request**

So how can we ensure that the request comes from our Slack workspace? Luckily, Slack has thought about this, and sends a message signature with its requests[161]. From the guide in Slack's docs[162], we can put together some code to check that the request is legitimately from Slack. The main parts of the check, copied from the docs, looks like this:

- Retrieve the X-Slack-Request-Timestamp header on the HTTP request, and the body of the request.
- Concatenate the version number, the timestamp, and the body of the request to form a basestring. Use a colon as the delimiter between the three elements. For example, v0:123456789:command=/weather&text=94070. The version number right now is always v0.
- With the help of HMAC SHA256 implemented in your favorite programming language, hash the above basestring, using the Slack Signing Secret as the key.
- Compare this computed signature to the X-Slack-Signature header on the request.

We can also check the timestamp to ensure that it is not a replay attack[163] of a message from long ago.

Ok, let's implement this in our project. First, we somehow need to access the raw body of the request, before it has been parsed by `body-parser`. This is to ensure that the signing hash we calculate is

[160] https://www.postman.com
[161] https://api.slack.com/authentication/verifying-requests-from-slack
[162] https://api.slack.com/authentication/verifying-requests-from-slack#verifying-requests-from-slack-using-signing-secrets__a-recipe-for-security__step-by-step-walk-through-for-validating-a-request
[163] https://en.wikipedia.org/wiki/Replay_attack

using the same data that Slack did. After parsing, there could be extra characters and formatting etc. Luckily, the body parser package has a verify option[164], which passes a binary buffer of the raw body request to a user defined function. Let's make a function that conforms to the specs given by `body-parser`. Add this code to your `index.js` file:

```
var rawBodySaver = function (req, res, buf, encoding) {
  if (buf && buf.length) {
    req.rawBody = buf.toString(encoding || 'utf8');
  }
}
```

In this function, we grab the bit stream buffer `buf`, and check that it is not null and that it is not empty (by checking that it has a length). Then we tack it onto the request `req` as a new property `rawBody`. We also convert the buffer to a string, using the encoding supplied, or fall back to `utf8` as a default. Now that the `rawBody` is added to the request, it will be available for use by subsequent middleware. We can add it to the body parser by modifying the code where we add the body parser to the app.

```
app.use(bodyParser.urlencoded({ verify: rawBodySaver}));
```

In the code above, we added options to our body parser initialisation. We set the `verify` option to the method we added above.

Now, let's make a new middleware function[165] to calculate the signature and compare it. We'll be able to call this middleware before our current code for responding to our Slack slash command. Making it a middleware function will also allow us to easily re-use it on other routes, if we want to add more slash commands, or other commands from Slack in the future. We'll make a new file to hold this code. We'll call it `signing.js`.

In the new file, let's add this code:

```
const crypto = require('crypto');

function checkSlackMessageSignature(req, res, next){
    const timestamp = req.headers['x-slack-request-timestamp'];
    const fiveMinutesAgo = Math.floor(Date.now() / 1000) - (60 * 5);

    if (timestamp < fiveMinutesAgo) {
        return res.sendFail(401, "mismatched timestamp");
    }

```

[164] https://github.com/expressjs/body-parser#verify-3
[165] http://expressjs.com/en/guide/writing-middleware.html

```
    const signing_secret = process.env.SLACK_SIGNING_SECRET;

    const slack_signature = req.headers['x-slack-signature'];
    const [version, slack_hash] = slack_signature.split('=');

    const sig_basestring = version + ':' + timestamp + ':' + req.rawBody;
    const hmac = crypto.createHmac('sha256', signing_secret);
    hmac.update(sig_basestring);
    const our_hash = hmac.digest('hex');

    if (crypto.timingSafeEqual(Buffer.from(slack_hash), Buffer.from(our_hash))) {
        return next();
    }
    else {
        return res.send(401, "Invalid request signature");
    }
}

module.exports = checkSlackMessageSignature;
```

Let's take a look at this code. Firstly, we import the crypto (cryptography) library[166]. We don't need to install this as a package, as it is built into Node.js. This library will allow us to perform the hash[167] of the basestring to compare with the signature.

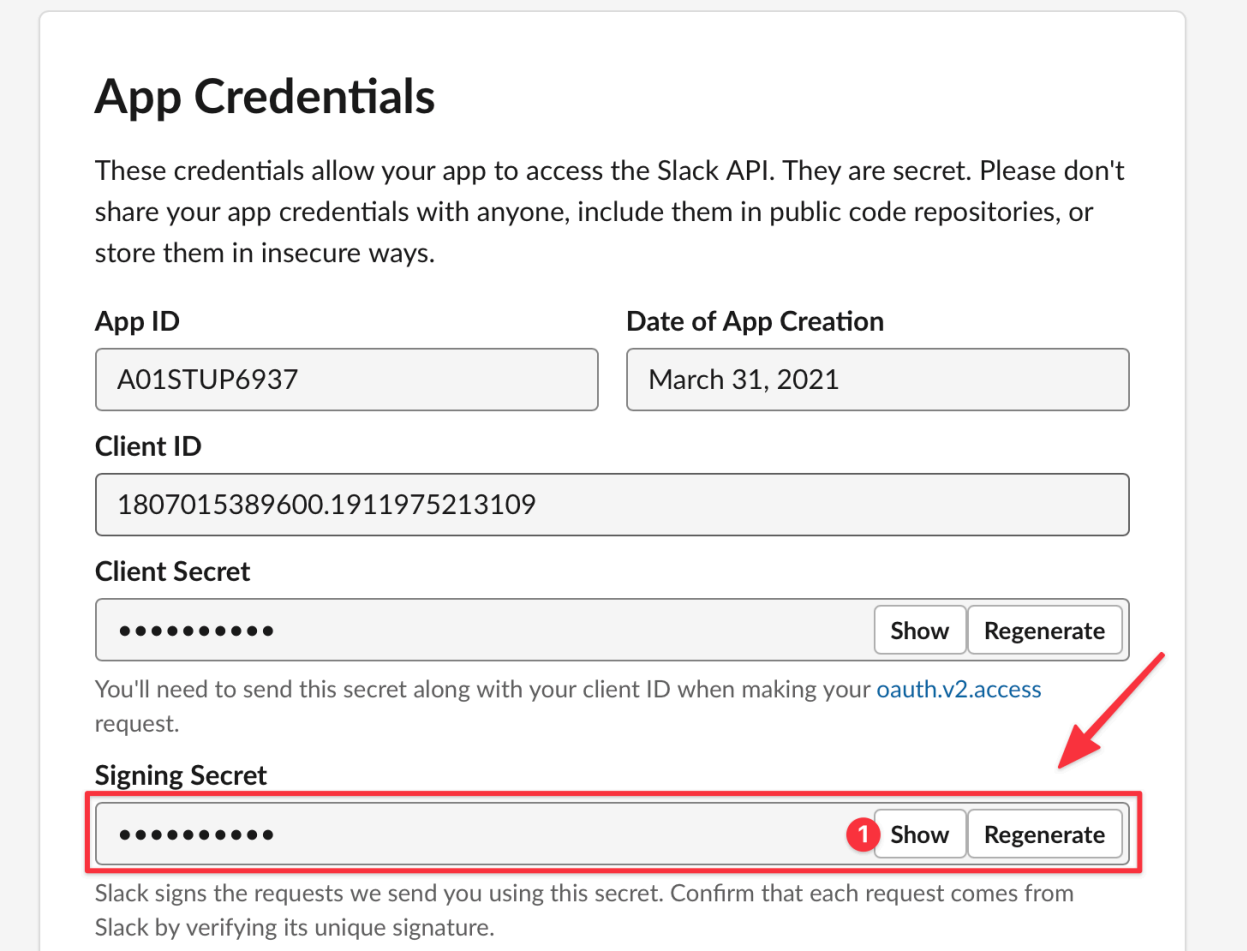
Next, we create a function, with the standard Express middleware parameters[168]:

- `req`, representing the request data.
- `res`, representing an output object that we return results to the user via.
- `next`, representing a function to call if we want to hand control to the next middleware function in the chain. It can also be used to pass an error object back up if something goes wrong processing the request.

Then, on the first few lines of the function, we get the timestamp Slack sends from the request headers, and check that it is within the last few minutes. Note the name of the header is all in lowercase, even though Slack specifies that the header is capitalised. This is because Express converts all header keys to lowercase when serving a request.

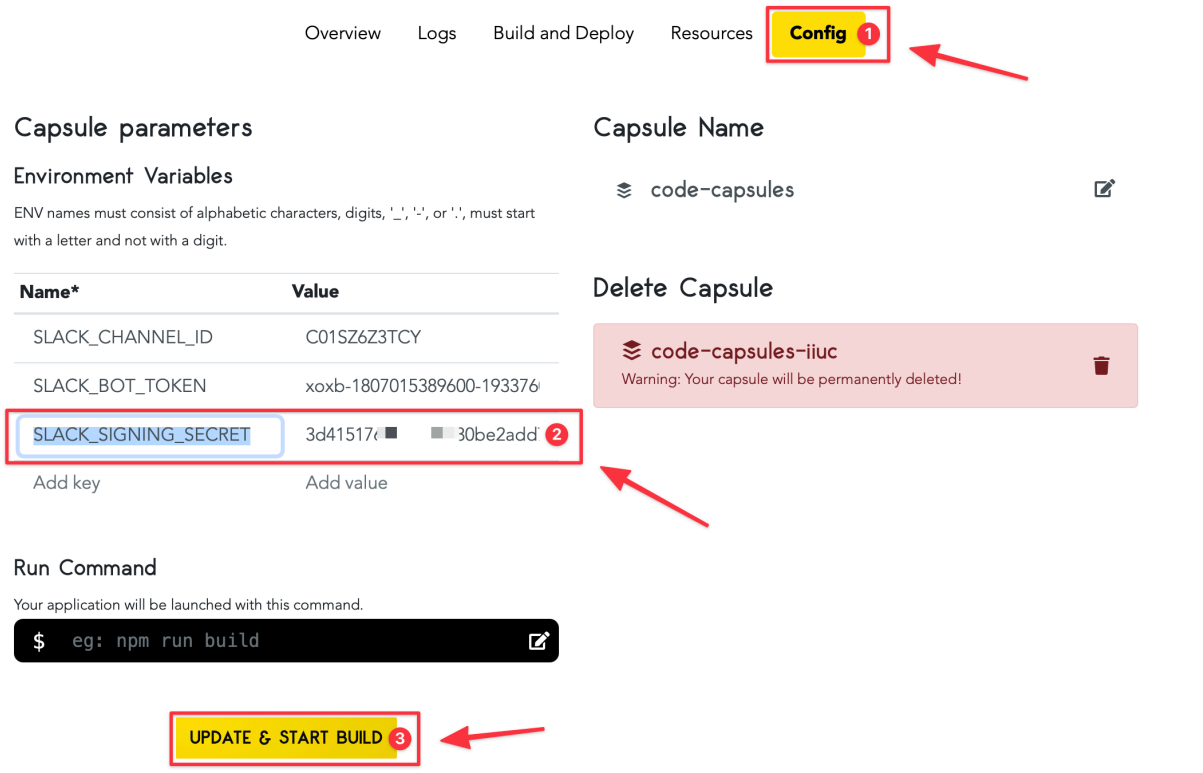
After that, we retrieve the Slack Signing Secret from our environment variables. Let's get our Signing Secret from Slack and add it to the Code Capsules environment now. Head over to your Slack app dashboard, and click on "Basic Information" in the left-hand sidebar. Then scroll down to App Credentials, and look for the Signing Secret. Click "Show", and copy the secret.

[166] https://nodejs.org/api/crypto.html#crypto_crypto
[167] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms
[168] http://expressjs.com/en/guide/writing-middleware.html




**copying signing secret from slack**

Now head over to your Capsule on Code Capsules, and click on the Config tab. Add a new environment variable with Name `SLACK_SIGNING_SECRET` and paste in the value of the Signing Secret we copied above. Click "Update & Start Build" to save the changes.




**setting signing env variable on Code Capsules**

Ok, back to the function. After we retrieve the signing secret from the environment variables, we read out the hash calculated and sent by Slack from the headers using `const slack_signature = req.headers['x-slack-signature']`. This will be a string that looks something like `v0=xxxxxxxxxxxxxxxxxxxxxxxx`, where the `xxxx` represents the actual hash value. We need to split the version identifier `v0` from the beginning of the string though, as this is not part of the hash value. We do this in the next line, `const [version, slack_hash] = slack_signature.split('=')`. Now we have both the version, and the hash string in variables that we can access.

After this, we construct our basestring, made from the version we extracted above, the timestamp of the request, and the `rawBody` (which we extracted in our body parser `verify` function earlier).

The next two lines are where we actually calculate the hash. First, we set up the `crypto` module with our crypto algorithm type SHA256[169], and with our unique Signing Secret. This allows us to then create an HMAC – or Hash Based Message Authentication code[170], which is the fancy name for the message signature. We then use the `update` method on our newly created HMAC to load in our basestring that we constructed above.

Now that the crypto HMAC is primed with all the info it needs, we can call the `digest` function to actually calculate the hash. We pass in as a parameter `hex` to indicate that we want the result back

[169] https://en.wikipedia.org/wiki/SHA-2
[170] https://en.wikipedia.org/wiki/HMAC

in hexadecimal format[171], as this is the same format that Slack sends their calculated hash value in.

Great, so now we have Slack's signature hash, and our hash. We need to check that they are the same, which will prove that the message was legitimately sent by Slack. We could just use a normal string compare, i.e. `if (slack_hash === our_hash)`, but there is a slight security issue with this, known as a timing attack[172]. This type of attack is based on the knowledge that a normal string compare function takes a different amount of time to compare two strings, depending on how close the strings are to each other. An attacker can take advantage of this timing difference to repeatedly send messages and, based on the time for our server to respond, can guess at how close their hash is to what we are expecting. With much patience and many thousands of messages, an attacker could eventually guess our Signing Secret, compromising all our checks.

Luckily, there is a simple way to protect from this, and it's built right into the `crypto` library. This is where we call `crypto.timingSafeEqual`. This compare always returns in the same amount of time, regardless of how close the hashes are to each other. Therefore, we don't give any extra information away to would-be attackers.

Now, if the hashes are equal, from our `timingSafeEqual` test, we just call `return next()` which exits our function and passes control to the next middleware function (which will be our slash command handler).

If the hashes are not equal, then we know this request is not genuinely from Slack, so we can end early and send a `401`, which is a standard HTTP code[173] for `Unauthorized`. Basically, we boot the imposter out.

Now, the last line in this file is `module.exports = checkSlackMessageSignature`. This allows our middleware function to be visible to other modules that import this file.

Ok, now that we've got this middleware created, let's link it to our slash command handler. Head on back to the `index.js` file, and import the middleware function by adding this line near the top of the file:

```
1 const checkSlackMessageSignature = require('./signing');
```

Now, we can navigate to our slack command handler, which started like this: `app.post('/slack/command/stats'`. Modify that to include a call to the message signature check before the actual handler, like this:

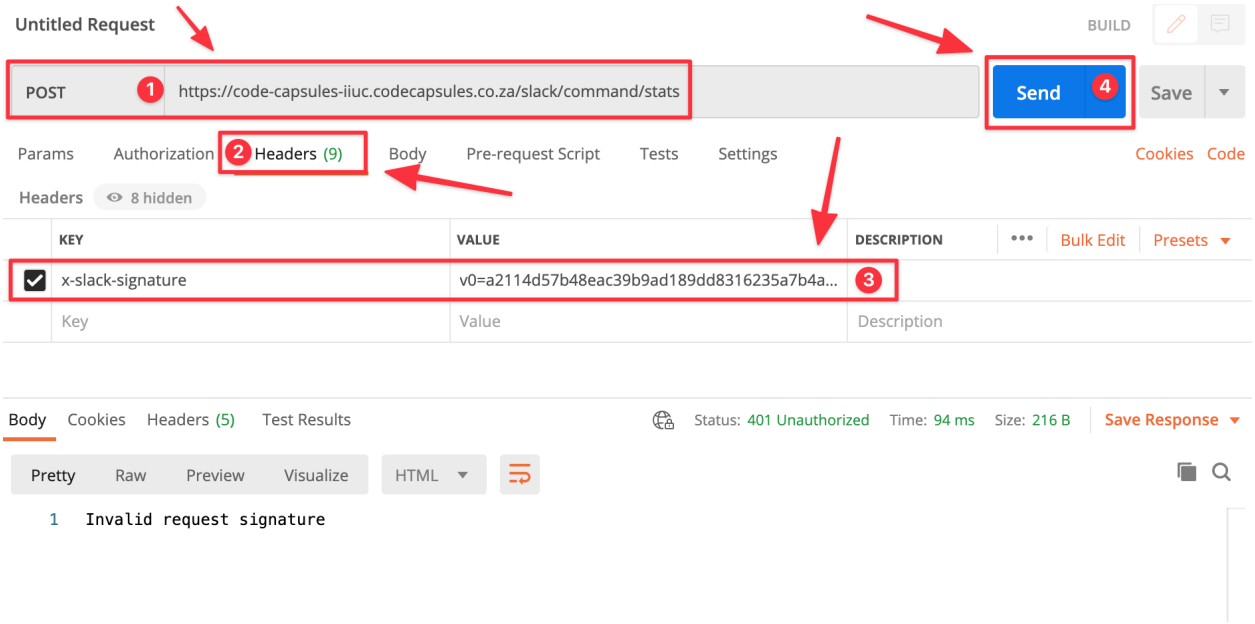
```
1 app.post('/slack/command/stats', [checkSlackMessageSignature, function(req,res){
```

Fantastic, now our app is secure. You can commit all the changes, and push it up to Git, which will kick off our final deploy to Code Capsules:

[171] https://en.wikipedia.org/wiki/Hexadecimal
[172] https://codahale.com/a-lesson-in-timing-attacks/
[173] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#4xx_client_errors

```
git add .
git commit -am 'added message signature checking'
git push origin
```

Once the code is up and running on Code Capsules, test it out to see that it still responds to the Slack slash command. Then you can try again from Postman or other similar apps, and see that it will not send any info without a valid signature (you can use `v0=a2114d57b48eac39b9ad189dd8316235a7b4a8d21a10bd27519666489c69b503` as an example `x-slack-signature` parameter):




**returning 401 for invalid signature**

# Things to Try Next

What else can we do? It's almost endless!

- Add this code to an existing app you have built to get easy info straight from Slack!
- Add in more slash commands for more info – for example, you could get current user count on your app, number of database records etc. Basically, any information you could need for dev ops[174].
- Look at some of the other functionality Slack offers for integration; for example, using modals[175], or listening in for keywords in messages[176].

[174] https://en.wikipedia.org/wiki/DevOps
[175] https://api.slack.com/surfaces/modals
[176] https://api.slack.com/messaging/managing